Static Binary Rewriting to Improve Software Security, Safety and Reliability

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

William H. Hawkins III

May 2018

© 2018 William H. Hawkins III. This dissertation is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Abstract

The notion that software runs the modern world is generally accepted. Humans interact with software to conduct most of the tasks in their day-to-day lives: they talk to friends and family on smart phones, make purchases on the Internet and follow their GPS device to get from the airport to the hotel when they travel. However, it is when people rely on software without realizing it that the stakes are the highest. Today, software controls the power grid, dams, airplanes, cars, surgery, the economy, national defense and so on.

Disruption of safety-critical systems, whether caused by the malicious activity of an attacker, the nature of the complexity of the system, operator error, etc., can cost humans their lives or their livelihoods. For this reason, the security, safety and reliability of the software that run these systems is of the utmost importance. The public would greatly benefit from a general tool that can be used to improve their security, safety and reliability.

There are a number of reasons, though, such a tool is difficult to find, much less build. First, such a tool must operate on software without access to its source code or metadata, known as software of unknown provenance (SOUP). Second, such a tool must be able to rewrite software that operates on a variety of platforms. Third, the tool must provide the system designer the flexibility to design imaginative security and reliability improvements without restriction. Fourth, the architecture and algorithms of the static binary rewriter must produce transformed, rewritten programs that adhere to a variety of performance parameters.

The design, architecture and algorithms of the retargetable, static binary rewriter presented in this dissertation give system designers the power to apply post hoc transformations that improve the security and reliability of SOUP. The evaluation presented in this dissertation demonstrate that statically rewritten SOUP exhibits no excessive size and performance overhead with respect to the original program/library. Finally, this dissertation presents the results of three case studies that demonstrate the utility of the design and architecture of the static binary rewriter to allow third-party developers to build transformations that add security and reliability to potentially vulnerable software.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

William H. Hawkins III

This dissertation has been read and approved by the Examining Committee:

Jack Davidson, Adviser

James Lambert

David Melski

Baishaki Ray

Westley Weimer, Chair

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

May 2018

Dedication

I have never accomplished anything on my own. I do not believe I will *ever* accomplish anything on my own. I have always had help from friends, family, colleagues and God. This work is dedicated to anyone who has ever been my ally and to those who will help me in the future. I hope that I can repay you one day.

Contents

Contents v
List of Tables ix
List of Figures xi

I Introduction 1

1 Introduction 2

2 Motivation 5

II Fundamentals 9

3 Correctness 10
3.1 Correctness of a Statically Rewritten Program/Library 10
3.2 Prerequisites to Meet Correctness Definition 11
3.3 Parallels to the Definition of Correctness in Other Fields 13
3.4 Discussion 14
3.5 Future Work 15
3.6 Conclusion 15

4 Design and Architecture of a Static Binary Rewriter 16
4.1 Introduction 16
4.2 IR Construction 18
4.2.1 Disassembly 18
4.2.2 CFG Reconstruction 23
4.3 Transformations 28
4.3.1 Transformation Architecture 28
4.3.2 Mandatory Transformations 28
4.3.3 User-specified Transformations 29
4.4 Reassembly 31
4.4.1 Initial Link Placement 33
4.4.2 Handling Dense Pinned Addresses 38
4.4.3 Link Expansion and Chaining 39
4.4.4 Link Resolution and Dollop Actualization 40
4.4.5 Example 42
4.4.6 Actualization and the Size of Statically Rewritten Program/Library 43

5 Evaluation 47
5.1 Experimental Bias 47
5.2 Evaluation Platform Specifications 48
5.3 Validation Experiments 49

5.4 Time to Rewrite . . . 53
5.5 SPEC . . . 56
5.5.1 Performance . . . 58
5.5.2 Memory Usage . . . 62
5.5.3 Instruction Cache Usage . . . 66
5.5.4 Filesize Overhead . . . 67
5.6 CGC . . . 68
5.7 Dense Pinned Addresses and Sled Usage . . . 72
5.8 Conclusion . . . 73

6 Related Work **75**
6.1 Introduction . . . 75
6.2 ATOM . . . 76
6.3 Etch . . . 76
6.4 Vulcan . . . 77
6.5 DIABLO . . . 77
6.6 SecondWrite . . . 77
6.7 Ramblr . . . 78
6.8 UROBOROS . . . 81
6.9 Conclusion . . . 83

III Optimizations **84**

7 Locality Layout **86**
7.1 Introduction . . . 86
7.2 The Problem . . . 87
7.3 The Solution . . . 89
7.4 Evaluation . . . 96
7.4.1 SPEC . . . 97
7.4.2 CGC . . . 98
7.5 Conclusion . . . 98

8 Profile Layout **99**
8.1 Introduction . . . 99
8.2 The Problem . . . 99
8.3 The Solution . . . 101
8.3.1 Profiles . . . 101
8.3.2 Collecting and Storing Profiles . . . 104
8.3.3 Algorithm . . . 104
8.4 The Importance of Profile Data . . . 112
8.5 Evaluation . . . 117
8.5.1 Performance . . . 118
8.5.2 Memory Usage . . . 119
8.5.3 Instruction Cache Usage . . . 121
8.5.4 Filesize Overhead . . . 122
8.5.5 Placement Ratios . . . 122
8.5.6 CGC . . . 123
8.6 Related Work . . . 125
8.7 Conclusion . . . 130

9 Relax Layout 131
9.1 Introduction . . . 131
9.2 Savings . . . 132
9.3 Algorithm . . . 132
9.3.1 The Modified Profile Layout Algorithm . . . 134
9.4 Evaluation . . . 140
9.4.1 Performance . . . 140
9.4.2 Memory Usage . . . 141
9.4.3 Instruction Cache Usage . . . 142
9.4.4 Filesize Overhead . . . 143
9.4.5 Relax Ratios . . . 144
9.4.6 CGC . . . 145
9.4.7 Discussion . . . 148
9.5 Related Work . . . 150
9.6 Conclusion . . . 153

IV Applications 154

10 Mixr 156
10.1 Introduction . . . 156
10.2 Motivation . . . 157
10.2.1 Threat Model and Defenses . . . 160
10.2.2 Design . . . 162
10.3 Operation . . . 163
10.3.1 Metadata . . . 163
10.3.2 Rerandomization . . . 167
10.4 Alternate Design and Implementation . . . 169
10.5 Results . . . 172
10.5.1 Original Design and Implementation . . . 172
10.5.2 Alternate Design and Implementation . . . 175
10.6 Related Work . . . 177
10.6.1 Remix . . . 177
10.6.2 RuntimeASLR . . . 180
10.6.3 Timely Address Space Randomization . . . 181
10.6.4 Marlin . . . 182
10.6.5 Binary Stirring . . . 184
10.6.6 CodeArmor . . . 185
10.6.7 STABILIZER . . . 187
10.6.8 Chronomorph . . . 188
10.6.9 Shuffler . . . 189
10.6.10 librando . . . 191
10.6.11 selfrando . . . 193
10.7 Conclusion . . . 194

11 Dynamic Canary Randomization 196
11.1 Introduction . . . 196
11.2 Design . . . 197
11.3 Implementation . . . 199
11.4 Evaluation . . . 200
11.4.1 Steady-State Performance Overhead . . . 200
11.4.2 Case Study: Rerandomizing Canaries in bzip2 . . . 202
11.4.3 DCR and BROP . . . 205
11.5 Future Work . . . 206

11.6 Conclusion . . . 207

12 Control-Flow Integrity **208**
12.1 Introduction . . . 208
12.2 Design . . . 209
12.2.1 Analysis . . . 209
12.2.2 Instrumentation . . . 217
12.3 Related Work . . . 226
12.3.1 BinCFI . . . 228
12.3.2 BinCC . . . 231
12.3.3 Opaque CFI . . . 231
12.3.4 Object Flow Integrity . . . 232
12.4 Evaluation . . . 233
12.4.1 Security . . . 233
12.4.2 Performance . . . 235
12.4.3 CGC . . . 247
12.5 Conclusion . . . 260

13 Conclusion **262**

Appendices **269**

A Locality Layout Algorithm Evaluation **270**
A.1 SPEC . . . 270
A.2 Memory Usage . . . 271
A.3 Instruction Cache Usage . . . 273
A.4 Filesize Overhead . . . 274
A.5 Placement Ratios . . . 275
A.6 CGC . . . 276

Bibliography **282**

List of Tables

4.1 The result of linear disassembly (in the right column) on a set of bytes (in the left column). . 19

5.1 Success and failure of the coreutils unit tests when run against the Zipr-written version of the original tools compiled with different optimization levels. No matter what optimization level was used, the Zipr-rewritten version passed the expected number of unit tests. 52
5.2 Times to statically rewrite programs/libraries of different sizes after transformation with either the Null or P1 User-specified Transformations using the architecture and algorithms described to this point in the dissertation as implemented in a prototype. All times are in seconds and all sizes are in kilobytes. 54
5.3 A pair of semantically equivalent programs. The only difference is whether the program invokes `function` with a `call` or a combination of `push`/`jmp`. The difference has performance implications. 60
5.4 Distribution of dominators for the RCBs in the CGC dataset. 73

6.1 Timeline of introduction of related work. 76
6.2 UROBOROS's assumptions that Ramblr authors believe do not hold for a non-trivial number of real-world programs. 79

8.1 Distribution of dominators for the RCBs in the CGC dataset generated using the Profile Layout algorithm. 124

9.1 Distribution of dominators for the RCBs in the CGC dataset generated using the Relax Layout algorithm. 146

10.1 Comparison of requirements and features of each of the available runtime rerandomization technologies. 178
10.2 Average basic block and function sizes, in bytes, for the programs in the SPEC benchmark suite when compiled with different optimization levels using gcc version 4.8.4. 179
10.3 Average code sizes per bundle in the Mixr-prepared versions of the programs in the SPEC benchmark suite with different granularities. 179

11.1 Results of BROP attacks against software protected with traditional canaries (SSP) and with Dynamic Canary Randomization (DCR) 206

12.1 Invoking a function conditionally and the idiomatic object code emitted by the compiler. . . . 210
12.2 A switch statement implemented in C and the idiomatic object code emitted by the compiler. 211
12.3 Sources of Hell Nodes. 214
12.4 An example of two indirect program control flow operations whose targets are not identical but share some targets in common. 218
12.5 Enforcement instrumentation for CFI. 220
12.6 Methods for instrumenting for safety the return indirection program control operation from a function statically deemed safe. 221

A.1 Distribution of dominators for the CBs in the CGC dataset when reassembled using the Locality Layout algorithm. 277

List of Figures

4.1 The rewriting pipeline. 17
4.2 Pinned addresses, transformations and links. 24
4.3 Discovery of pinned address from a typical jump table implementation. Instructions at pinned addresses are highlighted in red. 25
4.4 Discovery of pinned address from function calls. Instructions at pinned addresses are highlighted in red. 26
4.5 The internal state of the reassembly process as it produces a statically rewritten binary program/library. 33
4.6 The algorithms of the Reassembly phase are one cause of overhead in the statically rewritten program/library. Shown here is the overhead of the algorithms of the Reassembly phase from links at pinned addresses and dollop splitting and the overhead from User-specified transformations. 45

5.1 The relationship between the size of the input program/library and the time it takes to statically rewrite using the Zipr prototype implementation. Data for programs rewritten after transformation by both the Null and P1 Transformations are included in the graph. Each axis is logarithmically scaled. 55
5.2 The relationship between the number of dollops in the input program/library and the time it takes to statically rewrite using the Zipr prototype implementation. Data for programs rewritten after transformation by both the Null and P1 Transformations are included in the graph. Each axis is logarithmically scaled. 56
5.3 Performance overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A. 58
5.4 Performance overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B. 59
5.5 There is no correlation between performance and the common causes of performance overhead identified by the investigation into the `perlbench`, `povray`, `xalancbmk` and `gobmk` applications in the SPEC2006 benchmark suite. 62
5.6 Maximum RSS overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A. 63
5.7 Maximum RSS overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B. 63
5.8 The minor page fault overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host A. 64
5.9 The minor page fault overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host B. 65
5.10 The instruction cache miss overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host A. 67
5.11 The instruction cache miss overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host B. 68
5.12 Filesize overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A. 69

5.13 Filesize overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B. 69
5.14 Availability scores for the RCBs in the CGC dataset when reassembled using the default algorithms of the Reassembly phase. 73

7.1 How the basic algorithms of the Reassembly phase cause excessive runtime memory usage because it does not respect program locality. 88
7.2 An example of the Locality Layout algorithm reassembling a program to take advantage of program locality. Compare the reassembly with the reassembly in Figure 7.1 91
7.3 An example of how the size and density of links on a single page affects whether links and target dollops can fit on the same page. 92
7.4 An example of how the size and density of links on the same page limits the ability to place dollops in the gaps between pinned addresses. 93
7.5 An example of how a user transformation to a dollop may make it so that a link and its target dollop cannot fit on the same page. 94
7.6 An example of how the Overwriting Pinned Addresses extension results in a smaller maximum RSS and fewer page faults. 95
7.7 An example of how a user transformation that extends a dollop makes it impossible to apply the Overwriting Pinned Addresses extension. 96

8.1 An example of a case where the Locality Layout algorithm places a dollop on the same page in the statically rewritten binary that it was on in the original program which results in excessive paging. 100
8.2 An example of a case where the Locality Layout algorithm cannot take advantage of hints from the original program's CFG to place the *hot* connected dollops on the same page. 102
8.3 An example of how information from a profile guides the Profile Placement algorithm in its decision of where to place dollops. 108
8.4 Comparison of availability scores for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. 114
8.5 Comparison of the percentage of time the target dollop could be placed on the winning bidder's preferred page for each 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. . 115
8.6 Comparison of execution overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph. 116
8.7 Comparison of memory overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph. 117

8.8 Comparison of filesize overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph. . . . 118
8.9 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 119
8.10 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 120
8.11 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 121
8.12 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 122
8.13 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 123
8.14 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 124
8.15 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm without profiles. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 125
8.16 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm without profiles. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 126
8.17 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. . . . 127

8.18 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green. 128

8.19 The percentage of the time target dollop could be placed on the winning bidder's preferred page. Host A's placement ratios are shown in light blue. Host B's placement ratios are shown in dark blue. 129

8.20 Availability scores for the RCBs in the CGC dataset when rewritten using the Profile Layout algorithm. 129

9.1 An example showing why the Relax Layout algorithm is not a simple extension of the Profile Layout optimization. Because the Relax Layout algorithm potentially changes the size of the implementation of links, the hidden assumption of the other layout algorithms that the dollop size is constant is violated. 133

9.2 The Relax Layout algorithm *drapes* the dollops over the memory space of the statically rewritten binary and accommodates its *gaps*. 134

9.3 Coalescing gaps can have a cascading effect on the size of the program control instructions needed to jump over them. 137

9.4 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 141

9.5 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 142

9.6 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 143

9.7 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 144

9.8 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 145

9.9 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. 146

9.10 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm (without profiles) are shown on the left, in light blue and light green. . . . 147
9.11 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm (without profiles) are shown on the left, in light blue and light green. . . . 148
9.12 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. . . . 149
9.13 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green. . . . 150
9.14 The percentage of the time that an unconstrained link can be replaced with a constrained link in the process of statically rewriting the applications of the SPEC benchmark suite. Host A's relaxation ratios are shown in light blue. Host B's relaxation ratios are shown in dark blue. . . 151
9.15 Availability scores for the RCBs in the CGC dataset when rewritten using the Relax Layout algorithm. . . . 151
9.16 An example of the way that a single relaxation of the implementation of a link on a platform with variable-length instructions can cause a dollop to slide to a different page. . . . 152

10.1 A program with a vulnerable function operating normally and after being hijacked. This program has three ROP *gadgets* that, when used together in a ROP *program*, can be used to print "Hello" to the screen. . . . 158
10.2 A Mixr-prepared program at startup. . . . 164
10.3 Creation of dollop bundles from straight line code in the original program. . . . 165
10.4 The links between the metadata components of a Mixr-prepared program. . . . 166
10.5 A Mixr-prepared program after a single rerandomization. . . . 169
10.6 Mixr runtime overhead for the programs of the SPEC2006 benchmark suite. . . . 172
10.7 Mixr maximum RSS overhead for the programs of the SPEC2006 benchmark suite. . . . 173
10.8 Mixr on-disk file size overhead for the programs of the SPEC2006 benchmark suite. . . . 174
10.9 Rerandomization time plotted against the number of dollops in the Mixr'd program when prepared using the original design and implementation. . . . 175
10.10 Runtime overhead for the programs of the SPEC2006 benchmark suite when Mixr-prepared with the alternate design. . . . 176
10.11 Comparison of the overhead for the programs of the SPEC2006 benchmark suite when Mixr-prepared with the original and alternate design. . . . 176
10.12 Affect of the choice of granularity on the performance overhead of the programs of the SPEC2006 benchmark suite when Mixr-prepared with the alternate design. . . . 177

11.1 All canary values on the stack must be rewritten when the reference canary value is rewritten. 198
11.2 Building the linked list of canary addresses on the stack at runtime. . . . 199
11.3 Overhead of GCC's SSP in SPEC2006 benchmark. Experiment (1) is the baseline (no stack protection). Experiment (2) is selective SSP, and Experiment (3) is full SSP. . . . 202

11.4 Overhead of DCR's steady-state performance in SPEC2006 benchmark. Experiment (4) is static rewriting only. Experiment 5 is selective DCR applied with static rewriting. Experiment (6) is full DCR applied with static rewriting. 202
11.5 Performance impact of rerandomizing canary values at every `memcpy()` and `fwrite()` during an execution of *bzip2* under SPEC2006. 204

12.1 The SelectiveCFI instrumentation of a jump. 222
12.2 The SelectiveCFI instrumentation of a function call. 223
12.3 The contents of the program and return address stacks at runtime. 224
12.4 The traditional method of SelectiveCFI instrumentation of `ret`s in a program. This enforcement method does not take advantage of the return address predictor. 225
12.5 The SelectiveCFI instrumentation of `ret`s in a program using executable nonces. This enforcement method is able to take advantage of the return address predictor. 226
12.6 AIR for the programs of the SPEC benchmark suite protected with the basic implementation of SelectiveCFI. 230
12.7 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host A. The results for the applications protected with the basic version of SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase when transformed with the Null Transformation are shown on the left, in light blue and light green. 237
12.8 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host B. The results for the applications protected with the basic version of SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase when transformed with the Null Transformation are shown on the left, in light blue and light green. 238
12.9 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host A. 239
12.10 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host B. 239
12.11 Percentage of the marginal overhead of programs secured with Basic SelectiveCFI as compared to programs transformed with the Null Transformation attributable to the code required to check the safety of targets. 240
12.12 Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with the basic implementation of SelectiveCFI on Hosts A and B. 241
12.13 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host A. The results for the applications protected with Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green. 242
12.14 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI without checking the safety of `ret` targets in functions statically deemed safe. These results are for Host B. The results for the applications protected with Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green. 243
12.15 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host A. The results for the applications protected with the Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green. . 244
12.16 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host B. The results for the applications protected with the Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green. . 245

12.17 Comparison of the ratios of the functions of the applications of the SPEC benchmark suite statically deemed safe on Hosts A and B. 246
12.18 Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with Safe SelectiveCFI on Hosts A and B. 246
12.19 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green. 247
12.20 Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host B. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green. 248
12.21 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with the SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green. 249
12.22 Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green. 250
12.23 Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with Safe SelectiveCFI on Hosts A and B. 251
12.24 Availability of the Challenge Binaries protected with the Basic Selective CFI implementation. 251
12.25 Execution overhead of the Challenge Binaries protected with the Basic Selective CFI implementation. 252
12.26 Memory overhead of the Challenge Binaries protected with the Basic Selective CFI implementation. 252
12.27 Filesize overhead of the Challenge Binaries protected with the Basic Selective CFI implementation. 253
12.28 Percentage of successful placement of nonces in Challenge Binaries protected with the Basic SelectiveCFI implementation. 253
12.29 Availability of the Challenge Binaries protected with the Safe Selective CFI implementation. 254
12.30 Memory overhead of the Challenge Binaries protected with the Safe Selective CFI implementation. 254
12.31 Filesize overhead of the Challenge Binaries protected with the Safe Selective CFI implementation. 255
12.32 Execution overhead of the Challenge Binaries protected with the Safe Selective CFI implementation. 256
12.33 Percentage of functions in the Challenge Binaries statically deemed safe by the SelectiveCFI analysis. 256
12.34 Percentage of successful placement of nonces in Challenge Binaries protected with the Safe SelectiveCFI implementation. 257
12.35 Availability of the Challenge Binaries protected with the Selective CFI with Coloring implementation. 258
12.36 Memory overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation. 259
12.37 Filesize overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation. 259
12.38 Execution overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation. 260
12.39 Percentage of successful placement of nonces in Challenge Binaries protected with the SelectiveCFI with Coloring implementation. 261

A.1 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 271
A.2 Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 272
A.3 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 273
A.4 Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 274
A.5 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 275
A.6 Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green. 276
A.7 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm without profiles. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green. 277
A.8 Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm without profiles. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green. 278
A.9 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 279
A.10 Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green. 280

A.11 The percentage of dollops that could be placed either on the same page as their link or the same page that they were on in the original program/library. Host A's placement ratios are shown in light blue and light green. Host B's placement ratios are shown in dark blue and dark green. . . . 281
A.12 Availability scores for the RCBs in the CGC dataset when reassembled using the Locality Layout algorithm. . . . 281

List of Algorithms

1 Pseudocode of the algorithm for a linear scan disassembler. 19

2 Pseudocode of the algorithm for a recursive descent disassembler. 21

3 Algorithms of the Reassembly phase, Part I: Main reassembly method 33

3 Algorithms of the Reassembly phase, Part II: Initial link placement and link expansion methods 34

3 Algorithms of the Reassembly phase, Part III: Actualization methods 35

3 Algorithms of the Reassembly phase, Part IV: Support methods 36

3 Algorithms of the Reassembly phase, Part V: Support methods, continued 37

4 Override the default behavior of the Reassembly phase to place a dollop according to the Locality Layout algorithm. 90

5 Override the default behavior of the Reassembly phase to place a dollop according to the Profile Layout algorithm. 105

6 The updated version of the AwardBids function for the placement phase of the Relax Layout algorithm. 135

7 The algorithms to create and coalesce gaps. 135

8 The algorithms to perform the size-efficient placement of dollops around the gaps in the memory space of the statically rewritten program/library. 138

9 Swapping a pair of Dollop Bundles 168

10 Swapping a pair of dollop bundles using the alternate encapsulation technique. 171

Part I

Introduction

Chapter 1

Introduction

The notion that software runs the modern world is generally accepted. Humans interact with software to conduct most of the tasks in their day-to-day lives: they talk to friends and family on smart phones, make purchases on the Internet and follow their GPS device to get from the airport to the hotel when they travel. However, it is when people rely on software without realizing it that the stakes are the highest. Today, software controls the power grid, dams, airplanes, cars, surgery, the economy, national defense and so on.

"Computing systems in which the consequences of failure are very serious are termed safety-critical" [242] . Disruption of safety-critical systems, whether caused by the malicious activity of an attacker, the nature of the complexity of the system, operator error, etc., can cost humans their livelihood or their lives. For this reason, the security, safety and reliability of these systems is of the utmost importance.

The public would greatly benefit from a *general* tool that can be used to improve the security, safety and reliability of these systems. There are a number of reasons, though, such a tool is difficult to find, much less build. First, safety-critical software operates on myriad different hardware platforms. Some of the software operates on state-of-the-art desktops and servers while some runs on low-power embedded devices. Second, these critical systems run on different processors. Some of the software runs on Intel whereas some runs on ARM. Third, safety-critical system developers give their users different levels of visibility into the system. Some of the software is provided with source code while some is proprietary. Finally, the source code for the software may no longer exist or the software itself may have outlived those who developed it.

Given these constraints, a tool that improves the security, safety and reliability of critical software systems must meet a number of requirements. First, it must be able to operate on software that runs on several different operating systems (e.g., Linux, UNIX, Windows, etc.) and hardware platforms (e.g., servers, mainframes, desktops, embedded devices, etc). Second, it must be able to operate on software whose source

code is unavailable or whose provenance is unknown, referred to as software of unknown provenance (SOUP). Third, it must be able to improve software whose operating context strictly limits overhead (memory, time, power).

This dissertation presents the design, the architecture and algorithms for a retargetable, static binary rewriter that gives system designers the power to apply *post hoc* transformations that improve the security and reliability of SOUP without excessive size and performance overhead. These capabilities, in turn, give developers the tools they need to improve the software that drives daily life, irrespective of its provenance or operating context.

The dissertation is divided into three parts. Part I is an argument about the importance of a retargetable, static binary rewriter in today's software-driven world. Part II describes the fundamental design, architecture and algorithms of the retargetable, static binary rewriter. Part III explains a series of optimizations that improve on the fundamental architecture and algorithms to improve the performance of the statically rewritten programs/libraries. Part IV describes three applications of the static binary rewriter for improving the security and reliability of software.

Part I, which contains this section, continues with Chapter 2 that expands on the discussion of this section about the need for a retargetable, static binary rewriter that gives system designers the power to apply *post hoc* transformations that improve the security and reliability of SOUP without excessive size and performance overhead. Chapter 3 formalizes the notion of *correctness* of the architecture and algorithms underlying such a static binary rewriter and presents the theoretical limitations of such a conception.

Part II begins with chapters 4.2, 4.3, and 4.4 that discuss the fundamental architecture and algorithms underlying the static binary rewriter. Chapter 5 presents the results of an evaluation of a prototype implementation of a static binary rewriter based on the architecture and algorithms described in the previous chapters. Part II concludes with chapter 6 which contains a discussion of other static binary rewriters.

Part III contains chapters that describe optimizations to the fundamental architecture and algorithms presented in Part II. Chapters 7, 8 and 9 describe three different algorithms for code layout and evaluate their effect on the performance of a statically rewritten binary. Each chapter also contains an overview of the historical research and related work associated with each of the techniques.

The chapters in Part IV describe three different practical applications for a retargetable, static binary rewriter. Chapter 10 describes the design and implementation of runtime rerandomization moving target defense, an improvement on address space layout randomization (ASLR) that protects vulnerable programs from return-to-libc and return oriented programming (ROP) attacks. Chapter 11 describes the design and implementation of dynamic canary randomization, an extension of stack canaries, the powerful, effective method for protecting programs against attacks targeting stack overflow vulnerabilities. Chapter 12 describes

an implementation of enforcement of control-flow integrity, a technique that protects against program hijack at runtime, using a retargetable, static binary rewriter.

Chapter 2

Motivation

"In short, software is eating the world" [26]. Those words, from the mouth of famed software engineer turned venture capitalist Marc Andreessen, are the short version of the thesis that "[m]ore and more major businesses and industries are being run on software and delivered as online services" [26]. As each new industry and business is transformed to run according to the algorithms implemented in software, the cost incurred by failure of those software systems grows and grows. Failures may result from the malicious activity of an attacker, the nature of the complexity of the system, human error, etc. Whatever the cause, the consequences of failure can be dire, financially and physically.

In June 2014, McAfee estimated that the annual costs of identity theft may be as much as $160 billion [14]. That is a staggering figure and, alone, enough to justify research and development of a tool to improve the security and safety of software that stores and processes sensitive user information. Yet the problem is much greater. Although almost two-thirds of Americans carry a very powerful computer in their pockets [202] and more than ninety percent have access to the Internet [170], it is the times that humans interact with computers and do not realize it that the risks are greatest. Software hidden from users' view regulates the power grid, dams, reactors, power plants, transportation, democracy and delivery of medical treatment. When these systems fail – because of adversarial attacks, operator error or the naivete of those building the systems about the potential for failure – the costs are more than financial. No doubt there are monetary costs, but the damage extends into the physical world and can easily lead to a loss of life and/or liberty. A few examples will show how pervasive software has become in these contexts and the costs of their failure.

Automobiles have become increasingly automated by software. "[T]he Volt uses an estimated 10 million lines of code, running about 100 control units. [That is] up from about six million lines of code in typical 2009 model cars and as little as 2.4 million in 2005" [149]. Tesla, Mercedes and Cadillac offer systems that allow

for semi-autonomous operation of their vehicles [140]. When this software operates erroneously, the possibility of physical damage is real. In 2015, researchers attacked the software in a Jeep and remotely controlled the vehicle to crash into a ditch [151]. A tool for third parties that could rewrite compiled, proprietary software running on an embedded ARM platform [151] could have been used to defeat real-world exploitation immediately after the vulnerability was discovered and before the carmaker fixed and update its software and distributed the new version. Although no one was injured in the Jeep demonstration, the possibility of serious injury and death is real. In the case of "Unintended Acceleration", ultimately blamed on faulty software in Toyota's control units, more than 89 people died [121]. During the investigation into these deaths, the manufacturer revealed the source code for the software that runs these automobiles to certain experts under very strict conditions [121, 223]. In other words, a tool for third parties to add safety assurances to such a system would have to have been able to operate on binary-only software after it has been compiled, linked and possibly stripped of debugging information.

Medical devices implanted in humans and used in surgery are more and more often controlled by software. When that software fails, patients are at risk. In Summer and Fall of 2017, the FDA announced that the software running a commonly used pacemaker contained a vulnerability that exposed hundreds of thousands of patients to attack [131]. In February, 2016, a medical device controlled by a personal computer stopped responding after the patient was sedated because of a scheduled anti-virus scan [75]. The patient was unharmed but the surgery was delayed while the monitor computer was rebooted. A tool to add safety assurances to such a system would have to have been able to operate on software that runs on both a Windows-based desktop computer [75] and firmware that runs on an embedded, battery-powered hardware platform [20].

For electricity production and distribution, NIST has recognized the potential for losses incurred by failure in software: "Vulnerabilities might allow an attacker to penetrate a network, gain access to control software, and alter load conditions to destabilize the grid in unpredictable ways" [159]. That grid constitutes $1 trillion in assets and generates revenue of $350 billion per year [28]. A failure in power delivery affects hospitals, businesses and national security. The damages would be catastrophic. Part of the system that delivers electricity to end users are so-called smart meters that monitor customer usage and communicate back to electricity producers to help them better calibrate production. A Navigant study predicted that 1.1 billion of these smart meters will be deployed worldwide by 2022 [141]. The hardware and operating systems (OS) that run these smart delivery systems vary widely, from powerPC processors running a VX-based OS to x86 processors running Windows [178]. In order to add security and reliability to systems that operate on smart meters, then, a tool must be able to operate on software that runs on all these different hardware and software platforms.

Much has been made of Russia's attempts to influence the United States' presidential election in 2016 by manipulating news sources to sow discord among the electorate [165]. Less has been said about the possibility that exists for foreign or domestic adversaries to manipulate election results directly by accessing electronic voting machines or state boards of election [52]. At the computer security conference DEF CON hosted in Las Vegas in 2017, security researchers were given the opportunity to attack several different models of electronic voting machines actually used for state, local and national elections. In only a short period of time, researchers identified vulnerabilities in those voting machines, including at least one voting machine that was running an outdated vulnerable version of the open source software (OSS) OpenSSL [220]. As a result of the discoveries at DEF CON, Virginia decertified one of the voting machines subject to the researchers' attacks [55].

It is reasonable to suggest that all software, especially safety-critical software, be built safely and securely from the beginning rather than secured *post hoc* using the algorithms described here. Countless engineering standards and methodologies do exist to describe the procedures to be followed during development of software that will operate critical systems. For instance, in the aviation industry, the designer of software that controls aircraft engine functionality "must design, implement, and verify all associated software to minimize the existence of errors by using a method, approved by the FAA, consistent with the criticality of the performed functions" [8]. The problem is that attention to safety and security during design and implementation is not enough to make the resulting systems fully safe or secure. In 2015, incorrectly installed software caused engines on several Airbus planes to malfunction and led to fatalities [163]. Such a catastrophic failure could have been prevented if the software was modified by the end-customer to monitor the system's state and take non-fatal corrective actions to shutdown safely the engines that reached this configuration. Therefore, a tool that allows for *post hoc* transformation of critical software to add security and reliability is warranted.

Then there are those legacy systems that are at risk because of threats and failure cases that have evolved since their installation. A 2013 survey of federal information technology (IT) managers at 179 organizations found that 79% of their IT budget was spent maintaining legacy applications [128]. Migrating and/or upgrading legacy systems is not an easy task. Researchers continue to study the most efficient way to transition outdated systems to new platforms but the problem remains largely unsolved. More than that, there is the problem of maintaining the human resources needed to maintain legacy code [188]. The design and algorithms described herein allow developers to add security, safety and reliability to these legacy systems even if the application's source code has long since been lost or the human capital required to maintain that code has been depleted.

Finally, there are those who lament the slow pace of innovation in the software that runs critical systems [54]. One way to increase the pace of innovation in software is to adopt OSS. OSS is increasingly becoming the

backbone of the software that runs the exploding number of physical devices connected to the Internet, the so-called Internet of Things (IoT). Flexera surveyed 400 companies that use OSS in their commercial software products and found that only 37% have a policy to track its use and 39% say that either no entity is responsible for OSS compliance or that they could not identify that party [77]. This means that the software produced by these companies includes SOUP that will eventually need to be updated in place by users of the software embedded on the physical devices since even the developers do not know their software is vulnerable [77]. In its State of Software Security 2017, Veracode found that "only 28% of organizations [surveyed] do any kind of regular composition analysis to understand which [OSS] components are built into their [Java] applications" [227]. Furthermore, advocates for increasing the pace of innovation seem unwilling to spend the time in development to formally prove the safety and security of systems that operate in critical contexts. The result of such time-to-market pressure is that the software running critical systems will be more likely to need *post hoc* transformation to improve its security and reliability as flaws are discovered in production. Algorithms for after-the-fact transformation of programs like the ones described herein are well suited for just this use case.

Part II

Fundamentals

Chapter 3

Correctness

More fundamental than a static binary rewriter's ability to satisfy the constraints outlined in Chapter 1 is its correctness. Defining correctness of a static binary rewriter makes it possible to identify the criteria that bound the scope of its architecture and algorithms and will guide its design and implementation. This section presents a definition of a *correct* static binary rewriter and shows how that definition generates a baseline set of criteria that can be used to analyze any static binary rewriter to determine its correctness.

3.1 Correctness of a Statically Rewritten Program/Library

Program P_o is an original program to be rewritten. P_o, and each of the programs defined throughout this chapter, takes input that produces behavior. Input to a program may come from the keyboard, timers, the network, etc. The behavior of a program on an input are the actions of the program that an *external* viewer can observe about the program. This includes, but is not limited to, the program's output. Throughout the chapter, an equation like

$$B = P(x) \tag{3.1}$$

assigns to B the behavior of the program P when given input x.

The user has a set of zero or more transformations $T = [t_1, t_2, \ldots, t_n]$ that he/she will apply to modify the behavior of P_o. Applying T to P_o yields an ideal, *hypothetical* program P_m that represents P_o's behavior as modified by T. In other words,

$$P_m = T(P_o). \tag{3.2}$$

The static binary rewriter R takes P_m as input and generates P_r:

$$P_r = R(P_m). \tag{3.3}$$

Conceptually, P_r is a correctly rewritten version of P_m if the behaviors of the two programs are identical for every input. Formally, a correctly rewritten program P_r meets the following constraint:

$$\forall x \in X, P_r(x) = P_m(x) \tag{3.4}$$

where X is the set of valid program inputs for P_m.

According to the definition of correctness in Equation 3.4, P_r's correctness depends on P_m's *operation* not P_m's *correctness*. The definition implies nothing about whether, $\forall t_i \in T$, t_i correctly encodes the high-level semantics intended by its designer. The correctness of the transformations are the responsibility of the entity producing that transformation. Equations 3.4 and 3.2, read together, *does* imply that a correct static binary rewriter accurately applies all $t_i \in T$ to P_o when generating P_r from P_m.

3.2 Prerequisites to Meet Correctness Definition

Therefore, the correctness of P_r primarily, but not entirely, depends on successful recovery of P_m's control flow graph (CFG). The CFG of P_m is analyzed rather than the CFG of P_o because, as described above, correctness depends on the behavior of the ideally (but hypothetically) modified program. To analyze P_o instead would be to drop the requirement that the rewriter accurately applies the transformations of T.

A program's CFG is comprised of direct and indirect program control statements. Assume there are two oracles $\mathcal{O}_I$ and $\mathcal{O}_A$ that perfectly identify the addresses of program control statements and their targets, respectively. The application of the oracle $\mathcal{O}_I$ to program p yields an unordered set of the addresses of program control instructions in p. The application of the oracle $\mathcal{O}_A$ to program p yields an unordered set of the addresses of the targets of all the program control instructions (direct and indirect) in p.

$$I = \mathcal{O}_I(P_m) \tag{3.5}$$

$$A = \mathcal{O}_A(P_m) \tag{3.6}$$

During analysis, the static binary rewriter deploys a concrete implementation of the two oracles, $\mathcal{O}'_I$ and $\mathcal{O}'_A$, respectively, to generate I' and A'.

$$I' = \mathcal{O}'_I(P_m) \tag{3.7}$$

$$A' = \mathcal{O}'_A(P_m) \tag{3.8}$$

Section 4.2 describes the research challenges associated with building these concrete implementations.

The values in I and I' are addresses of either direct or indirect program control instructions. Direct program control instructions and their targets are easy to recover from P_m because the target address is explicitly included with the instruction. Indirect program control instructions are harder to analyze because they use a computed address as the indirect branch (IB) target (IBT) address for the program control transfer. The computed target address is loaded from memory, calculated arithmetically, or both. For more information, see Section 4.2). There is a *method* for calculating the target addresses of each IBT. That method may take as input some data internal (e.g., embedded addresses) or external (e.g., user input, memory state, etc.) to P_m.

Let i be an indirect program control instruction from P_m whose address is $i_a \in I$. The addresses of the targets of i are in set $T \subseteq A$ where $t_a \in T$ (because $T \subseteq A$, $t_a \in A$ also) is the target's address and t is the target instruction itself. By the definition of correctness in Equation 3.4, when P_r's program control reaches t', where t' is the rewritten version of t, P_r must execute the same behavior as would P_m when it reaches t.

If it were possible to determine completely the method of indirect program control instruction's target address calculation and the *provenance* of the data used as input, that method could be adjusted by the rewriter to allow for the placement of t' at any location in P_r. Unfortunately, discovering the methods for calculating the target addresses of indirect program control transfers is not trivial (Section 4.2 describes this in detail).

There are two important consequences of the assumption that it is not possible to completely understand the method of the indirect program control instruction's calculation of the target address. First, $T = A$. To assign T any other value would imply that there is some understanding of the method of calculating the addresses of the indirect program control's targets. Second, the semantics of P_r when program control is passed to address t_a $\forall t_a \in T$ in P_r must match the semantics of P_m when program control is passed to address t_a in P_m, a more strict requirement than the requirement outlined two paragraphs above.

Therefore, the conservative, safe way to guarantee correctness of a statically rewritten program/library is to make it so that, $\forall a \in A$, the semantics of P_r when program control is passed to address a in P_r match the semantics of P_m when program control is passed to address a in P_m. This criterion is part of the definition of pinned addresses, objects that play a vital role in the novel reassembly technique presented in this dissertation (See Section 4.2.2).

There are at least two ways to meet this requirement. First, the static binary rewriter could copy the instructions from P_m at a, $\forall a \in A$, directly to a in P_r. Second, the static binary rewriter could copy the

sequence of instructions at a, $\forall a \in A$, in P_m, place them at address b in P_r and place a *link* at a in P_r to b. There may be other ways to meet this requirement, but these two are the most straightforward. The first method is only applicable when the user does not make any functional changes to the program so the second method is preferred but has a implications for overhead. See Section 4.4 for more information.

For these reasons, if there is an $a \in A$ and $a \notin A'$, then it cannot be recreated in P_r and, as a result, the rewriter cannot guarantee that P_r is a correctly rewritten binary. Therefore, a necessary criterion for a correct static binary rewriter is that $A \subseteq A'$. As described in Section 4.2, an analysis that builds an A' that satisfies this criterion is challenging, but possible.

This relationship between A' and A is not a sufficient requirement, however. There are other scenarios that will affect the correctness of a program rewritten by the static binary rewriter. First, there are programs that perform a self-check before executing. Often, the self-check is implemented using an embedded checksum. The checksum is inserted into a program binary and is used to ensure that the binary is not modified after compilation. At runtime, the self-check implementation in the program generates a checksum of the program's in-memory representation and compares it with the value computed offline. If the value computed online and value computed offline do not match, the self-checker halts the program's execution. Because the rewritten version will have changed the binary representation of the program, this type of check will always cause a statically rewritten binary program to fail.

Second, there are SOUP binaries that are obfuscated before distribution. Obfuscation is done for a number of reasons but it is most commonly used as a way to prevent third parties from reverse engineering the binary. In general, obfuscated binaries are hard to analyze – security researchers spend enormous amounts of time analyzing obfuscated malware manually. If humans have a hard time, automated disassembly and semantic recovery tools will struggle too. When those processes fail on obfuscated binaries, the static binary rewriter will fail to generate a correctly rewritten binary. These two scenarios are explicitly out of scope.

3.3 Parallels to the Definition of Correctness in Other Fields

The formal definition of correctness in Equation 3.4 is derivative of the definitions in Barak et al. on cryptographic program obfuscation [32]. In their seminal work, the authors presented results about the impossibility of *virtual black box (VBB) obfuscation.* $\mathcal{O}_v$ is a VBB obfuscator of a program P

- when no probabilistic polynomial time (PPT) adversary A,
- given oracle-only access to P through $O(P)$ and
- given unlimited access (read, inspect, analyze, etc.) to $\mathcal{O}_v(P)$,
- can learn some bit of information i from $\mathcal{O}_v(P)$ beyond what it can learn from $O(P)$.

A's ability to learn i from $\mathcal{O}_v(P)$ that it cannot learn from $O(P)$ implies that $\mathcal{O}_v(P)$ leaks information about the internals of P and does not completely obfuscate its implementation. Barak et al. contrived a class of "self-eating functions" [185] that cannot be VBB obfuscated. Based on this impossibility result, they proposed an alternate version of obfuscation known as indistinguishability obfuscation (iO). The function iO is an indistinguishable obfuscator of a class of programs $\mathcal{P}$ whose members all compute the same function

- when no PPT A
- who has unlimited access to $P_1, P_2 \in \mathcal{P}$
- can distinguish between $(P_1, P_2, iO(P_1))$ and $(P_1, P_2, iO(P_1))$.

This parallels the notion outlined in the section describing the requirement that a statically rewritten version of a program produce the same behavior as the original program for all possible inputs. Using a definition similar in form to that of iO and reusing the notation presented above,

- when no polynomial time user U[1]
- can distinguish between $(P_m, O(P_r))$ and $(P_m, O(P_m))$

then P_r is a correct rewriting of P_m, the ideal, hypothetical transformation of P_o modified according to zero or more transformations T.

3.4 Discussion

Nothing in this section is meant to imply that exhaustive testing is a valid means of verifying the correctness, as it is defined here, of a statically rewritten program. There are programs that come with an extensive suite of test inputs for developers to test the correctness of a program/library after components are added, subtracted or updated. Extensive literature from the domain of generate-and-validate automated program repair shows that relying on these regression test inputs to automatically generate a patch makes it likely that the patched software will pass all regression tests but break intended but untested functionality [175, 202].

By implication, it cannot be said that a statically rewritten version of a program/library that passes developer-supplied tests is formally correct. In other words,

$$\forall x \in D, P_r(x) = P_m(x), \tag{3.9}$$

where D is a set containing all the inputs from the developer-supplied tests, is a necessary but not sufficient condition for the formal correctness of P_r. The fact that Equation 3.9 is not sufficient to prove the correctness of the rewritten program/library does not diminish the practical importance of validating the rewritten program/library against its user-supplied tests. The results of such tests are described in Chapter 5

[1]This is not a strict requirement – it is here only to emphasize the parallelism of the two definitions.

to demonstrate the practical ability of the architecture and algorithms of the static binary rewriter defined in this dissertation and its prototype implementation to rewrite large, complicated programs and libraries.

3.5 Future Work

Left for future work is research on whether the static binary rewriter could include features that guarantee the formal correctness of a statically rewritten program/library. It is possible that variations on proof-carrying code could solve this problem [155]. The static binary rewriter could serve as the untrusted code producer that generates the safety proof for the statically rewritten program/library. Instead of asserting that the software is safe, the proof included in the statically rewritten program could adhere to a "correctness" safety policy. The user of the static binary rewriter could take the statically rewritten artifact and subject it to verification before distribution. If the certification succeeds, the user can have confidence running the statically rewritten version of the program/library. It may also be possible to include a verifier directly in the OS that certifies statically rewritten programs/libraries when they are executed/loaded.

Another possibility left for future consideration is whether matching invariants between the original and statically rewritten program/library could guarantee formal correctness of the rewriter's output artifact [71]. In such a system, a tool could extract the invariants from the original program and determine whether the statically rewritten version upholds those invariants under all possible execution scenarios.

3.6 Conclusion

It is imperative that a static binary rewriter be correct. Although the creation of a formally correct static binary rewriter is explicitly beyond the scope of this work, the formal definition of correctness described in this chapter is useful for guiding the development of the architecture and algorithms of a static binary rewriter.

Chapter 4

Design and Architecture of a Static Binary Rewriter

4.1 Introduction

The algorithms designed to construct a retargetable, static binary rewriter that accomplishes the goals and conforms to the requirements outlined in Part I form a pipeline. Figure 4.1 illustrates the three phases of the pipeline: Intermediate Representation (IR) Construction, Transformation and Reassembly. Although the architecture used to coordinate the operation of the algorithms designed to construct the rewriter could have been monolithic, a pipeline architecture is more flexible and makes it easy to add new steps in the future as research progresses in any of the fields of computer science encompassed in each of the phases. This is not a new insight – it was the rationale for pipes in the UNIX OS [135, 180].

The IR Construction phase contains the algorithms necessary to take an input program, disassemble it into an IR, deduce the program's CFG and prepare the IR for modification by transformations (see Section 4.2). The Transformation phase contains algorithms to apply User-specified transforms that programmatically alter the IR and either correct a program's defects or add new functionality (see Section 4.3). The Reassembly phase contains the algorithms necessary to convert the modified IR back into executable machine code (see Section 4.4).

The IR database (IRDB) mediates communication among the algorithms of the individual pipeline phases. Depending on the task, the phases may read, write, or read and write the IRDB. The IRDB is an SQL-based system that stores a variety of information about the binary obtained from different sources. Besides information traditionally represented by an IR (e.g., the CFG), the IRDB is designed to store

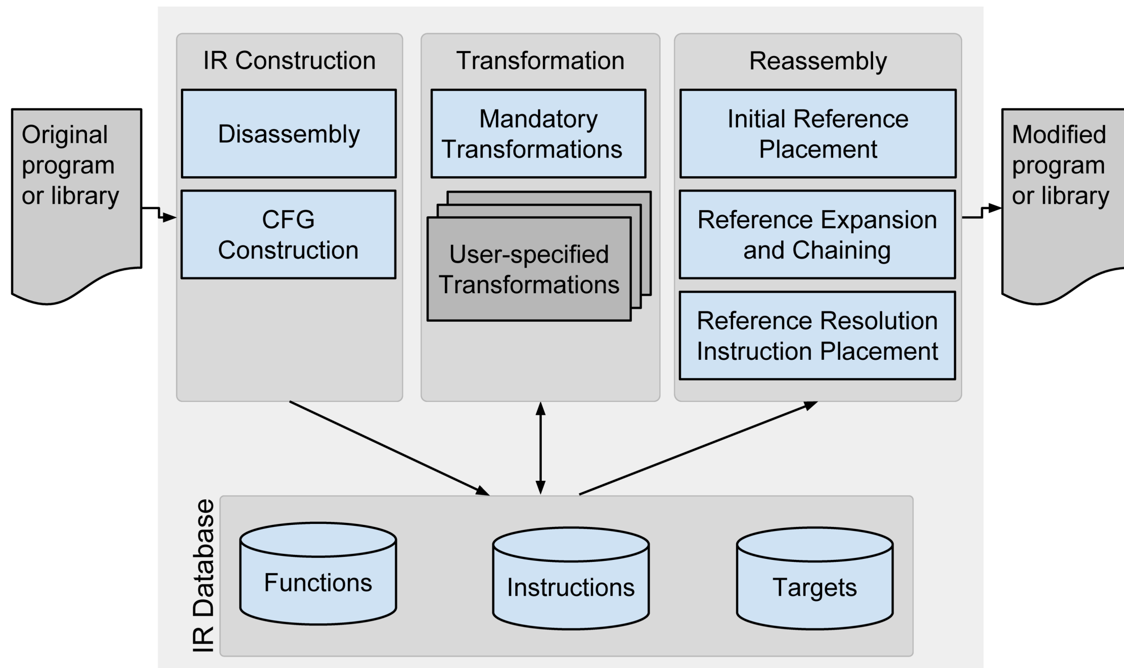

Figure 4.1: The rewriting pipeline.

information about the original and modified program that supports the algorithms that perform analysis and transformation. Section 4.2 describes the algorithms that populate the IRDB and the information about the original program collected; Sections 4.3 and 4.4 discuss, as appropriate, the algorithmic use of the IRDB in those contexts.

The choice of the IRDB as the mediator among the different phases of the pipeline was made for two reasons. First, the IRDB is specifically suited to flexibly hold a program's IR. The flexibility makes it possible for different phases to programmatically alter statements in the IR and store the result back to the database. The IRDB stores these changes in revision-controlled versions so that other phases can compare and isolate changes. Second, previous research has resulted in transformations (see Section 4.3) compatible with this storage mechanism. Although the choice of the IRDB was done consciously, it was not the only option and is not an inextricable part of the architecture and algorithms designed for the static binary rewriter.

4.2 IR Construction

The IR Construction phase consists of stages normally associated with "binary code analysis" [148]: disassembly and CFG reconstruction. The IR Construction phase begins with disassembly of the instructions of the program/library to be statically rewritten. Next, it reconstructs the original program's CFG and collects associated metadata. The results of both stages are stored in the IRDB for processing by the Transformation and Reassembly phases.

4.2.1 Disassembly

Disassembly of a binary program, "[d]ecoding bytes into machine instructions" [148], is not an easy task. A program may have overlapping instructions or non-code bytes may be interspersed among instructions [148]. Disassembly is even more difficult when the binary program does not contain symbols [148].

The design and architecture of the static binary rewriter do not require perfect disassembly but do rely on accurate disambiguation between code and data. In jump tables or in code that computes on embedded, read-only data elements, instructions and data are often mixed [148]. The problem is particularly difficult because bytes of data often decode into valid instructions [148].

In the general case, given access only to a program's machine code, code/data disambiguation is more than difficult – it is impossible [174]. In practice, however, disambiguation between code and data is feasible but, again, not easy. And although there are many existing disassemblers that disambiguate code and data, none is perfect.

Schwarz et al. give a good overview of two popular types of disassemblers and the techniques each uses to disambiguate code and data: the linear scan and recursive descent [191]. Linear scan disassemblers work by linearly stepping through the bytes of a program's code, beginning at the first byte. Upon deciphering an instruction at an address, a linear scan disassembler skips the number of bytes contained in that instruction and repeats the process.

Algorithm 1 shows one possible way to implement such a linear scan disassembler. The function DECODE performs the task of analyzing the bytes at address a and adding the decoded instruction to the IR. It returns the length, in bytes, of the instruction at a. DECODE uses DECODEINSTRUCTION to perform the actual conversion between a sequence of bytes and an instruction. It takes two parameters, an architecture and an address, and uses the architecture to drive the disassembly of the bytes at the address into the IR of the instruction at that address. DECODE uses ADDINSTRUCTION to add the instruction returned by DECODEINSTRUCTION to the IR. DECODE returns the size of the instruction, as determined by a utility function SIZEOFINSTRUCTION.

Bytes	Disassembly output
0x00: 55 48	0x00: push %rbp
0x02: 89 e5	0x01: mov %rsp,%rbp
0x04: bf d4	0x04: mov $0x4005d4,%edi
0x06: 05 40	
0x08: 00 00	

Table 4.1: The result of linear disassembly (in the right column) on a set of bytes (in the left column).

MAIN is the driver of the linear scan disassembler. It takes a "pointer" to a program to disassemble as its parameter. MAIN determines the size of the program and uses that to limit the scope of the search for instructions. MAIN invokes a support function DETECTARCHITECTURE – typically implemented by reading P's header information – that discovers the program's architecture. MAIN creates an empty IR to which instructions are added as they are decoded from P. MAIN uses the counter a to step through the entirety of P at instruction boundaries. The steps of the iteration are determined by the return value of DECODE, which, as a side effect, adds the decoded instruction to the IR. Ultimately, MAIN returns the IR to its caller.

Algorithm 1 Pseudocode of the algorithm for a linear scan disassembler.

```
1: function DECODE(Address a, Architecture arc, IntermediateRepresentation IR)
2:     i ← DECODEINSTRUCTION(arc, a)
3:     ADDINSTRUCTION(IR, i)
4:     return SIZEOFINSTRUCTION(i)
5: end function
6: function MAIN(Program P)
7:     ProgramSize ← FILESIZE(P)
8:     a ← 0
9:     arc ← DETECTARCHITECTURE(P)
10:    IR ← NEWINTERMEDIATEREPRESENTATION()
11:    while a ≤ ProgramSize do
12:        a ← a + DECODE(a, arc, IR)
13:    end while
14:    return IR
15: end function
```

Consider the example shown in Table 4.1. The linear scan disassembler would start with the sequence of bytes at `0x0`. Deciphering the byte `0x55` as a `push` instruction, the linear disassembler would skip to the bytes at `0x01`. Deciphering the bytes `0x48 0x89 0xe5` as a `mov` instruction, the linear scan disassembler would skip to the bytes at `0x04`. Deciphering the bytes as another `mov` instruction, the linear scan disassembler would continue at `0x09` and the bytes following.

Linear scan disassemblers are powerful because they are fast. The downside is that they often fail to make the proper distinction between code and data [191].

A recursive descent disassembler attempts to correct for this flaw in linear scan disassemblers. Instead of simply disassembling bytes in a program's code from the beginning, a recursive descent disassembler attempts to disassemble only at the instructions that are actually executed at runtime. In order to accomplish this

goal, a recursive descent disassembler keeps a list of addresses where previously disassembled instructions could pass program control at runtime. As the disassembler completes its work on a particular sequence of instructions, it begins disassembly again at the next list in the queue. It finishes disassembly when there are no more locations in the queue. The starting address of program execution is usually embedded in the program's header; disassembly in a recursive descent disassembler begins at this starting address.

Algorithm 2 shows one possible way to implement such a recursive descent disassembler. The function Decode performs the task of analyzing the bytes at address a and adding the decoded instruction to the IR. It returns the instruction at a. Decode uses DecodeInstruction to perform the actual conversion between a sequence of bytes and an instruction. It takes two parameters, an architecture and an address, and uses the architecture to drive the disassembly of the bytes at the address into the IR of the instruction at that address. Decode uses AddInstruction to add the instruction returned by DecodeInstruction to the IR. Decode returns the instruction at a for further inspection by the caller.

Main is the driver of the recursive descent disassembler. It takes a "pointer" to a program to disassemble as its parameter. Main determines the size of the program and uses that as an upper limit on its search for instructions. Main invokes a support function DetectArchitecture – typically implemented by reading P's header information – to discover the program's architecture. Main creates an empty IR to which instructions are added as they are decoded from P. Before finally beginning the actual work of disassembling the program, Main initializes a queue with the program's entry point. The entry point of a program is typically found by reading the program's header information. That queue holds the addresses where code is absolutely known to exist.

From every address in the queue, the disassembler begins a linear scan. That scan uses the Decode function to perform the actual decoding operation. The linear scan only stops upon reaching the end of the program or an unconditional program control instruction, as determined by utility function IsUnconditionalPCInstruction. Because program execution does not fall through an unconditional program control instruction, there is no reason to continue a linear disassembly at the subsequent instruction. If there is code after the unconditional program control instruction its address will eventually end up in the queue because another instruction will target it. As instructions are decoded, the targets of those that transfer program control are added to the queue. As mentioned earlier, the process terminates when the queue is empty. Ultimately, Main returns the IR to its caller.

For example, consider the assembly code in Listing 4.1. Assume that the program's header defines `0x400511` as the program's starting address. Assume further that the disassembler had done the disassembly that generated the assembly instructions shown. At this point, the disassembler's queue would contain the addresses `0x400523 0x4004ed 0x40053f 0x400535 0x4004f4 0x4004fb` because the program control

```
400511: cmpl $0x0,-0x4(%rbp)
400515: jne 400523
400517: mov $0x0,%eax
40051c: callq 4004ed
400521: jmp 40053f
400523: cmpl $0x0,-0x4(%rbp)
400527: jns 400535
400529: mov $0x0,%eax
40052e: callq 4004f4
400533: jmp 40053f
400535: mov $0x0,%eax
40053a: callq 4004fb
```

Listing 4.1: Recursive descent disassembler example

instructions found within this series of instructions transfer control to those addresses.

Algorithm 2 Pseudocode of the algorithm for a recursive descent disassembler.

```
1: function DECODE(Address a, Architecture arc, IntermediateRepresentation IR )
2:     i ← DECODEINSTRUCTION(arc, a)
3:     ADDINSTRUCTION(IR, i)
4:     return i
5: end function
6: function MAIN(Program P)
7:     ProgramSize ← FILESIZE(P)
8:     arc ← DETECTARCHITECTURE(P)
9:     IR ← NEWINTERMEDIATEREPRESENTATION()
10:    DefiniteInstructionAddrs ← NEWQUEUE()
11:    PUSH(DefiniteInstructionAddrs, ENTRYPOINT(P))
12:    while not EMPTY(DefiniteInstructionAddrs) do
13:        a ← POP(DefiniteInstructionAddrs)
14:        while a ≤ ProgramSize do
15:            i ← DECODE(a, arc, IR)
16:            a ← a + SIZEOFINSTRUCTION(i)
17:            if ISPCINSTRUCTION(i) then
18:                PUSH(DefiniteInstructionAddrs, TARGETOF(i))
19:                if ISUNCONDITIONALPCINSTRUCTION(i) then
20:                end if
21:            end if
22:        end while
23:    end while
24:    return IR
25: end function
```

This is a simple example; the program control transfers are all direct. Heuristics and deeper analysis are also present in a recursive descent parser in order to handle situations where program control is indirect through registers or memory (e.g., a `call` through a register). IDA Pro is a recursive descent disassembler [98, 68] which makes it "[simple] and [effective] at avoiding disassembly of data" [191].

The work of Schwarz et al. also exemplifies the research being done to improve disambiguation techniques, in particular, and disassembly techniques, in general [191]. The modularity of the design and architecture of

the rewriting technique described in this dissertation make it possible to aggregate the output of multiple disassemblers and leverage each tool's strengths and compensate for their weaknesses. Moreover, the modular design makes it possible to incorporate information provided by new disassemblers as they are written. As of this writing, the prototype implementation of the overall design and architecture presented in this dissertation combines the output of `objdump` and IDA Pro.

No matter whether the IR Construction phase employs a linear scan or recursive descent disassembler or a combination thereof, there exists the possibility that no definitive assessment can be made about whether a particular range of bytes is data or code. There are four possible outcomes for labeling a range of bytes. The disassembler can:

Case 1 correctly and conclusively label the range of bytes as data or instructions: No special handling is required for this case, obviously.

Case 2 incorrectly, but conclusively, label the range of bytes as data.

Case 3 ambiguously label the range of bytes: In both Cases 2 and 3, the bytes are treated as data *and* instructions. The bytes themselves are placed in the statically rewritten binary at the same address they were found in the input binary (to handle the case that the bytes are referenced by address and treated as memory). At the same time, the bytes themselves are also treated as instructions, and disassembled and incorporated into the IR.

Case 4 incorrectly, but conclusively, label the range of bytes as instructions: During the static analysis, it is impossible to detect this case and, if it happens, there exists the possibility that the statically rewritten binary will fail. For this reason, the analysis and disassembly is designed to be as conservative as possible.

In addition to capturing the information traditionally associated with an IR, the design and architecture of the static binary rewriter relies on the collection of additional information about each instruction. In particular, for each instruction, the IR contains:

fallthrough instruction The fallthrough instruction is used for reassembling linear sequences of instructions and is undefined if the instruction is an unconditional program control instruction.

target The target is only included if the instruction itself is a control flow instruction and the target is known statically. The target is used for correctly matching instructions that refer to one another by their addresses (See Section 4.3.2).

pinned address Assignment of the pinned address is discussed next.

In the IR, an instruction's fallthrough and target instructions are eventually stored in a logical format where the relationships between instructions can be recreated without reference to their position in the original program. This independence makes subsequent analysis and transformation of the IR more flexible. Section 4.3.2 contains more information about how a disassembled instruction, with its link to other instructions based on absolute addresses, is converted to represent logical links between instructions.

4.2.2 CFG Reconstruction

Even assuming the perfect disassembly of a program's binary code, reconstructing a program's CFG from its machine code is still not a straightforward process. Meng et al. present several cases where CFG reconstruction is complicated even when accurate disassembly is possible: indirect control flow, non-return functions, functions that share code, non-contiguous functions and functions with tail calls [148]. While each complicating factor must be managed, and even the best of existing tools (e.g., IDA Pro) are not able to handle every case correctly [148], algorithms in the Transformation and Reassembly phases assume proper analysis of indirect control flow. Therefore, the correct design of the algorithms that reconstruct the CFG is vitally important.

The CFG reconstruction algorithm relies on *pinned addresses*, locations of instructions in the original program/library that may be targeted indirectly at runtime. Addresses of units of data are always pinned because there are many complicated ways that a program can reference data by calculating its address at runtime using information external to the program (e.g., user input, network input, the contents of memory, etc.). On the other hand, the address, a, of an instruction, i, in the original program is pinned only if the original program calculates dynamic program control references to a at runtime. Using the definitions from Chapter 3.1, each address a that the original program calculates dynamic program control references to at runtime is in A. Pinned addresses play a crucial role in the static binary rewriter's ability to meet the correctness requirements in Section 3.2.

When there are dynamic program control references to a at runtime, a will be stored as the pinned address value of i in the IR. In other words, pinned address analysis depends directly on correct calculation of indirect control flow.

Pinned addresses of instructions are important for the algorithms of the Reassembly phase. Throughout reassembly, a pinned address of an instruction in the original program, a, corresponds to exactly one instruction, i. The algorithms of IR Construction assign the original correspondence between a and i. During the Transformation phase, one or more Mandatory/User-specified Transformations may change i to i' and a will correspond to i'. For the statically rewritten program to function according to the semantics of

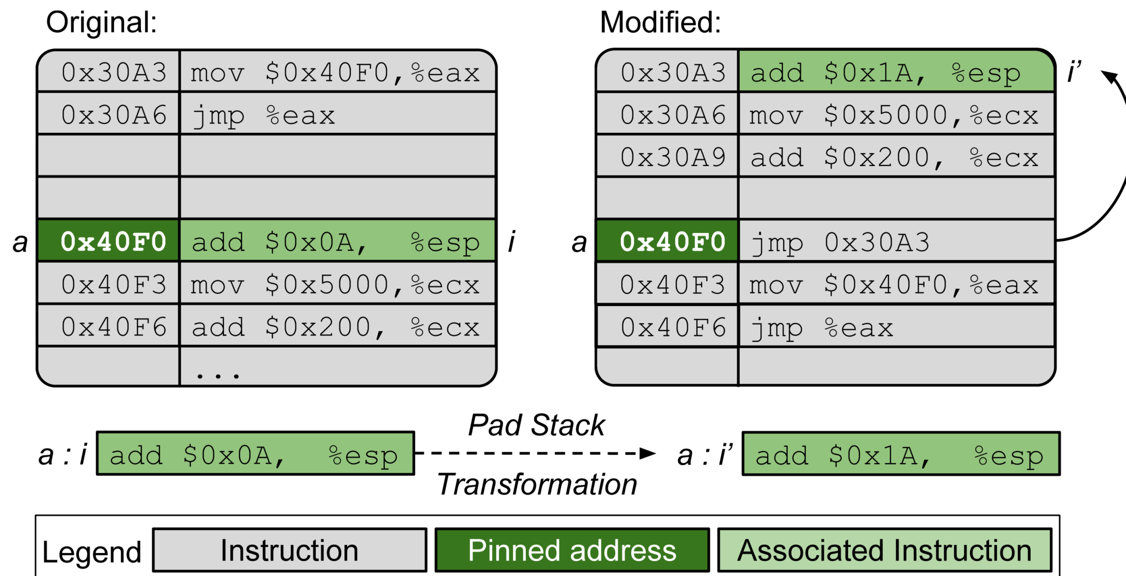

Figure 4.2: Pinned addresses, transformations and links.

the original program, as subsequently modified through transformations, when the transformed program's program counter (PC) reaches address a, instruction i' must be executed. Again, this is an implication of the definition of correctness of a static binary rewriter from Equation 3.4 and the further explanation of Section 3.2. Section 4.4 discusses the algorithms of the Reassembly phase that maintain this invariant in the statically rewritten program/library.

Figure 4.2 shows an example of this process. Instruction i, `add $0x0A, %esp` is associated with pinned address a. The illustrative *Pad Stack* transformation modifies i so it allocates a larger stack (the `$0x0A` constant is changed to `$0x1A`). The modified instruction, i', is still associated with a. When i' is eventually placed at address `0x30A3` in the modified program, the link[1] at a is updated appropriately.

Addresses of instructions may be pinned for a number of reasons. Most commonly, however, addresses are pinned because they are the targets of IBs. IB targets (IBTs) appear in jump tables, immediately after call instructions, the beginning of functions, etc. Just because program control reaches an instruction indirectly does not mean that its address must be pinned. There are cases where the program's behavior with respect to an IBT can be analyzed and modeled statically.

The example in Figure 4.3 shows the pinned addresses discovered in a typical jump table implementation. The five instructions at the beginning of the Text section are representative of a compiler's typical translation of a high-level-language's (HLL) `switch` statement into a jump table. The first instruction ensures that the value used to select the target case of the switch statement is within the limits (9, in this case, although only three cases are shown). If the value exceeds that limit, the second instruction will transfer program control to

[1]The use of links in the Reassembly phase is discussed in detail in Section 4.4

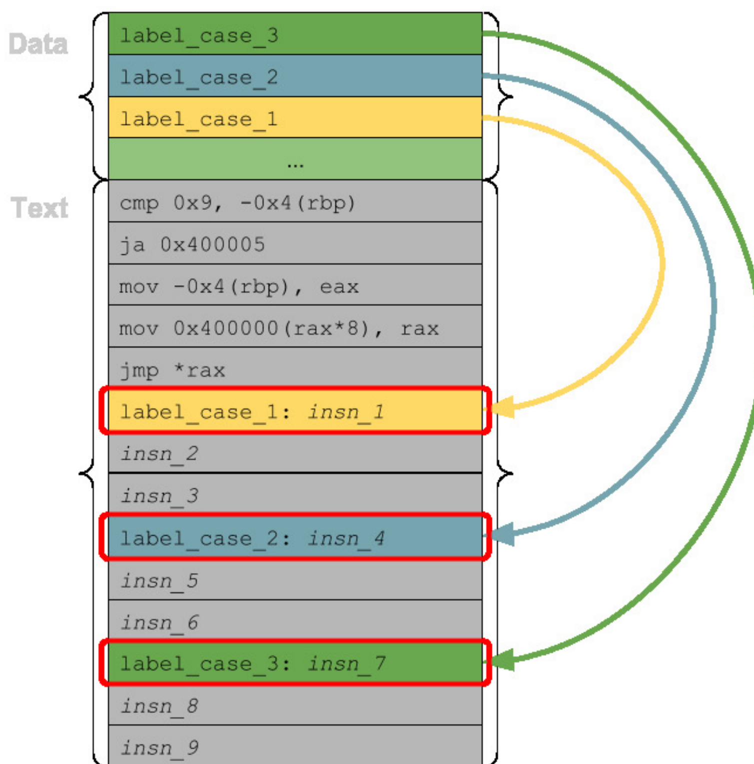

Figure 4.3: Discovery of pinned address from a typical jump table implementation. Instructions at pinned addresses are highlighted in red.

some handler. The next three instructions use the value to index an array in read-only memory that contains an array of addresses of the code that implements each of the cases. The second of the three instructions gives a hint to the CFG reconstruction algorithm about the address and the layout of this array. The bytes at `0x400000` have the address of the code for the first of the cases. `(rax*0x8)` indicates that the addresses themselves are 64-bit. The first instruction of the jump table indicates the total number of addresses in the array.

From this information, the CFG reconstruction algorithm can read 9 addresses from the $9 * 8 = 72$ bytes beginning at `0x400000`. These addresses are potential targets of the indirect control flow at the fifth, and final, instruction in the jump table implementation. Therefore, the CFG reconstruction algorithms pins those addresses.

Not all jump implementations follow this format. The CFG reconstruction algorithm has an extensive infrastructure for recognizing the various, common idioms that compilers follow when translating the high-level-language's `switch` statements into assembly instructions.

The addresses of instructions after `call` instructions are pinned. The semantics of function calling are

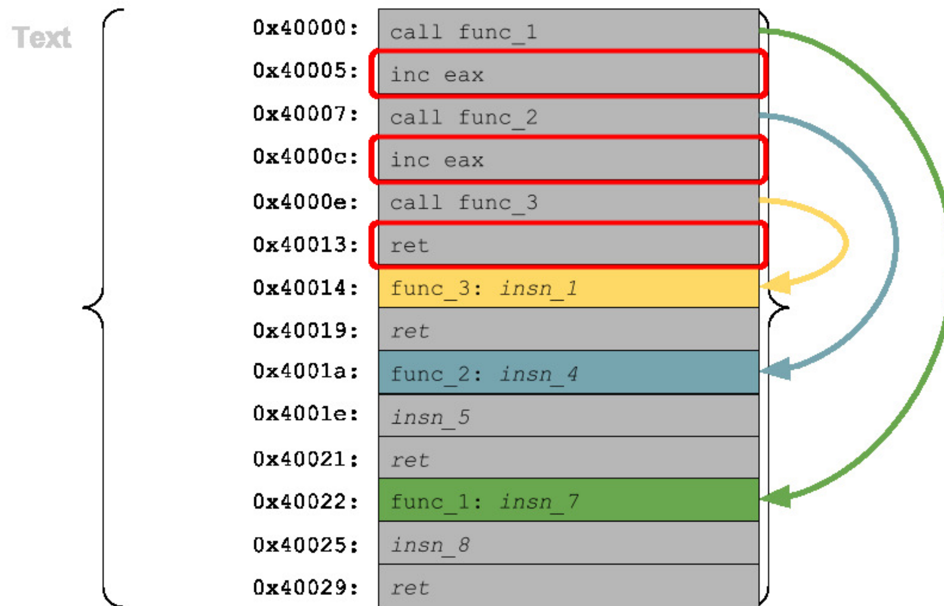

Figure 4.4: Discovery of pinned address from function calls. Instructions at pinned addresses are highlighted in red.

relatively straightforward [108]. First, the address of the instruction after the `call` is pushed on the program's runtime stack. Second, program control is passed to the target of the call instruction. When the invoked function `ret`urns, the address on the top of the program stack is popped and used as the indirect target of an implied program control instruction thereby transferring control of execution back to the instruction following the original `call`.

Assume that there is a call instruction c with a return address r in an original program and c is rewritten as c' with a return address of r' in the statically rewritten program. There seems to be no technical reason why r and r' must be equal. However, programs may rely on the value of the return address at runtime in order to perform calculations, access memory, etc. For this reason, in the rewritten program, r must equal r' and, therefore, r is pinned. Pinning solves the problem because what matters is the address r on the stack, not necessarily the instructions at that address. The example in Figure 4.4 shows the addresses of the fallthrough instructions from the three `call`s being pinned.

CFG reconstruction algorithms do *not* need to determine the set of possible targets for every particular IB instruction. Instead, the algorithms rely only on the fact that P, the set of all pinned addresses in a program, contains *at least* all the addresses of IBTs in the original program. In other words, it must be that

$$B \subseteq P \tag{4.1}$$

where B is the set containing the addresses of every IBT from the original program. Using the terms and symbols of Chapter 3 (and specifically Section 3.2), B is A and P is A' where the CFG reconstruction algorithms are the implementation of the oracle $\mathcal{O}'_A$. Rewriting 4.1,

$$\begin{aligned} A = B \subseteq A' = P \\ A \subseteq A' \end{aligned} \tag{4.2}$$

which *is* the necessary requirement from Section 3.2 for constructing a correct statically rewritten program/library.

It is possible to calculate P naïvely by adding every instruction of the original program to P . This assignment clearly satisfies the requirement. As discussed in Section 4.4, however, such an assignment does not give the algorithms of the Reassembly phase any opportunity to flexibly re-place instructions in the statically rewritten program/library. Moreover, it does not allow for the creation of an *efficient* rewritten binary program.

Ideally $B = P$. As $|P - B|$ grows, the algorithms of the Transformation and Reassembly phases will have an increasingly difficult time generating a space-efficient rewritten binary because pinned addresses are fixed and fragment the space available for placing rewritten instructions (see Section 4.4 for additional information about how fragmentation affects the effectiveness of rewriting). Therefore, the CFG reconstruction algorithm leverages a set of heuristics that analyze the original program's CFG to select pinned addresses. Again, it is imperative that the technique be conservative. Even a single address $a \in B$ where $a \notin P$ will cause the Transformation and Reassembly algorithms to generate a transformed binary that may not operate correctly.

For a more detailed description of the algorithms used to identify pinned addresses of instructions and data, see Section 12.2.1, Hiser et al. [97] and Zhang et al. [250]. At the time of this writing, the CFG reconstruction algorithm is able to handle very complex programs and libraries, even when they include large amounts of handwritten assembly code (see Chapter 5).

At the conclusion of this phase, the algorithms have constructed an IR and a CFG that are usable by the algorithms of the subsequent Transformation and Reassembly phases.

4.3 Transformations

The algorithms of the Transformation phase modify the original program's IR. The IR of the original program is modified by two types of transformations: Mandatory and User-specified. Mandatory and User-specified transformations use the same architecture to modify the IR but perform different tasks. Mandatory transformations prepare the IR for modification by User-specified transformations. Preparation of the original program/library's IR is required so that subsequent transformations can operate on the IR without regard for the details of the specific target platform. User-specified transformations are optional transformations written by users of the static binary rewriter that modify or add/remove functionality to/from the original program/library.

4.3.1 Transformation Architecture

Transformations, Mandatory or User-specified, are implemented using an application programming interface (API) that enables programmatic manipulation of the IR through the IRDB. Each transformation is invoked with a single parameter, the identifier of the program in the IRDB. Using this identifier, the transformation can retrieve the IR constructed by the IR Construction phase of the static rewriter pipeline. The API provides methods to iterate through the IR by function, basic block, instruction, etc. Further, the API provides the ability to read/modify each of those units. Most commonly, a transformation reads or modifies the IR at the instruction level. Once the transformation has completed its work, the modified IR is saved back to the IRDB. The process is repeated for each of the Mandatory and User-specified transformations.

4.3.2 Mandatory Transformations

Mandatory transformations in the Transformation phase produce a modified IR that can be more easily modified by User-specified transformations and enables the algorithms of the Reassembly phase to place recreated instructions arbitrarily in the statically rewritten program/library's address space. Mandatory transformations most commonly address issues with the target platform and its instruction set architecture (ISA).

For example, on the x86 platform, there are several Mandatory transformations that handle `call` instructions and instructions that rely on relative addressing. Many x86 instructions can use PC-relative addressing. The `jmp` instruction is one such instruction. Assume instruction i_1 transfers control to i_2 with a `jmp`. On an x86, i_1 is a `jmp` to a_{i_2}, the address of i_2. However, a_{i_2} is encoded in i_1 relative to a_{i_1}. To relocate instructions when they are placed in the statically rewritten program/library, relationships like these that rely on the instructions' addresses in the original program have to be translated into logical links. Although the

algorithms in the IR Construction phase construct a CFG, Mandatory transformations like this one ensure that the links between instructions in the IR are logical and not based on absolute addresses. Returning to the example, after this Mandatory transformation, the CFG in the IR logically links i_1 to i_2, not a_{i_2}.

There are other examples where relative addressing poses a particular problem for the algorithms of the Reassembly phase and require preparation by the Mandatory transformations. Memory operations may also be PC-relative. If there is a memory access instruction that references memory relative to its address in the original program, it cannot be arbitrarily placed in the statically rewritten binary without affecting correctness. For example, consider a `load` encoded as `0x48 0x8b 0x05 0TB 0x09 0x20 0x00` at address `0x5f6`.[2] This instruction loads memory `0x2009db` bytes after `0x5fd`. The Mandatory transformation ensures that the IR represents this instruction as a `load` that refers to the 8 bytes at `0x200fd8`. This guarantees that, no matter where this `load` is placed in the statically rewritten binary, its target will be correct.

The `call` instruction on the x86 is subject to a Mandatory transformation.[3] As described extensively in Section 4.2, a callee function may use the caller's address, as stored on the stack, at runtime. For this reason, statically rewritten `call` instructions must be handled carefully to ensure that the return address of a `call`ed function in a statically rewritten binary program matches the return address of the corresponding `call`ed function in the original program. To handle this case, a Mandatory transformation exists that converts each `call` into a `push`/`jmp` combination. For example, consider the case where the input binary calls function *func_1* with a `call func_1` at `0x5f6`. At runtime, the return address on the stack of the original program after this instruction has been executed is `0x5fb`. The Mandatory transformation that fixes `call` instructions transformations that `call` instruction into `push 0x5fb; jmp func_1`.

4.3.3 User-specified Transformations

After the Mandatory transformations are applied, User-specified transformations are applied. These are transformations that the user implements to modify the original program. As mentioned earlier, there are many ways a user could modify the original program to improve its security, reliability and dependability. Examples include the three applications described in Part IV.

Consistent with the goals and motivations for this research set forth in Chapters 1 and 2, the user must be able to choose to implement their own transformations instead of being forced to choose from set of predefined transformations. The algorithms of the Transformation phase allow any practical implementation to provide APIs for users to develop their own transformations.

[2]This example is taken from [35]

[3]This is not always the case. The details are not important for this example.

Transformations, Mandatory and User-specified, modify the program/library to be rewritten. They accomplish different tasks using the same underlying architecture. Transformations are the final step in the pipeline before the program/library is statically rewritten by the algorithms of the Reassembly phase.

4.4 Reassembly

At the heart of the static binary rewriter are the algorithms that reassemble the (potentially transformed) IR and CFG into a rewritten output program/library. The reassembly algorithms are the most novel of all the algorithms developed in the fundamental, basic architecture and design of the static binary rewriter.

The reassembly algorithm operates in the address space of the statically rewritten binary program. At the outset, the address space of the statically rewritten binary is empty. As input, the reassembly algorithms accepts the (again, potentially transformed) IR and CFG and converts it into a series of instructions and units of data which are then assigned a location in the statically rewritten program's address space.

The first element placed in the address space of the statically rewritten binary program/library are the links at pinned addresses. Recall that the instructions in the original program that transfer program control to pinned address cannot be precisely identified. Therefore, the correctness of the rewritten artifact cannot be guaranteed if the instructions at pinned addresses are moved during reassembly. In other words, the rewritten program/library must behave semantically equivalently at pinned address a as it does at address a in the original program/library. See Chapter 3 for the derivation and rationale of this requirement. The reassembly algorithms use links to maintain this equivalence.

Links are just one of the ways to perform reassembly that guarantees correctness of the statically rewritten program. An alternate solution is to simply re-place instructions in the statically rewritten binary program/library at exactly the same addresses they were at in the original program/library – for instruction i whose address in the original program/library is a, instruction i', its semantically equivalent instruction after all transformations are applied, and its fallthrough instructions are re-placed at a in the address space of the statically rewritten program/library. This approach may improve the runtime efficiency of the statically rewritten program/library and remove the overhead introduced by links[4] but it is only applicable when the user does not make any functional changes to the program/library. As soon as the user transforms a sequence of instructions at a pinned address, those instructions may no longer fit in the space after a and before a subsequent pinned address (or other entity with a fixed position that cannot be overwritten) in the statically rewritten program/library. The alternate solution, links, however, gives the reassembly algorithm flexibility to place instructions throughout the statically rewritten binary program/library by placing a *link* at every pinned address that targets i' and its fallthrough instructions no matter where they are re-placed in the address space of the statically rewritten program/library. Under the assumption that users will deploy the architecture and algorithms described in this dissertation to transform the semantics of a program/library, links are the preferred solution although they do have limitations.

[4]The source of the overhead is described in the remainder of this section and quantification of the orverhead is discussed in Chapter 5; Part III describes ways to limit the impact of the overhead.

Depending on the ISA, the mechanism used to implement the link has a certain size l. Consider an original program/library that has two consecutive instructions that must be pinned, i_1 and i_2. i_1 has length i_{1_l} and i_2 has length i_{2_l}. When $l > i_{2_l}$, there is not enough space to implement a link. Section 4.4.2 describes the solution to this problem.

More importantly, links have performance implications. First links may cause P_r, the statically rewritten binary program/library, to use more space than P_m, the ideally transformed version of the original binary program/library, which may mean a larger in-memory size (as measured by the resident set size (RSS)) and/or larger on-disk size. Because links have to be placed at certain addresses in P_r for the statically rewritten program/library to be correct (See above and Sections 4.2.2 and 3.2), the links fragment P_r's address space. The fragmentation turns what would otherwise be a contiguous free area of memory into a series of smaller segments of free memory that can be used to place the rewritten instructions and data from the IR and CFG prepared in previous phases. In the case of a free segment of memory that is too small to contain any sequence of instructions or data in its entirety, one option is to leave the space unused. The other option is to divide a sequence of instructions into different blocks to fit in the small open spaces. Splitting a sequence of instructions to utilize those small spaces requires the use of links between the multiple pieces which adds even more performance overhead. No matter whether the choice is made to leave the space empty or split a sequence to fill the space, P_r will use more space than P_m.

Second, links may negatively impact the instruction cache performance. Third, and finally, program control transfers through links add additional program control transfer entries into the branch prediction cache that may negatively affect performance runtime overhead. Solutions to these performance problems are considered in Part III.

Reassembly is driven by the REASSEMBLY function (see Algorithm 3) and is an iterative process that terminates when all *unresolved links* are resolved. An unresolved link is a link to a target that exists only in the IRDB. A link is resolved when the (again, potentially transformed) sequence of instructions in the IRDB at the target of that link are placed in the address space of the statically rewritten binary and the two are associated. Reassembly begins with an initial set of unresolved links placed at pinned addresses. In the process of resolving those links by *actualizing dollops* (See 4.4.1) in the memory space of the statically rewritten program/library, new unresolved links to other targets may be introduced. The targets of those links are then actualized in the memory space of the statically rewritten program/library and the links are resolved. The iterative process continues until there are no more unresolved links. Figure 4.5 illustrates the internal state of the algorithm as it reassembles a program. The following sections explain the reassembly process in detail.

Algorithm 3 Algorithms of the Reassembly phase, Part I: Main reassembly method

1: **function** REASSEMBLY(Pins, Original, Rewritten)
2: RewrittenMemorySpace ← MEMORYSPACE($Original$) ∪ $InfinitePage$
3: ConstrainedUnresolvedLinks ← ∅
4: UnConstrainedUnresolvedLinks ← ∅
5: SizeOfConstrainedLink ← $PlatformConstrainedLinkSize$
6: SizeOfUnConstrainedLink ← $PlatformUnConstrainedLinkSize$
7: RangeOfConstrainedLink ← $PlatformConstrainedLinkRange$
8: ActualizationMap ← ∅

9: INITIALLINKPLACEMENT(RewrittenMemorySpace, Pins, ConstrainedUnresolvedLinks)
10: LINKEXPANSIONANDCHAINING(RewrittenMemorySpace,
ConstrainedUnresolvedLinks,
UnConstrainedUnresolvedLinks,
SizeOfConstrainedLink,
SizeOfUnConstrainedLink)
11: **while not** ISEMPTY(UnConstrainedUnresolvedLinks) **do**
12: LINKRESOLUTIONANDDOLLOPACTUALIZATION(RewrittenMemorySpace,
UnConstrainedUnresolvedLinks,
ActualizationMap)
13: RESOLVELINKS(RewrittenMemorySpace, ActualizationMap, UnConstrainedUnresolvedLinks)
14: **end while**
15: **end function**

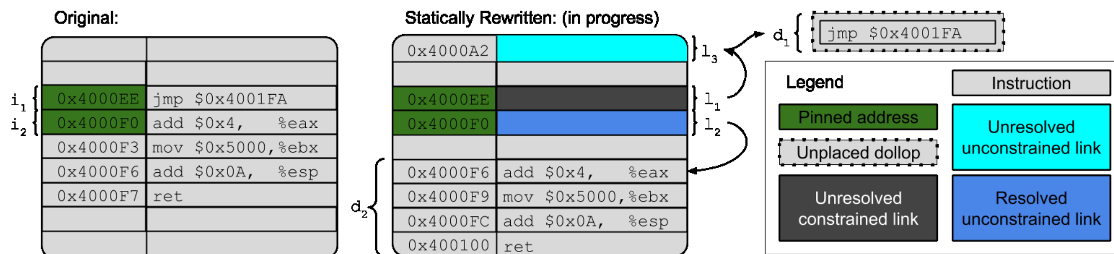

Figure 4.5: The internal state of the reassembly process as it produces a statically rewritten binary program/library.

4.4.1 Initial Link Placement

The statically rewritten binary program/library's memory space begins empty. To ensure that there is always enough space for additional code that may have been added in a Mandatory or User-specified transformation, the statically rewritten program/library's address space is augmented with an "infinite" overflow area whose contents will be appended to the rewritten binary as necessary (Line 2 in Algorithm 3). Before any other reassembly occurs, the data segment is copied directly from the original program/library to the statically rewritten program/library. The reassembly algorithm begins by attempting to place unresolved constrained links at pinned addresses (INITIALLINKPLACEMENT in Algorithm 3).

Again, a link is a reference to a sequence of instructions. The reassembly algorithms refers to a sequence

Algorithm 3 Algorithms of the Reassembly phase, Part II: Initial link placement and link expansion methods

```
16: function InitialLinkPlacement(MemorySpace, Pins, ConstrainedUnresolvedLinks)
17:     for all p ∈ Pins do
18:         unresolved ← UnresolvedLinkFromPin(p, true)
19:         WriteUnresolvedLink(MemorySpace, unresolved)
20:         Add(ConstrainedUnresolvedLinks, unresolved)
21:     end for
22: end function

23: function LinkExpansionAndChaining(MemorySpace,
      ActualizationMap,
      ConstrainedUnresolvedLinks,
      UnConstrainedUnresolvedLinks,
      PlatformConstrainedLinkSize,
      PlatformUnConstrainedLinkSize)
24:     RangeOfConstrainedLink ← CalculateRangeFromSize(PlatformConstrainedLinkSize)
25:     while cul ← Pop(ConstrainedUnresolvedLinks) do
26:         Address ← cul.From
27:         if IsFree(Address, PlatformUnConstrainedLinkSize) then
28:             unconstrained ← UnresolvedLink(Address, cul.To, false)
29:             Add(UnConstrainedUnresolvedLinks, unconstrained)
30:             Write(MemorySpace, unconstrained)
31:         else
32:             AddressForConstrainedLink ← FindNearbySpaceForConstrainedLink(MemorySpace,
                  Address,
                  PlatformConstrainedLinkSize,
                  RangeOfConstrainedLink)
33:             NewConstrainedUnresolvedLink ← UnresolvedLink(AddressForConstrainedLink, cul.To, true)
34:             Add(ConstrainedUnresolvedLinks, NewConstrainedUnresolvedLink)
35:             ResolveLink(ActualizationMap, cul, AddressForConstrainedLink)
36:         end if
37:     end while
38: end function
```

Algorithm 3 Algorithms of the Reassembly phase, Part III: Actualization methods

```
39: function LinkResolutionAndDollopActualization(MemorySpace,
      UnConstrainedUnresolvedLinks,
      ActualizationMap)
40:   while uul ← Pop(UnConstrainedUnresolvedLinks) do
41:     Address ← 0x0
42:     if IsActualized(ActualizationMap, uul.To) then
43:       Address ← ActualizedAddress(uul.To)
44:     else
45:       Address ← ActualizeDollop(ActualizationMap, uul.To, UnconstrainedUnresolvedLinks)
46:     end if
47:     ResolveLink(ActualizationMap, uul.From, Address)
48:   end while
49: end function

50: function ActualizeDollop(ActualizationMap, Address, UnconstrainedUnresolvedLinks)
51:   FreeSpace ← FreeMemorySpace(MemorySpace, SizeOfDollop(InstructionAt(Address)))
52:   UsedSpace ← 0
53:   repeat
54:     Coalesce ← true
55:     Dollop ← NewDollop(Address)
56:     Instruction ← InstructionAt(Address)
57:     while Instruction and
            not IsTargeted(Instruction) and
            (UsedSpace + SizeOfInstruction(Instruction)) < FreeSpace.Size do
58:       AddInstruction(Dollop, Instruction, Address)
59:       Address ← Address + SizeOfInstruction(Instruction)
60:       UsedSpace ← UsedSpace + SizeOfInstruction(Instruction)
61:       if HasTarget(Instruction) then
62:         Target ← TargetOf(Instruction)
63:         LinkToTarget ← UnresolvedLink(Address, Target.OriginalAddress, false)
64:         if IsActualized(Target.OriginalAddress) then
65:           ResolveLink(LinkToTargetAddress,ActualizedAddress(Target.OriginalAddress))
66:         end if
67:       end if
68:       Instruction ← Instruction.FallThrough
69:     end while
70:     WriteDollop(Dollop)
71:     if Instruction and UsedSpace ≥ FreeSpace.Size then
72:       Coalesce ← false
73:     else
74:       if Instruction then
75:         FallthroughDollopLink ← UnresolvedLink(Address, Instruction.OriginalAddress, false)
76:         Add(UnConstrainedUnresolvedLinks, FallthroughDollopLink)
77:       end if
78:     end if
79:   until not Coalesce
80: end function
```

Algorithm 3 Algorithms of the Reassembly phase, Part IV: Support methods

```
81: function ActualizedAddress(ActualizationMap, Address)
82:     if IsActualized(ActualizationMap, Address) then
83:         return ActualizationMap[Address]
84:     end if
85:     return 0x0
86: end function

87: function IsActualized(ActualizationMap, Address)
88:     if KeyExists(ActualizationMap, Address) then
89:         return true
90:     end if
91:     return false
92: end function

93: function ResolveLinks(RewrittenMemorySpace, ActualizationMap, UnConstrainedUnresolvedLinks)
94:     StillUnConstrainedUnresolved ← ∅
95:     while uul ← Pop(UnConstrainedUnresolvedLinks) do
96:         if IsActualized(uul.Target) then
97:             TargetAddress ← ActualizedAddress(uul.Target)
98:             ResolveLink(ActualizationMap, uul.From, TargetAddress)
99:         else
100:            Push(StillUnConstrainedUnresolved, uul)
101:        end if
102:    end while
103:    UnConstrainedUnresolvedLinks ← StillUnConstrainedUnresolved
104: end function

105: function ResolveLink(ActualizationMap, Link, Target)
106:    ActualizationMap[Link.From] ← Target
107: end function

108: function FindNearbySpaceForConstrainedLink(MemorySpace,
       LinkAddress,
       SizeOfConstrainedLink,
       RangeOfConstrainedLink)
109:    Address ← LinkAddress + RangeOfConstrainedLink.Start
110:    for Address < LinkAddress + RangeOfConstrainedLink.End do
111:        if IsFree(MemorySpace, Address, SizeOfConstrainedLink) then
112:            return Address
113:        end if
114:    end for
115:    return 0x0
116: end function

117: function UnresolvedLinkFromPin(Pin, Constrained)
118:    return UnresolvedLink(Pin.Address, Pin.Address, Constrained)
119: end function
```

Algorithm 3 Algorithms of the Reassembly phase, Part V: Support methods, continued

```
120: function UnresolvedLink(From, To, Constrained)
121:     if Constrained then
122:         return ConstrainedLink(From, To)
123:     else
124:         return UnConstrainedLink(From, To)
125:     end if
126: end function

127: function IsFree(MemorySpace, Address, Length)
128:     i ← Address
129:     for i < Length do
130:         if IsBusy(MemorySpace[i]) then
131:             return false
132:         end if
133:     end for
134:     return true
135: end function
```

of instructions linked by their fallthroughs as a *dollop*.[5] Links target dollops and dollops may target one another. Dollop a targets Dollop b if there is a link in Dollop a that targets Dollop b. A link from dollop a to dollop b always targets the first instruction in b. A dollop a is *connected* with a dollop b dollop a targets dollop b. Two dollops a and b are *interconnected* if dollop a targets dollop b and dollop b targets dollop a. The more that one dollop targets another, the more (inter)connected they are – this can range from *loosely* (inter)connected to *tightly* (inter)connected.

The concrete instantiation of a link requires implementation which normally depends on the target platform of the statically rewritten binary/program. Commonly, the reassembly algorithm uses an ISA's jump instruction(s) as the mechanism to implement the link. Again, links are *unresolved* when they target only data or dollops in the IR that do not yet exist in the memory space of the statically rewritten program/library. A dollop is *actualized* when its instructions are reconstructed from the IR and assigned a location in the statically rewritten program/library's address space. An unresolved link points to targets that have not been actualized. Links are *resolved* to the address of a dollop's actualization in the statically rewritten program/library. In Figure 4.5, l_1, l_2 and l_3 are links. l_1 and l_2 are resolved and l_3 is unresolved.

A link to a dollop is *constrained* when there is a restriction on where its target dollop may be actualized in the modified program's address space with respect to the link's address. Constraints on links may be the result of the density of pinned addresses or the mechanism used to implement links on a particular platform. The density of pinned addresses refers to the number of pinned addresses in a unit area of memory of the original program, for any arbitrarily defined length unit. The more pinned addresses there are per unit area,

[5] The description of dollops here is limited to their relationship to links. Please see Section 4.4.4 for a more detailed description of dollops.

the greater the density. In a very dense part of the original input program/library, consecutive instructions that are pinned may not be far enough apart to place links without overlapping. Section 4.4.2 describes the solution to this problem in detail.

In other cases, density requires the use of a short link implementation mechanism. Some ISAs feature multiple versions of instructions that can be used to implement links. For instance, the x86 ISA has a short jump instruction that use PC-relative branches with limited branch offsets. In these cases, the actualized address of the target dollop is constrained by the size of the branch offset field in the encoded instruction (See 4.4.3).

In addition to being unresolved, the initial set of links placed at pinned addresses are constrained. Subsequent passes in the algorithm unconstrain or chain each link. See Section 4.4.3 for the details. Initially using constrained links makes it possible to avoid the workaround described in Section 4.4.2 as often as possible.

Placement of the unresolved constrained links at the pinned address is accomplished by the InitialLinkPlacement function. InitialLinkPlacement is implemented as a simple loop over each of the pinned addresses in the set $Pins$. Each entry in the set $Pins$ is a pinned address. Each of the pinned addresses is converted to an unresolved constrained link using the UnresolvedLinkFromPin function. An unresolved link is a record with three fields: the address of the link in the memory space of the statically rewritten program/library ($From$), the address of the target of the link in the IR (To) and a flag indicating whether the link is constrained ($Constrained$). After creation, each of the unresolved constrained links created for the pinned addresses is stored in the set $ConstrainedUnresolvedLinks$ and written to the memory space of the statically rewritten program/library.

In Figure 4.5, there is a 2-byte instruction i_1 at `0x4000EE` and a 3-byte instruction i_2 at `0x4000F0` and both have pinned addresses. The control transfer implementing the link to the transformed instruction i'_1 will have to encode the address of i'_1 when it is placed. No matter how the ISA encodes addresses (relative or absolute), the encoding cannot exceed two bytes without interfering with the link at the adjacent pinned address. If the ISA does not support addressing the full address space in two bytes, the link at `0x4000EE` must be constrained.

4.4.2 Handling Dense Pinned Addresses

On machines with variable-length instructions such as x86, it is possible for pinned addresses in the code to be too close together for the control transfer instruction used to implement a link to fit. More precisely, for platforms with variable-length instructions where the length of the smallest instruction is shorter than the

smallest instruction that could be used to implement a link, there may arise a situation where two pinned addresses are too close to accommodate the instruction used to implement the link. For example, on x86 two consecutive addresses may be pinned but there is no useful one-byte control transfer instruction.[6]

In such a situation, to resolve the links, the reassembly algorithm uses *sleds*. A sled is a sequence of bytes, any of which can be the target of a program control instruction with program execution always reaching a single, known synchronization point. For example, on x86, consider the bytes: `0x68, 0x90, 0x90, 0x90, 0x90, 0xf4`. If control is transferred to the first byte of the sequence, the machine executes a `push 0x90909090` instruction (since `0x68` is the push immediate opcode), then the instruction for the `0xf4` byte. If control is transferred to any of the `0x90` bytes, the machine executes 1 or more `nop` instructions (since `0x90` is the opcode for `nop`), then the instruction for the `0xf4` byte. If a longer sled is needed to deal with more densely packed links, the technique can be extended by simply adding more `0x68` bytes at the beginning of the sequence. The four `0x90`s indicate the end of the sled.

Sleds have another nice property: by examining the value(s) on the stack at the synchronization point, it is possible to determine which path was taken through the sled. Carefully crafted dispatch code at the sled's synchronization point inspects the stack contents and can disambiguate and implement two or more adjacent links appropriately.

The implementation of links using sleds can be particularly slow to execute, but links are very rarely necessary. In practice, more than two or three consecutive links requiring sleds are rare. See Chapter 5 for the experimental results. As a result, the performance impact of sleds is negligible compared to their flexibility.Simple optimizations (such as replacing the `nop`s with a 2-byte control transfer) eliminate any practical need for more sophisticated techniques.

The sled implementation is necessarily architecture dependent. However, the technique itself can likely be used on any architecture with variable-length instructions.

By design and definition on platforms with fixed-length instructions, no instruction is shorter than the shortest instruction used to implement a link. On platforms with fixed-length instructions, then, there is no need for sleds. For example, on ARM, each instruction is 32 bits and 4-byte aligned. Two pinned addresses are always at least 32 bits apart which leaves enough space for the instruction used to implement the link on the ARM platform.

4.4.3 Link Expansion and Chaining

Once a constrained unresolved link is placed at each pinned address, the reassembly process determines the ones that can be unconstrained. LinkExpansionAndChaining accomplishes this by handling each entry, *cul*, in

[6]The `ret` instruction is a 1-byte control transfer, but is not suitable for resolving a link.

the set of unresolved constrained links, $ConstrainedUnresolvedLinks$. Depending upon the ISA of the implementation target, there is a minimum size, $PlatformUnConstrainedLinkSize$, necessary to implement a link that addresses the entire address space. If the space $[cul.Address + PlatformConstrainedLinkSize, cul.Address + PlatformUnconstrainedLinkSize]$ in the statically rewritten program/library is free, LinkExpansionAndChaining converts *cul* to an unresolved unconstrained link. Unresolved unconstrained links are tracked in the same record type as a unresolved constrained link. The difference between the two is, obviously, that the *Constrained* flag is **false**.

If that space in the statically rewritten program/library is not free, a second unresolved constrained link $NewConstrainedUnresolvedLink$ is added within reach of a constrained link's addressable distance from $cul.Address$. In this case, *cul* is then resolved to the $NewConstrainedUnresolvedLink$ which itself is placed back in $ConstrainedUnresolvedLinks$. This process is known as *chaining* [130].

Finding space for the chain link is accomplished with FindNearbySpaceForConstrainedLink. This function accepts $MemorySpace$, the address space of the statically rewritten program/library, $LinkAddress$, the address of the existing constrained link and $Range$, and a pair of offsets $(Start, End)$ that describe the addresses reachable by the existing constrained link that exists at $Address$. FindNearbySpaceForConstrainedLink simply loops through the addresses in $[Address - Start, Address + End]$ and looks for $SizeOfConstrainedLink$ bytes, the fourth parameter to the function, of available memory.

In Figure 4.5, l_2 would initially have been an unresolved constrained link but has been converted to an unresolved unconstrained link because there are no pinned addresses in $[\texttt{0x4000F0}, \texttt{0x4000F0} + s)$. For every remaining unresolved constrained link, l, that targets instruction i, a new unresolved unconstrained dollop link, l', is added in the modified program's address space. l is resolved to l' through one or more intermediate links and l' is set to link to i. In Figure 4.5, l_1 is a link to d_1 that is chained through l_3.

4.4.4 Link Resolution and Dollop Actualization

At this stage of the reassembly, the only remaining unresolved links are unconstrained. The final stage of the reassembly process is iterative and largely handled by LinkResolutionAndDollopActualization where the entries of set $UnConstrainedUnresolvedLinks$ are resolved until that set is finally empty. This may require more than one complete pass through $UnConstrainedUnresolvedLinks$ because, as stated earlier, the process of resolving a link may itself introduce new unresolved unconstrained links.

Each entry *uul* in $UnConstrainedUnresolvedLinks$ is handled in one of two ways. Either *uul.To* is already present in the statically rewritten program/library's address space or it is not. In the former case, the reassembly algorithm simply writes a resolved unconstrained link at *uul.Address* to *uul.To*'s location in

the memory space of the statically rewritten program/library. Several helper functions and data structures exist to support the mapping of the target sequences of instructions in the IR to the address of the dollop in the statically rewritten program/library's address space (ActualizedAddress, IsActualized). The latter case is more involved – a dollop containing the instruction at $uul.To$ in the (again, potentially transformed) IR must be actualized.

Dollop actualization is straightforward (ActualizeDollop). Dollop *Dollop* begins with instruction $Instruction_0$ at $uul.To$ in the IR and includes $Instruction_0$'s fallthrough $Instruction_1$, $Instruction_1$'s fallthrough $Instruction_2$, and so on. The last instruction in *Dollop*, $Instruction_n$, is the first instruction that has no fallthrough.

However, a dollop *Dollop* may end before the first instruction that has no fallthrough instruction. In the case where the actualization algorithm is considering dollop *Dollop*'s x^{th} instruction, $Instruction_x$, if $Instruction_x$'s fallthrough is at a pinned address or is the target of another instruction, then *Dollop* ends at $Instruction_x$. This constraint makes it possible to maintain the invariant that all links point to the beginning of dollops. When actualization of dollop *Dollop* stops before reaching the first instruction without a fallthrough, a link is added at the end of *Dollop*'s actualization that targets $Instruction_x$'s fallthrough. The sequence of instructions beginning with $Instruction_x$'s fallthrough is referred to as *Dollop*'s *fallthrough dollop*. The link appended to *Dollop*is an unresolved unconstrained link which is then added to the set $UnresolvedUnConstrainedLinks$ for later actualization.

Actualization of a dollop *per se* happens when the reassembly algorithm places the instructions of *Dollop* linearly in a consecutive block of free addresses in the statically rewritten binary program/library. In Figure 4.5, d_2 is an actualized dollop.

The algorithms of the Reassembly phase try to *coalesce* dollops together where possible. After dollop *Dollop* is actualized in the range of free space $FreeSpace$, if there is additional space in $FreeSpace$, then *Dollop*'s fallthrough dollop, if one exists, is actualized immediately after the last instruction in *Dollop*. Consider dollop *Dollop* with instructions $\{Instruction_0 \ldots Instruction_n\}$. *Dollop*'s fallthrough dollop is the dollop that begins with instruction i_n's fallthrough instruction. Because dollop actualization usually ends with the first instruction with no fallthrough instruction, there will not normally be a fallthrough dollop. Dollops may get fallthrough dollops during the operation of the algorithms of the Reassembly phase if/when they are split.

The benefit of coalescing is that no link is needed to transfer program control between *Dollop* and *Dollop*'s fallthrough. Coalescing happens even if the free space remaining in $FreeSpace$ after actualizing *Dollop* cannot hold the entirety of *Dollop*'s fallthrough dollop. In this case, *Dollop*'s fallthrough is split. Dollop

splitting is explained in detail below. If there is space left in $FreeSpace$ after actualizing *Dollop*'s fallthrough dollop, the process continues with the fallthrough dollop of *Dollop*'s fallthrough dollop.

There are certain cases where fallthrough coalescing does not occur even if there is additional space in $FreeSpace$ after actualizing *Dollop*. Most often, coalescing does not occur because *Dollop*'s fallthrough dollop is already actualized. In this case, an unconstrained resolved link is inserted to associate *Dollop* with *Dollop*'s fallthrough. *This is a policy decision.* There is no reason that the fallthrough dollop could not be actualized a second time. The policy decision reflects the fundamental design goal of creating statically rewritten programs/libraries that are size efficient (See Part I). In Chapter 5, the effects of this policy decision are explored.

Although not shown in Algorithm 3, a dollop (not a fallthrough dollop) may also be split when there is no block of free space big enough to accommodate all of its instructions. Dollop *Dollop* of instructions $Instruction_1 \ldots Instruction_s \ldots Instruction_n$ is split at a split point, $Instruction_s$. *Dollop* is truncated to contain instructions $\{Instruction_1 \ldots Instruction_{s-1}\}$ and $Dollop'$ is actualized to contain instructions $\{Instruction_s \ldots Instruction_n\}$. An unconstrained unresolved link that targets $Instruction_s$ is appended to the end of *Dollop* and added to $UnResolvedUnconstrainedLinks$.

After LinkResolutionAndDollopActualization completes an iteration over the set $UnConstrainedUnresolvedLinks$, that set is inspected to see if the most recent actualization process satisfied any other outstanding unresolved links (ResolveLinks. Once ResolveLinks completes, the actualization process begins again by actualizing dollops referred to by the links in $UnConstrainedUnresolvedLinks$. The statically rewritten binary program is completely reassembled when $UnConstrainedUnresolvedLinks$ is empty.

4.4.5 Example

Again, Figure 4.5 illustrates these concepts in the context of a program being reassembled. For this example, the target platform is the x86 ISA where there are variable-length instructions. A `jmp`, the implementation of links for this target, can be as short as two bytes (program control transfer is constrained to nearby locations) and as long as five (program control can be transferred anywhere in the program). In this example there are two pinned addresses, two dollops and three links. Dollop d_2 is already placed; dollop d_1 is not. Link l_1 began as a constrained unresolved link to an instruction in dollop d_1. For expository purposes, assume that d_1 could not be actualized at an address that is addressable in 2 bytes from l_1. Chaining was used and link l_1 was resolved to l_3 and l_3 became an unresolved unconstrained link to the instruction in d_1.

Because dollop d_2 is already placed, link l_2 is resolved. Link l_2 began as a constrained link but because there were no pinned addresses in $[\texttt{0x4000F0}, \texttt{0x4000F5})$, l_2 was converted to an unconstrained link.

4.4.6 Actualization and the Size of Statically Rewritten Program/Library

Assuming that User-specified transformations do not remove features from the original library/program (the most common use case), the algorithms of the Reassembly phase will necessarily produce a statically rewritten binary program/library that is at least as large as the original binary library/program. One of the goals for the design and architecture of this static binary rewriter is to minimize the on-disk and in-memory size of the statically rewritten program/library (see Part I). Before optimizing the statically rewritten program/library to accomplish those goals, it is important to understand why the statically rewritten program/library increases in size.

First, there are the links at every pinned address of an instruction. In the original program/library, there was an instruction at the pinned address that is replaced with the link implementation in the statically rewritten program. For example, assume that there is an instruction i at pinned address a in the original program. i is an instruction that requires 4 bytes to encode. There is enough space around a in the rewritten program to use an unconstrained link without chaining to reach i' so the link implementation requires 5 bytes to encode. In this scenario, the size overhead for this one pinned address is 5, the size of the implementation of the link.

Chaining adds additional overhead. For example, assume that there is an instruction i at pinned address a in the original program. i is an instruction that requires 4 bytes to encode. This time, however, there is not enough space around a in the rewritten program to use an unconstrained link. Chaining (through a single intermediate link) is required to reach i'. A constrained link requires two bytes to encode and an unconstrained link requires 5 bytes to encode. In this scenario, the size overhead for this one pinned address is 7, the size of the implementation of the constrained and unconstrained links.

It is possible to quantify the amount of space wasted in a statically rewritten binary program attributable to pinned address and links. The implementation of each constrained link is C bytes. The implementation of each unconstrained link is U bytes. The function *ChainCount* takes a pinned address and returns the number of constrained links required to reach the unconstrained link that ultimately points to the target. The set A contains all the pinned addresses of instructions in the program/library to be rewritten. The wasted space attributable to pinned addresses and links is:

$$W = \sum_{i=1}^{|A|} (ChainCount(A_i) * C + U)$$

As the equation shows, as the number of pinned addresses increases, so too does the size of the statically rewritten binary program/library. Therefore, reducing the number of pinned addresses would reduce size overhead. See Hiser et al. [96] for one way to reduce the number of pinned addresses.

There are other ways to reduce the overhead in addition to simply decreasing the number of pinned addresses. If the target dollop is placed directly at the pinned address, that removes the need for a link. Chapter 7 explores this possible optimization. Further, if a constrained link could be used to reach the target whenever possible instead of always using unconstrained links, that would decrease the overhead on platforms that use variable sized instructions for the implementation of links. Chapter 9 explores just such an optimization.

Dollop splitting may increase the on-disk and in-memory size of the statically rewritten program. It does, however, represent a trade off. Not splitting dollops may leave gaps of unused space in the statically rewritten program which itself will increase the size overhead. The problem of choosing whether to split a dollop or leave an empty gap is akin to the classic bin packing problem. If any dollops are split when reassembling the statically rewritten program/binary, the links used to associate the pieces of the dollop with one another add overhead. The algorithms of the Reassembly phase implement those links as unconstrained links and it is possible to quantify the waste space in the rewritten output attributable to dollop splitting. Let function $SplitCount$ take a dollop as a parameter and return the number of pieces used for that dollop in the rewritten output. Let the set D contain all the dollops. Where U is again the size of the implementation of an unconstrained link, W is the wasted space attributable to dollop splitting:

$$W = \sum_{i=1}^{|D|} ((SplitCount(D_i) - 1) * U)$$

Of course, User-specified transformations that add functionality to the statically rewritten program/library will necessarily increase the size of the rewritten output, too.

The statically rewritten program illustrated in Figure 4.6 demonstrates the impacts of these three forms of actualization overhead. The algorithms of the IR Construction phase recognize that `start`, `label_a` and `label_b` are pinned addresses. A User-specified transformation extended `func_1`. The implementation of the links in the statically rewritten program, shown in red, add overhead to the rewritten program. The

Figure 4.6: The algorithms of the Reassembly phase are one cause of overhead in the statically rewritten program/library. Shown here is the overhead of the algorithms of the Reassembly phase from links at pinned addresses and dollop splitting and the overhead from User-specified transformations.

additional instructions `f_new_insn_1` and `f_new_insn_2` add additional overhead, too. The algorithms of the Reassembly phase chose to split the dollops associated with code at `label_a` and `func_1` and required additional program control transfer instructions be incorporated into the statically rewritten program in order to associate parts of the dollops with one another.

Finally, the algorithms of the IR Construction, Transformation and Reassembly phases give the user the opportunity to link in external code for use by callbacks. In this use case, the user of the rewriter has separately compiled code that he/she wants to invoke under certain conditions in the statically rewritten program. In addition to adding instructions to invoke that functionality, the code required to implement the callback must be added to the statically rewritten program/library. This additional code necessarily

increases the size of the output. The application described in Chapter 10 uses callbacks and quantifies the size overhead of the rewritten program with respect to the size of the callback implementation itself.

Chapter 5

Evaluation

This chapter describes the experimental validation and evaluation of the architecture and fundamental algorithms of the static binary rewriter described in the preceding three chapters. The actual evaluation is done on a prototype implementation of these algorithms. The prototype implementation, known as Zipr, was constructed for the purpose of evaluating the underlying architecture and algorithms and for demonstrating the static binary rewriter's ability to rewrite SOUP.

The first section addresses potential problems with experimental bias that may taint the results presented in this section and subsequent evaluations discussed in the remainder of the dissertation. The next section describes the experimental platform. All experimental results throughout the chapter and the remainder of this dissertation were generated on this platform. The third section describes the results of the validation experiments which were designed to demonstrate that the architecture and algorithms developed in this research, when implemented in an actual system, could successfully rewrite SOUP. The final section describes the results from performance experiments that compare the performance of programs statically rewritten by Zipr with their native versions.

5.1 Experimental Bias

Mytkowicz et al. make the case that measurement bias is common in systems research results [154]. According to their results, the bias is unintentional but significant. It may cause performance improvements to be reported as performance impairments or vice versa.

In their work, Mytkowicz et al. suggest two ways to prevent bias in systems research. First, they suggest that researchers use experimental setup randomization. If a diverse workload of benchmarks is unavailable, diversifying the setup and configuration of the experiment at the same time the variable under measurement

is invariant will prevent measurement bias. Second, they suggest performing a causal analysis for each conclusion.

The following experimental evaluation attempts to counter measurement bias using both techniques. First, the evaluation presents results using diverse workloads. Besides the use of industry standard benchmarks, the evaluation presents results from tests using a second set of benchmarks, the Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge (CGC) dataset (see 5.6 for a complete description).

Second, a version of causal analysis is applied to experimental results for each of the significant results (improvements and impairments). Causal analysis of the hypothesis that X causes Y is described in Mytkowicz et al. as a three step process:

Intervene Change the system under test in a way that modifies X without affecting any other system parameter.

Measure Retest the system after intervention.

Confirm Check that the results match the expected change after the intervention.

The version of causal analysis used in this dissertation involves examining the source code for the original and statically rewritten program/library, exploring the interaction between hardware and software, comparing different evaluation characteristics/criteria and/or a combination thereof. This type of causal analysis is used here because, through inference, it provides the same information as true causal analysis and is possible using the available benchmarks and datasets.

Finally, in this chapter and the entirety of the dissertation, the computation of averages of overheads is done using the geometric mean [76]. For averages of other quantities, especially those that could potentially have a zero value which makes the geometric mean inapplicable, the arithmetic mean is used.

5.2 Evaluation Platform Specifications

Two systems, Host A and B, were used for the experimentation and evaluation reported throughout the dissertation. Depending on the hypothesis tested, experiments were conducted on one or both of the two hosts.

Host A is a single CPU computer with 48GB of RAM. The CPU, an Intel Xeon CPU E5-2680 Version 2, runs at 2.80GHz and has 10 cores [13]. Each core has two threads for a hostwide total of 20 threads. Each CPU has three levels of cache. The first level of cache is per core and contains two separate 32K, 8-way set associative instruction and data caches. The second level of cache for each core is a 256K shared instruction/data cache for each core. Finally, the last level cache is 25MB and shared between all cores. The compiler used to generate the native versions of the statically rewritten programs is GCC version 4.8.4. The OS is the 14.04 LTS distribution of Ubuntu Linux.

Host B is a single CPU host that also has 48GB of RAM. The CPU, an Intel Xeon CPU E5645, runs at 2.40GHz and has 6 cores [10]. Each core has two threads for a hostwide total of 12 threads. Each CPU has three levels of cache. The first level of cache is per core and contains two separate 32K, 2-way set associative instruction and data caches. The second level of cache for each core is a 256K shared instruction/data cache for each core. Finally, the last level cache is 12MB and shared between all cores. The compiler used to generate the native versions of the statically rewritten programs is GCC version 4.8.4. The OS is the 14.04 LTS distribution of Ubuntu Linux.

The two hosts are mostly identical, except for their CPUs and the system software running in the background. Host A has twice the cache as Host B and a faster clock speed. The system software running on Host A uses 480.24 MB of resident memory. The system software running on Host B uses 4.24 GB of resident memory. Those values were calculated on the hosts when they were idle and represent the average of samples taken every minute over a twenty four hour period.

5.3 Validation Experiments

Tests of Zipr against software programs with large code bases served to validate the ability of the static binary rewriter's design and algorithms to rewrite SOUP. The GNU C Library, *glibc*, OpenJDK's *libjvm*, the Apache web server, and the GNU core utilities are large programs and libraries with characteristics that make it possible to evaluate the ability of the design and architecture's ability to handle SOUP, including software with compiler-generated and hand-coded assembly. Additionally, these software packages were chosen because they contain a significant number of unit tests. In order to isolate and validate the fundamental architecture and algorithms from user-defined transformations, the tests were run against programs statically rewritten by Zipr with only a *Null Transformation* installed.

The Null Transformation is the most basic User-specified transformation. In fact, it is not a transformation at all. It is simply a no-op modification invoked on the IR and CFG during the User-specified Transformation stage of the Transformation Phase. In other words, the original and the modified programs are semantically equivalent after transformation by the Null Transformation. Using this "no operation" transformation means that any change to program behavior after it has been rewritten is the result of an error in the architecture and algorithms of the static binary rewriter's rewriting technique.

GNU C Library, *glibc* Rewriting *glibc*, the GNU implementation of the C standard library, is a particularly difficult task. "The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, and the like. Instead, these facilities are defined in a

standard *library*, which [developers] compile and link with [their] programs" [81]. In addition to the functions prescribed by the ANSI specification of the C standard library, the GNU implementation provides wrappers for the OS's system calls which provides a convenient method for developers to invoke the functionality provided by the underlying host system. Because the library is so useful to developers, compilers link it into programs by default and very few programs written in the C programming language opt out.[1]

glibc contains over 10,000 C source and header files. The major challenge is not the amount of C code in its implementation. Rather, the major challenge for statically rewriting *glibc* is that the implementation contains over 38,000 handwritten lines of assembly code. The handwritten assembly code accounts for almost 22% of the entire codebase. As mentioned previously (see Section 4.2.1), binary program/library analysis is easier when that program/library is generated by a compiler because it can take advantage of the idioms that a compiler uses to generate code. That type of pattern analysis is not possible when disassembling handwritten assembly code.

The *glibc* implementation of the C standard library also contains other "shortcuts" to improve performance that make disassembly difficult.[2] For instance, internal functions that call one another sometimes pass parameters through registers rather than the stack, in violation of the platform's ABI. This is not necessarily a problem for code/data disambiguation, but it does potentially affect the ability to detect functions. There are also many places where the implementers disregard standard conventions, if not outright standards, of the C programming language. For example, `va_arg()` is invoked on a variadic argument that might not exist, pointers are aliased and function pointers are invoked differently than they are defined.

The unit tests of *glibc* were used to test the robustness of rewriting algorithms as implemented in Zipr. Their unit test system is thorough – it has more than 2500 individual tests. If a rewritten version of the library is able to pass those tests, it is reasonable to assume that the original and the rewritten version are semantically equivalent. Again, this equivalence is the expected behavior because the library was rewritten with the Null Transformation. As long as the two versions are semantically equivalent then the rewriting technique, as implemented by Zipr, is functionally correct.

For the experiment, *glibc* was built and compiled from source and its unit tests were run to establish a baseline. From these results it was possible to collect a list of the tests that passed, failed and did not execute. This list of passes, failures and skips was compared to the passes, skips and failures for unit tests linked against the rewritten version of the library. The executions of the unit tests on the native *glibc* and the statically rewritten *glibc* generated the same results.

---

[1] Of the searchable `Makefile`s stored on GitHub that use the *gcc* compiler, only 10% include the option to build without linking the C standard library.

[2] The information contained in this paragraph comes from direct correspondence with several of the implementers of *glibc*.

libjvm The *libjvm* shared object from OpenJDK was also transformed by the Null Transformation and run through its unit tests to validate the static binary rewriter's design and algorithms. This shared object contains the core of the Java Virtual Machine. On Hosts A and B its implementation is over 12MB in size – almost 5 times bigger than *glibc*. Like *libc*, *libjvm* contains hand-written assembly that implements the transition state between the high-level code of the library, and the JIT-compiled code generated for the Java application that the runtime is executing.

In addition to a confirming that the unit tests passed on the Zipr-rewritten *libjvm*, experiments were conducted to validate that it worked on a real-world application. The rewritten version of the *libjvm* was linked against the system-default *jedit*, a GUI text editor implemented in Java. The success of a variety of manually performed tasks performed on the editor executing against the rewritten runtime (i.e., open a file, save a file, search/replace, insert and delete text, etc.) demonstrated the success of Zipr in transforming *libjvm*. No abnormal slowness in the application and no faults or errors were observed.

Apache Apache was also transformed by the Null Transformation and rewritten by Zipr. Apache is a large C program, containing approximately 56,000 lines of code spread across several shared libraries. The native version given to Zipr for rewriting was compiled with the default compilation flags and included full compiler optimizations and complete stripping of debugging information. Validation tests were also run against a version of the native program compiled without position independent code (PIC).

Additional tests were conducted on the statically rewritten version of Apache configured to service requests in several different ways. One configuration ran service requests in multi-process mode where each request was serviced by separate a process. Another configuration ran separate services processes but each service process handled multiple requests in separate threads.

The rewritten version of the native Apache webserver compiled in each compilation mode and configuration was tested using Apache Jmeter. "Jmeter is a powerful, easy-to-use . . . load-testing tool" for web servers "that supports software load and regression test automation. Jmeter can be used to test static and dynamic [web] resources over a wide range of client/server software" [90].

The tests were executed in an environment where the statically rewritten webserver was loaded with approximately 45 MB of testing content. The testing content consisted of static files (image [png, gif, jpeg, etc.], data [pdf] and HTML files) and dynamic web applications (Common Gateway Interface and JavaScript). The tests were designed to assess the functionality of the statically rewritten Apache webserver by issuing requests with specific header configurations, requests with and without HTTP authentication, requests that attempted to exploit security vulnerabilities, requests that download parts of content, requests that were invalid, etc.

Optimization Level	Passed	Skipped	Failed
O0	486	106	0
Os	486	106	0
O1	486	106	0
O2	486	106	0
O3	486	106	0

Table 5.1: Success and failure of the coreutils unit tests when run against the Zipr-written version of the original tools compiled with different optimization levels. No matter what optimization level was used, the Zipr-rewritten version passed the expected number of unit tests.

Across all configurations and parameters, no failures were observed.

GNU Core Utilitiess The GNU Core Utilities (coreutils) were also transformed by the Null Transformation and rewritten by Zipr. The coreutils are a set of programs fundamental to the operation of a Linux-based host computer. Coreutils comprise the 113 command line tools that every user/system administrator relies on to operate their system. They include tools like `yes`, `test`, `chown`, etc. The coreutils are written in the C programming language and consists of more than 380,000 lines of source code.

Coreutils comes with a set of 592 test programs. The coreutils developers use those tests to prevent regressions from entering the code when new features are added or bugs are fixed. The test programs were executed against the native version of the coreutils and 489 tests passed and 103 were skipped. The 103 tests were skipped because they are long-running tests that are not enabled by default or rely on programs not installed on either Host A or Host B (e.g., Perl's Expect package and SELinux).

After establishing the baseline, the test programs were executed against the statically rewritten versions of the 113 coreutils. 486 of the tests passed and 106 were skipped. Three more test programs were skipped when executing the unit tests against the statically rewritten version of coreutils because of the particular way that the tests are written. For these tests the results are validated using the debugger and breakpoints and success or failure is calculated when the breakpoints are reached. In particular, when the program under test starts, a breakpoint is created from a line in the program's source code. The debugging information embedded in the program under test converts the line of source code into a memory address to create a breakpoint. However, after the program is statically rewritten, the debugging information in the rewritten version of the program no longer matches the debugging information in native version of the program. Therefore, when the test executes on the statically rewritten version, the breakpoint is never reached.

In order to compare with validation results for other static binary rewriters (see Section 6.7), the coreutils package was repeatedly rebuilt, rewritten and retested with different compiler optimization levels.

5.4 Time to Rewrite

Although the goal of design and architecture of the static binary rewriter described in this dissertation is to produce transformed programs/libraries that are efficient (See Part I), it is still important to measure the time that it takes the rewriter to create that program because the time it takes to rewrite may determine whether the system is applicable in certain use cases. For example, consider an online software security system that responds to the discovery of vulnerabilities in real-time. While the system applies the static binary rewriter to patch the vulnerable software to protect against each new vulnerability, the service appears offline to external users. If the time that the service is offline for rewriting is great, an adversary has only to find but not exploit a vulnerability to generate a denial of service attack.

Charting the execution time of Zipr when rewriting programs of various sizes is one way to assess the efficiency of the algorithms and architecture of the static binary rewriter as implemented in the prototype. Of course, it is important to remember that the measurements are taken from a *prototype* implementation where little attention has been paid to optimizations. The optimizations made to the architecture and algorithms of the static binary rewriter described in Part III are aimed at improving the performance of the rewritten program/library and not the rewriter itself.

Table 5.2 shows the execution times of Zipr when rewriting applications and libraries of various sizes after application of two different User-specified transformations. The experimental data were gathered on Host A.

Null in the *Transformation* column refers to the Null Transformation, described in detail in Section 5.3. *P1* in the *Transformation* column refers to the P1 Transformation, an implementation of stack layout randomization (SLR) which "...randomizes the locations of variables stored on the stack and adds random-sized padding between the variables" [181]. By applying the SLR transform to potentially vulnerable software, security architects can protect their software from exploits that rely on knowledge of the specific layout of local variables on the stack at runtime. The use of the P1 Transformation in this experiment is to demonstrate the hypothesis that the time to rewrite a program/library is not significantly changed through the application of a real world transformation (with respect to the Null Transformation).

The data recorded in Table 5.2 demonstrate a linear correlation between the overall time it takes to statically rewrite a program/library and the on-disk size of the program/library when plotted on a log-log scale ($r^2 = 0.848$). When the times to rewrite are analyzed separately according to whether the Null Transformation or the P1 Transformation is applied, there are stronger ($r^2 = .862$) and weaker ($r^2 = .845$) linear correlations when plotted on a log-log scale, respectively. See Figure 5.1. That the overall correlation is weaker than the individual correlations indicates that the time to rewrite depends on the User-specified Transformation applied. The difference, however, in the actual time to rewrite is not great. On average, it takes 0.086 seconds

Program/ Library	Size	Transformation	Real Time	User Time	System Time	Reassembly Real Time
Small						
sha512sum	256	Null	19.115	10.591	1.077	1.841
		P1	33.939	24.231	1.397	2.406
numfmt	260	Null	16.622	9.206	1.126	1.718
		P1	21.089	9.727	1.054	2.458
chmod	264	Null	17.504	8.983	1.067	1.785
		P1	23.023	13.083	1.266	2.340
shred	264	Null	17.295	7.549	0.986	1.672
		P1	21.865	11.700	1.241	2.322
shuf	264	Null	17.586	7.772	0.994	1.665
		P1	24.074	12.261	1.333	2.114
Medium						
gcc	760	Null	88.126	62.691	2.886	13.862
		P1	103.664	83.391	3.184	17.925
nginx	856	Null	113.198	85.749	3.661	22.254
		P1	152.101	121.560	4.394	29.878
apache2	640	Null	63.493	44.190	2.549	11.463
		P1	83.381	63.420	3.057	16.094
Large						
git	1544	Null	225.982	181.219	6.275	52.940
		P1	322.170	268.894	8.253	81.275
mysql	3392	Null	121.659	93.620	3.485	19.971
		P1	164.995	134.016	4.573	28.751
replace	2820	Null	60.445	46.670	2.162	6.172
		P1	71.712	55.725	2.393	7.305
Extra Large						
libjvm	12196	Null	2492.615	2243.324	31.917	528.448
		P1	3068.040	2802.731	35.437	523.774
doxygen	9816	Null	3110.006	2969.137	20.810	413.782
		P1	4458.965	4282.430	30.810	873.004

Table 5.2: Times to statically rewrite programs/libraries of different sizes after transformation with either the Null or P1 User-specified Transformations using the architecture and algorithms described to this point in the dissertation as implemented in a prototype. All times are in seconds and all sizes are in kilobytes.

per kilobyte and 0.115 seconds per kilobyte to rewrite programs/libraries transformed by the Null and P1 Transformations, respectively.

Unfortunately, a linear correlation between the size of the input program/library and the time to rewrite when plotted on a log-log scale does not adequately explain all the results. In particular, it does not explain why *replace* takes so much less time to rewrite than *git* and *mysql*, programs of similar size.

Investigation of this outlier led to the discovery that the time it takes to rewrite is more closely associated with the number of dollops in the IR submitted to the algorithms of the Reassembly Phase than the on-disk size of the original input program/library. See Figure 5.2. There is a stronger linear correlation between the overall time it takes to statically rewrite a program/library and the number of dollops in the original program/library when plotted on a log-log scale ($r^2 = 0.913$) than there is between on-disk size and the time

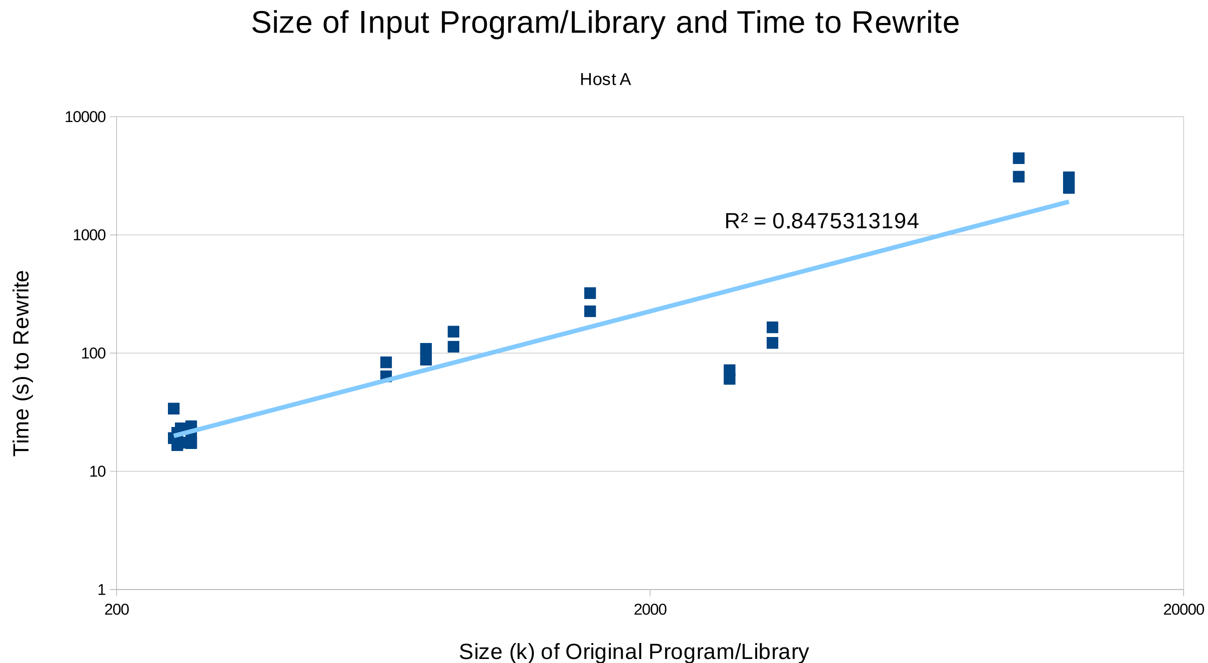

Figure 5.1: The relationship between the size of the input program/library and the time it takes to statically rewrite using the Zipr prototype implementation. Data for programs rewritten after transformation by both the Null and P1 Transformations are included in the graph. Each axis is logarithmically scaled.

to rewrite ($r^2 = 0.848$). When the times to rewrite are analyzed separately according to whether the Null Transformation or the P1 Transformation is applied, there are stronger ($r^2 = .920$) and equivalent ($r^2 = .913$) linear correlations when plotted on a log-log scale, respectively.

The data indicate that although, upon first analysis, on-disk size seems to be a good predictor of the time it takes to rewrite, the better predictor of the time to rewrite is the number of dollops in the IR submitted to the algorithms of the Reassembly Phase. That this is true is not surprising. The number of dollops in the IR passed to the algorithms of the Reassembly Phase depends on the complexity of the program/library's CFG upon which the runtime of the algorithms in the Disassembly, IR Construction and Reassembly phases depend.

This experiment proves the hypothesis that the application of a real world transformation to the original program/library would not significantly affect the time to rewrite. However, the data indicate that the hypothesis will not hold for all User-defined Transformations. The results indicate that the complexity of the CFG affects the time to rewrite. The P1 Transformation simply transforms the composition of the stack at runtime and does not change the original program's CFG. Therefore, the time to rewrite is not significantly changed. However, where a User-definted Transformation adds complexity to the CFG, the time to rewrite will increase with respect to the time to rewrite the same program with the Null Transformation.

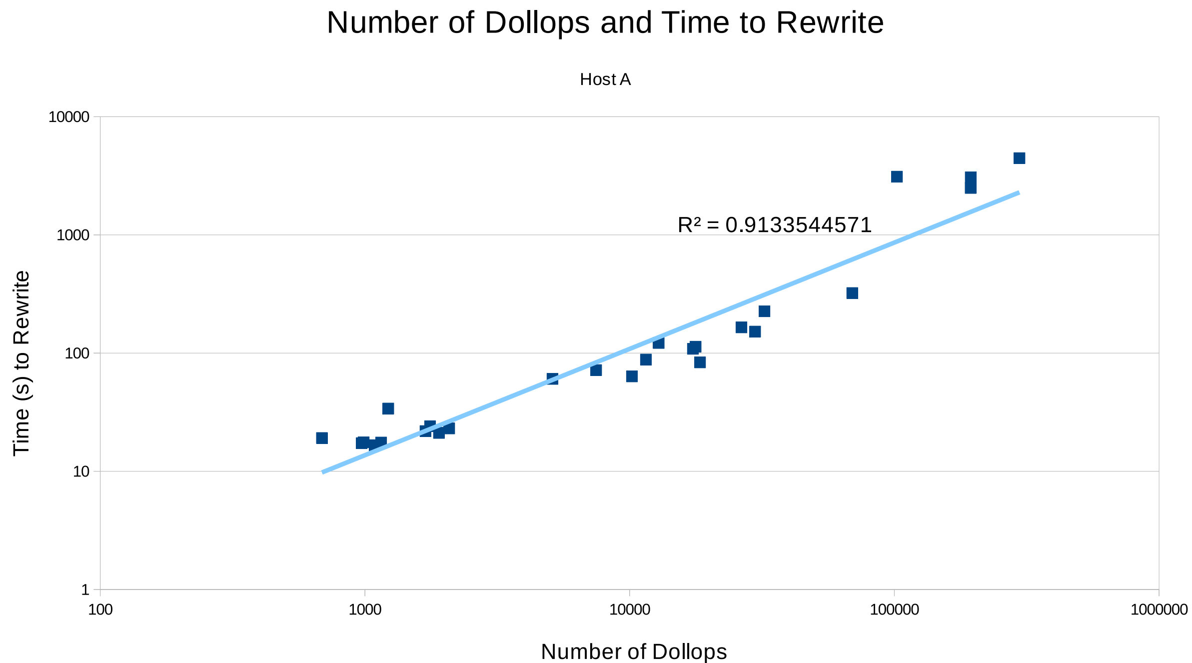

Figure 5.2: The relationship between the number of dollops in the input program/library and the time it takes to statically rewrite using the Zipr prototype implementation. Data for programs rewritten after transformation by both the Null and P1 Transformations are included in the graph. Each axis is logarithmically scaled.

5.5 SPEC

The Software Performance Evaluation Company (SPEC) publishes a benchmark known simply as SPEC. SPEC is benchmark suite that contains 19 tests to measure "compute-intensive integer performance" and 12 tests to measure "compute-intensive floating point performance." [208] SPEC tests "actual end-user applications, as opposed to [...] synthetic benchmarks" and it is used throughout industry and academia as a "common reference point" [208] for performance measurement.

The evaluation of the prototype implementation on the SPEC performance benchmark suite was done in two parts. The first, performance, measured the overall speed of the execution of the benchmark. The result of the performance tests measured the overall speed of the programs, "how fast the computer completes a certain task" [214]. These tests account for everything from the performance of the compiler to the CPU's performance to the memory subsystem throughput [214]. The graphs presented here represent the difference between the performance of the native and statically rewritten versions of the programs in the SPEC benchmark suite. A value greater than 1.0 represents a decrease in performance and a value less than 1.0 represents an increase in performance.

The second, memory usage, measured two specific aspects of the execution of the benchmark: the RSS and the number of minor page faults.

At a particular time during the execution of a program, the program's resident set are the pages of a system's memory (RAM) allocated to that program. As a program executes, it accesses its code and data. When a program accesses a memory location for the first time, the program *faults* and the OS *pages in* or *swaps in*[3] the necessary code/data from disk, where the program and its data are stored in between executions, into memory (For a complete description of paging in OSes that virtual memory systems, see, e.g., [217] or [200]. For a description of how *demand paging* is implemented in the Linux Kernel, see [44]).

The OS supports the illusion of executing multiple programs at the same time. In reality, there can only be a single program operating at a time on a CPU. To maintain the appearance of multiple processes operating simultaneously, each program operates for a fixed period of time before being paused (For a complete description of OS support for timesharing, see, e.g., [217] or [200]). Only the active process currently executing on the host's CPU needs to have its resident set in memory. The OS *pages out* or *swaps out* the memory used by resident sets of the paused, inactive programs if the OS needs that space to support the active program. The smaller each program's resident set, the less likely it will be that a paused program's memory will be paged out (For a description of the process by which memory is paged out in the Linux Kernel, see [44]).

When an active program accesses a page of memory that was paged out, another fault is generated and the OS pages in the required code/data again. Because a program cannot continue execution without access to its code/data, when a program faults it is paused until the OS can complete the paging operation. The effect is that faults increase the amount of time that it takes a program to complete its operations and decreases performance.

Again, the smaller each program's RSS, the less likely it will be that a paused program's memory will be paged out which, in turn, means that there will be fewer faults when the system is executing many programs simultaneously. Therefore, this evaluation concerns the maximum RSS, the highwater mark of the size of the resident set during a program's execution. Chapter 7 contains a more detailed discussion of the RSS and faults on performance.

The graphs presented here show the difference between the number of faults and the maximum RSS of the native and statically rewritten versions of the programs in the SPEC benchmark suite. A value greater than 1.0 represents an increase in the maximum RSS or number of page faults for the programs in the SPEC benchmark while a value less than 1.0 represents a decrease.

[3] Although originally distinct concepts, Silberschatz et al. admit that today the two terms are used interchangeably [200].

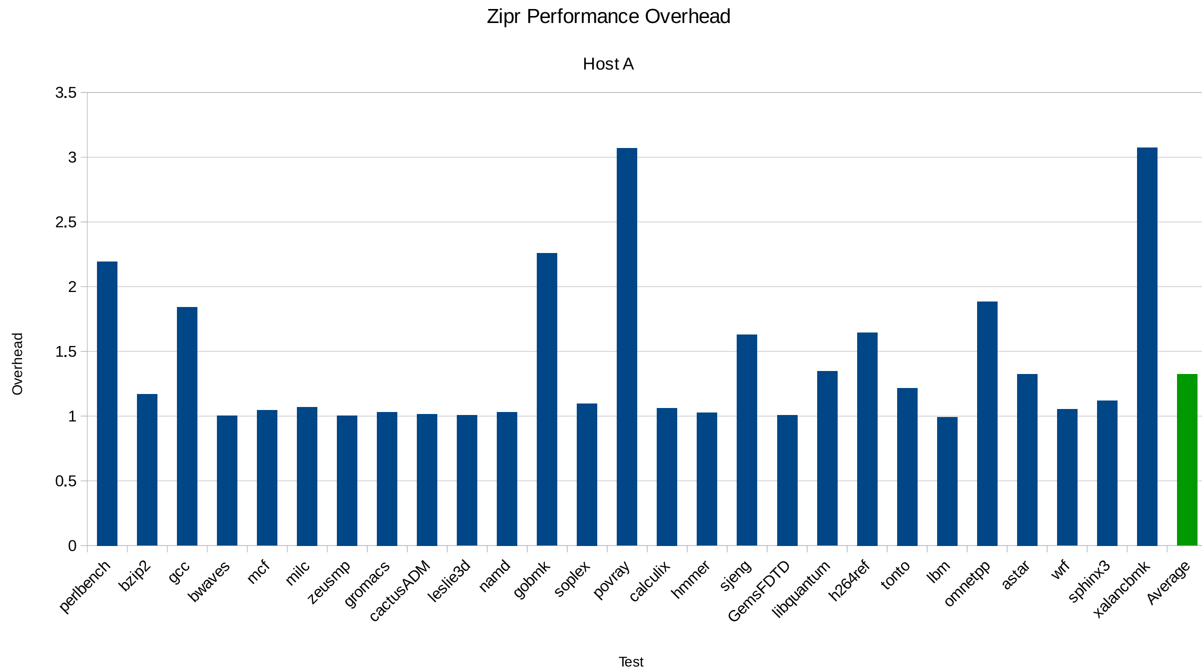

Figure 5.3: Performance overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A.

5.5.1 Performance

Figures 5.3 and 5.4 show the performance overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite when tested on Host A and B, respectively. On Host A, several benchmarks showed improved performance. For the remainder, overhead ranged up to 308% with an average of 1.323x. On Host B, several benchmarks also showed improved performance. For the remainder, overhead ranged up to 270% with an average of 1.285x.

`povray`, `perlbench`, `gobmk` and `xalancbmk` have the highest performance overhead among the programs in the SPEC benchmark suite in this particular configuration.

There are several reasons for the high overhead of the `perlbench` application. The first is the transformation of a frequently executed `call` into a `push`/`jmp` combination in the statically rewritten binary (see Section 4.2.2 for the rationale behind converting `call`s). The most commonly executed sequence of instructions in the benchmark application setup and invoke an indirect call to a function whose address is stored in memory. The first instruction in the sequence in the original program copies the address of the function from memory into a register (`rax`). The second is the indirect `call` through `rax`. In the statically rewritten binary program, this is converted into three instructions. The first of the transformed instructions is the equivalent of the first instruction in the sequence in the original program – it simply loads the address of the function into a register.

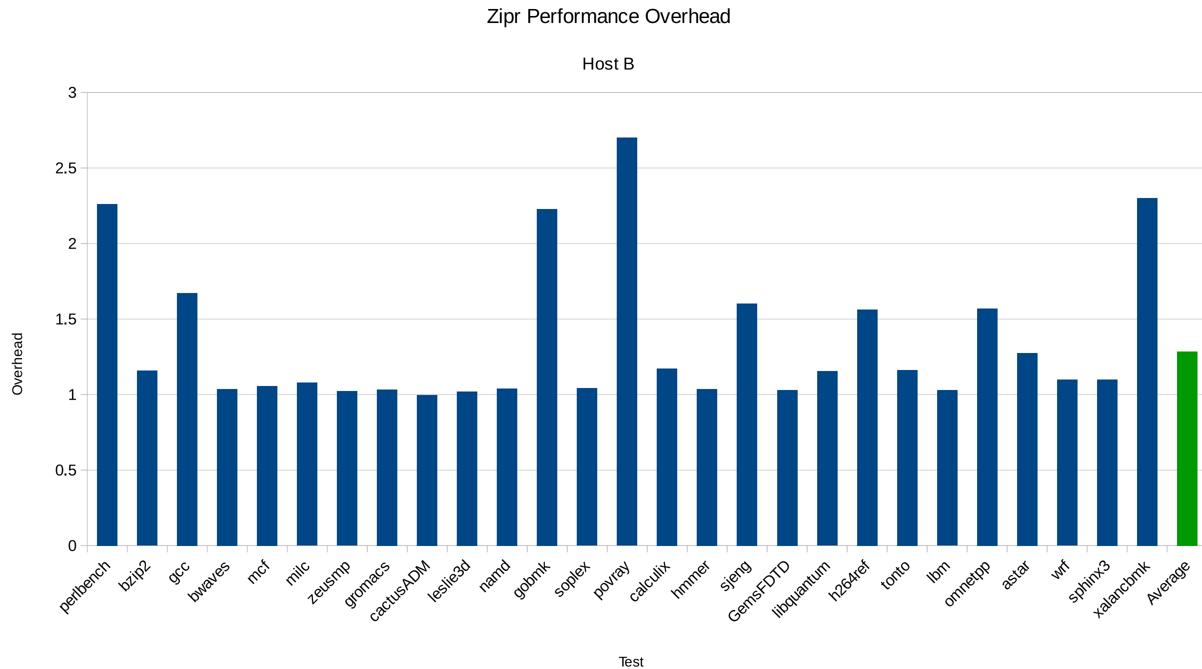

Figure 5.4: Performance overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B.

The second instruction in the statically rewritten sequence pushes the return address on to the stack. The third instruction in the statically rewritten program sequence then `jmp`s to the address stored in that register.

There are serious performance implications to replacing a `call` with a combination of `push` and `jmp`. Most of the common CPU architectures employ a return address stack that works like a branch predictor to optimize `ret`s from functions [107]. There is a significant amount of literature on the performance benefits of the CPU's call/return branch predictor (e.g., [201]). The predictor uses a stack data structure to match calls with returns to predict the destination of a return at runtime and speculatively begin executing instructions at the return site even before the return is reached [226]. See Section 12.2.2 for additional information about the return address stack and the call/return branch predictor and its performance impact.

To demonstrate the benefit of this optimization, consider the code snippets shown in Table 5.3. The two programs are semantically equivalent. They both invoke `function 0x10000000` (268,435,456) times. The only difference is whether `function` is invoked with a `call` or a combination of `push/jmp`. On modern hardware, the version that `calls function` runs almost four times as fast as the program that invokes `function` with a combination of `push/jmp`. For an invocation of the `perlbench` benchmark using the test input for SPEC2006, the transformed sequence accounts for 1.24% of all the instructions executed. The combination of the frequency of this instruction and the penalty from the conversion of the `call` into the

<table>
<tr><th>Call</th><th>Push/Jmp</th></tr>
<tr><td>

```
function:
ret
_start:
mov rcx, 0x100000000
mov rbx, 0x0
loop:
call function
after:
inc rbx
cmp rbx, rcx
jle loop
mov rbx, 1
mov eax, 1
int 0x80
```

</td><td>

```
function:
ret
_start:
mov rcx, 0x100000000
mov rbx, 0x0
loop:
push after
jmp function
after:
inc rbx
cmp rbx, rcx
jle loop
mov rbx, 1
mov eax, 1
int 0x80
```

</td></tr>
</table>

Table 5.3: A pair of semantically equivalent programs. The only difference is whether the program invokes `function` with a `call` or a combination of `push/jmp`. The difference has performance implications.

`push/jmp` combination accounts for a significant amount of the overhead.

The second most frequently executed sequence of code in the `perlbench` benchmark is a dollop targeted by a link from a pinned address. The frequently executed dollop accounts for 1.24% of the instructions executed during an invocation of the `perlbench` application on a test input and the link from the pinned address that targets those instructions accounts for 0.4% alone.

In the `perlbench` benchmark, there are several sequences of frequently executed instructions that span one or more dollops that the algorithms of the Reassembly phase do not coalesce (see Section 4.4.4). As described earlier, not being able to coalesce a dollop with its fallthrough adds overhead at runtime because of the introduction of links. In these highly executed sequences, the dollops are not coalesced because either there is not enough space in the area assigned for the actualization of the dollop to hold its fallthrough or because the fallthrough dollop is already actualized. The fact that the highly executed sequence of instructions spans dollops and their fallthroughs implies that performance would improve if those dollops were actualized consecutively and the links were removed. The presence of the link between the dollops and their fallthroughs adds overhead and the performance impact is exaggerated when the dollops are frequently executed.

The reasons for the high overhead of the `xalancbmk` application are the same as the reasons for the high overhead of the `perlbench` application. In the `xalancbmk` application, there are several highly executed sequences of instructions that span dollops that are not coalesced because of space constraints or because the fallthrough dollops were already actualized. There is also a highly executed sequence of instructions that contains a function call that is converted into a `push/jmp` in the statically rewritten binary. Finally, there is a highly executed sequence of instructions that are part of a dollop targeted by a link at a pinned address. In

the original program, the sequences affected by these undesirable changes account for .92%, 2.62%, 1.619% and 1.12% of all instructions executed for a test input, respectively.

There are several reasons for a high overhead of the `povray` application. The second most commonly executed sequence of instructions contain parts of three different dollops. In the original code, these instructions account for 6.22% of all instructions executed during a test. In the rewritten version of the application, there is one failure to coalesce the dollop and its fallthrough due to a lack of space. The same sequence of instructions contains two dollops that are targeted by links at pinned addresses. Finally, this highly executed sequence contains a function call converted to a `push/jmp`. With the addition of these instructions, the sequence accounts for 7.34% of all the instructions executed by the statically rewritten benchmark application for a particular test.

There is a single reason for the high overhead of the `gobmk` application but it affects multiple different highly executed sequences of instructions. The problem is that the highly executed sequence of instructions spans two different dollops that are not coalesced. The dollops are not coalesced in one instance because there is a lack of space necessary to place the fallthrough dollop in the same block of free space and in the other because the fallthrough dollop is already actualized elsewhere. In the original program, this highly executed sequence of instructions accounts for just under 1% of all instructions executed for a particular test input. In the statically rewritten version of the benchmark application, the sequence accounts for 1.75%.

The investigation into the causes of the performance overhead reveals three common causes: the presence of links to dollops from pinned addresses, the conversion of `call`s to `push/jmp`s and a failure to coalesce fallthrough dollops either because of lack of space or because the fallthrough is actualized elsewhere. Unfortunately, the number of pinned addresses, the number of conversions of `call`s to `push/jmp`s and the number of a failures to coalesce fallthrough dollops either because of lack of space or because the fallthrough is actualized elsewhere does not correlate with an increase in performance. See the graphs in Figure 5.5.

Given the lack of correlation, it seems that the problem is not the *quantity* of these problems' manifestation during the rewriting process but the *quality*. If these problems manifest themselves in highly executed sequences of instructions (their quality) then performance suffers greatly. Learning the quality of the problems statically is difficult. Fortunately, simply reducing the quantity of these problems makes it less likely that they will appear in highly executed sequences of instructions. It is this reduction in the quantity of these negative affects that is explored in Part III of this dissertation.

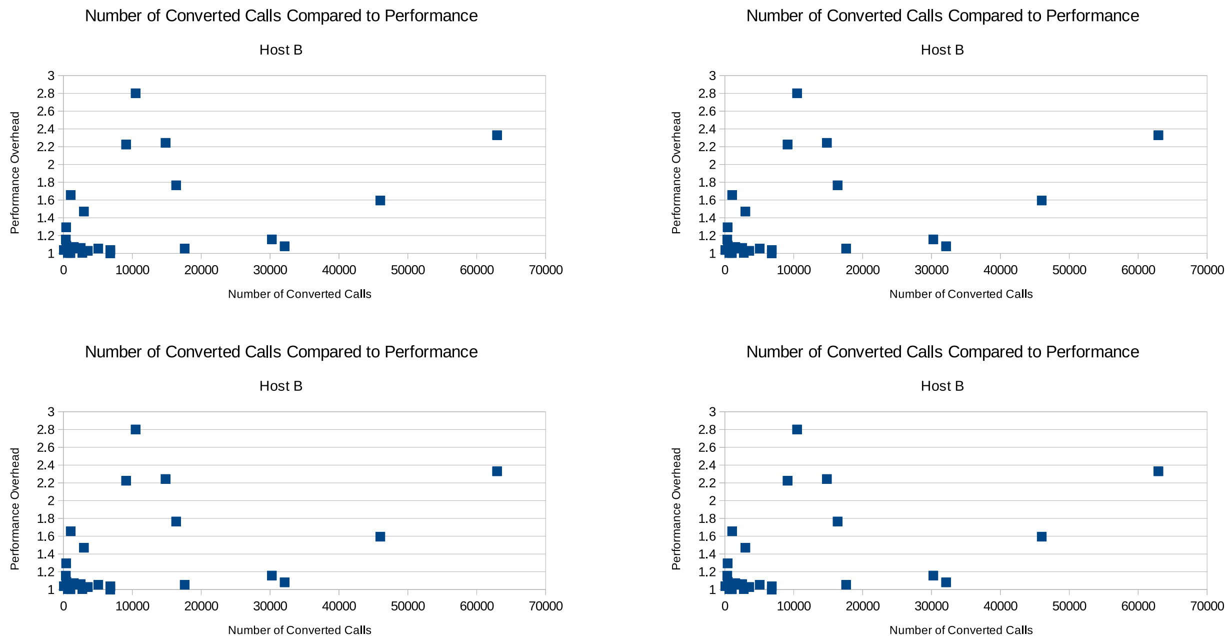

Figure 5.5: There is no correlation between performance and the common causes of performance overhead identified by the investigation into the `perlbench`, `povray`, `xalancbmk` and `gobmk` applications in the SPEC2006 benchmark suite.

5.5.2 Memory Usage

Figures 5.6 and 5.7 show the maximum RSS overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. On Host A, several benchmarks showed that the statically rewritten versions of the SPEC benchmark programs have a smaller maximum RSS compared to the native versions. For the remainder, the overhead ranged up to 9.3% with an average of 1.012x. On Host B, several benchmarks showed that the statically rewritten versions of the SPEC benchmark programs have a smaller maximum RSS compared with the native versions. For the remainder, the overhead ranged up to 11.4% with an average of 1.013x.

There is a significant difference in the maximum RSS for `cactusADM` and `sphinx3` between Hosts A and B. To understand the source of this difference, additional information is necessary. The number of page faults needed *to access code* during program execution will illuminate whether the difference in RSS is intrinsic to the program or contingent upon the memory system. To reiterate, the experiments whose results are shown in Figures 5.8 and 5.9 compare only the number of faults needed to page in program code. These experiments do not measure the number of page faults required to access the data needed for the computations of the benchmarks. Measuring this specific type of page fault makes it possible to compare the impact of the program's reassembly on its performance. In this dissertation, evaluation of page fault overhead will always

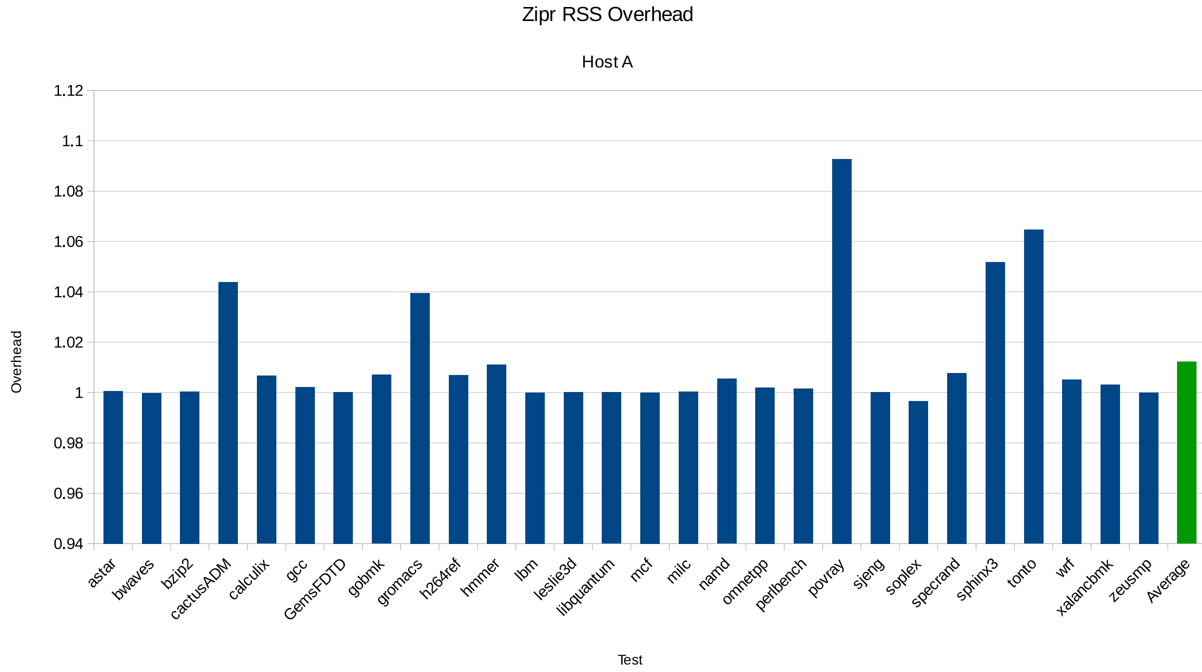

Figure 5.6: Maximum RSS overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A.

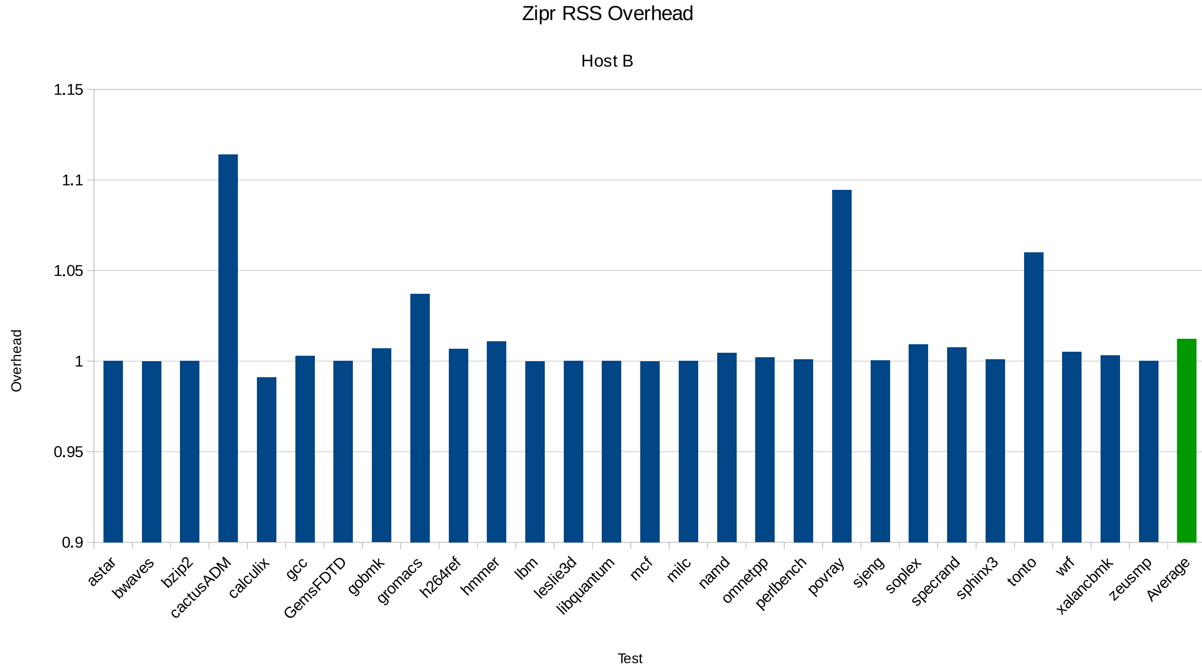

Figure 5.7: Maximum RSS overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B.

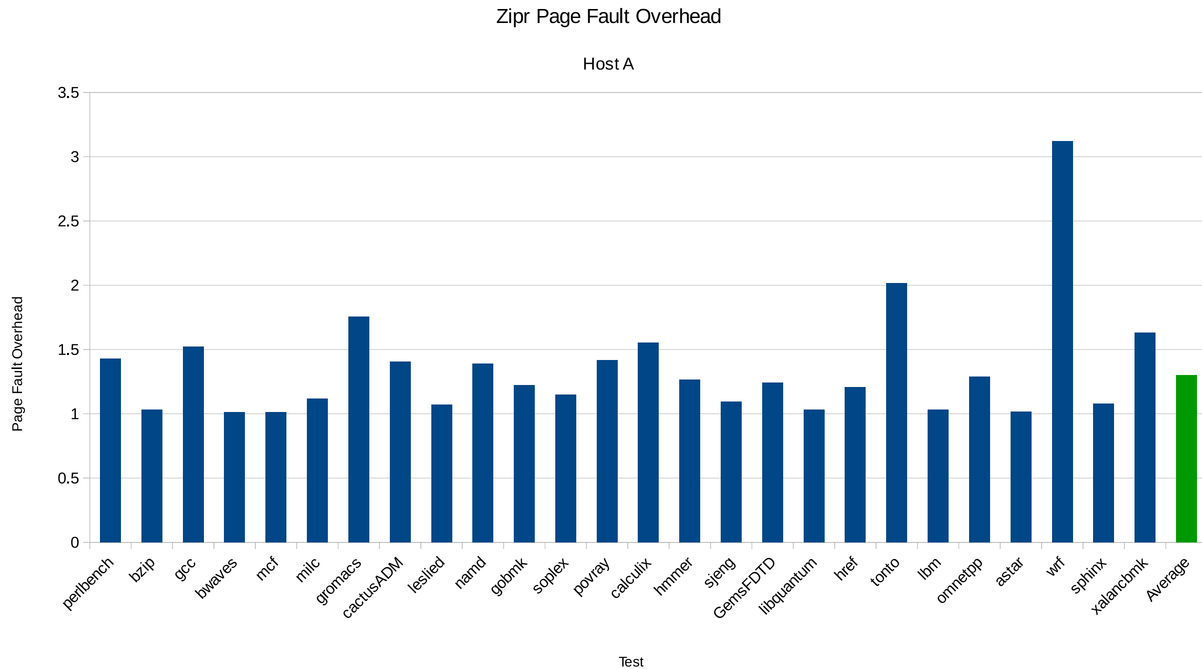

Figure 5.8: The minor page fault overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host A.

be restricted to those minor page faults required to access code, unless otherwise specifically mentioned.

Figures 5.8, 5.9 show the page fault overhead for code memory accesses for statically rewritten versions of each of the programs in the SPEC benchmark suite. On Host A, the overhead ranged from 1.7% to 312%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.302x more page faults than the native versions. On Host B, the overhead ranged from less than 1% to 291%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.293x more page faults than the native versions on Host B.

While the relative overheads of page fault overhead for Host A and Host B are nearly identical, so too are the absolute number of page faults. This means that the pages of memory in the program's resident memory that hold program code are nearly identical.

These experiments point to an explanation for the relative difference in the performance of the `cactusADM` and `sphinx3` benchmarks. That the experimental data show the number of minor page faults needed to access code during program execution are identical along with the inconsistency in the RSS size of the two benchmarks on the two different hosts means that the difference is caused by the way that the two benchmarks access the data used in their computations. The question, then, is, Does the additional memory resident during computation reflect something intrinsic to the process of statically rewriting the application

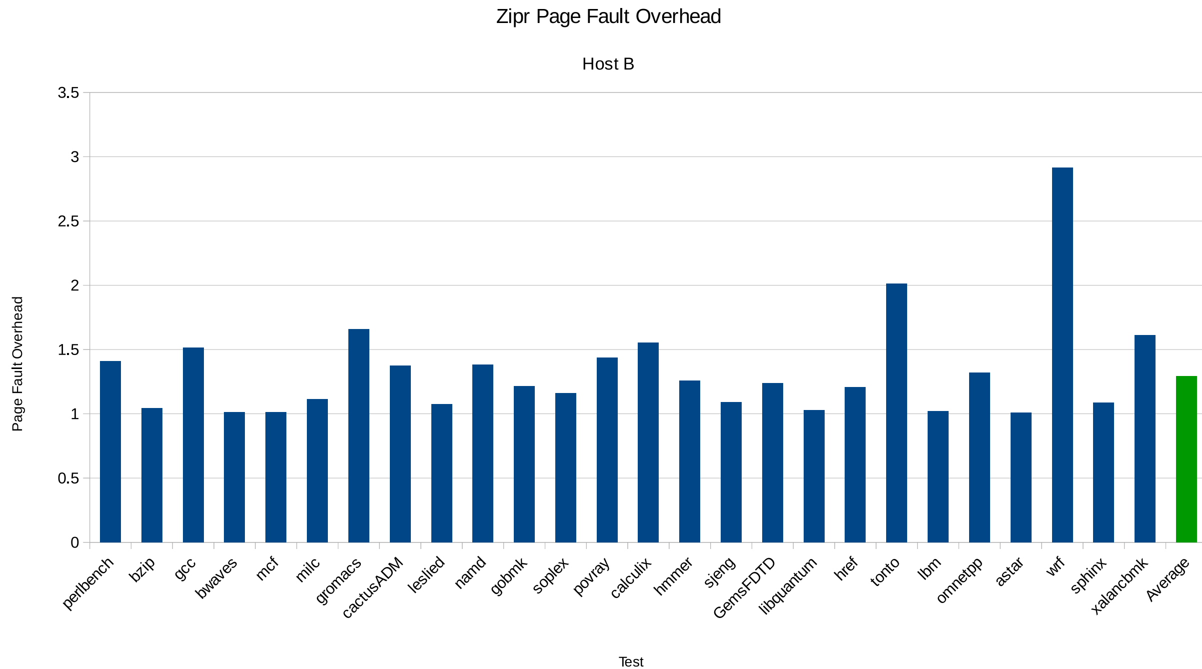

Figure 5.9: The minor page fault overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host B.

or something extrinsic that is beyond the control of the rewriter?

The answer lies in the test systems' use of huge memory pages [15]. Various Intel hardware platforms, including the ones for the experimental hosts, support multiple sizes of memory page tables: 4 kilobytes, 2 megabytes and 1gb [196, 109]. Besides fewer page faults, using larger pages decreases the number of TLB misses and there is extensive research that demonstrates that the efficient use of the TLB has a significant impact on performance [34]. The Linux kernel added so-called transparent support for huge memory pages to help applications transparently benefit from the hardware's support for these huge pages [16]. Starting in early 2011, when an application pages in anonymous memory (heap space) that is otherwise unmapped in the TLB (according to the *page middle directory* [44]), the request is preferentially satisfied with a huge page. If a huge page is unavailable for any reason, then the page fault is fulfilled with a page of memory of typical page size.

The allocation of a huge page is not always preferential, however. As the documentation for transparent huge page support in the Linux kernel admits, "[in] certain cases when hugepages are enabled system wide, application [sic] may end up allocating more memory resources" [16] which is exactly what happened for the `cactusADM` and `sphinx3` benchmarks. For `cactusADM` on Host B, several page faults were handled with huge pages and resulted in a larger maximum RSS when compared with the Maximum RSS for `cactusADM` on

Host A. The same is true for `sphinx3` on Host A – several page faults were handled with huge pages and resulted in a larger maximum RSS when compared with the Maximum RSS for `sphinx3` on Host B.

Because of the support for transparent huge pages on the experimental hosts, the results for the measurement of maximum RSS throughout the dissertation must be treated with suspicion. Because transparent huge pages are only applicable to requests for anonymous memory and not memory backed by files (like the ones for program code), the results for the minor page faults triggered by code accesses are not affected and are, therefore, useful for assessing the performance of the algorithms of the Reassembly phase of the static binary rewriter.

5.5.3 Instruction Cache Usage

The instruction cache is another example of the performance-enhancing optimizations that hardware manufacturers include in their products. The instruction cache is dedicated to storing the most-executed instructions and fetches sequences of instructions speculatively to improve access times. One prominent optimization guides puts it this way: "If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor [with a 16-byte instruction cache] will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16" [78] to take advantage of the instruction cache.

Although the Intel Architectures Optimization Reference Manual dedicates little space to discussing the impact of the instruction cache on program performance [107], modern compilers realize its importance and optimize code layout to use the cache efficiently. In GCC, the alignment of functions along instruction cache lines is performed by default for optimization levels O2 and O3 [19]. The GCC compiler also includes options for aligning loops, labels and jumps in a similar manner and those optimizations are included by default at the same optimization levels [19].

Fog opines that, "[i]n most cases, the effect of code alignment is minimal" [78]. However, experts disagree. On both server and embedded platforms, the effect of the instruction cache can have a major impact on performance. For servers, "[i]nstruction-cache misses account for up to 40% of execution time in online transaction processing (OLTP) database workloads" [91]. On embedded platforms, the effective use of the instruction cache not only improves performance but it also improves energy efficiency [221].

For these reasons, it is important to understand the effect of the algorithms of the Reassembly phase on the utilization of the instruction cache. The Intel family of processors offer hardware counters that measure the miss rate of the first level instruction cache. First level cache has the smallest capacity but fastest access

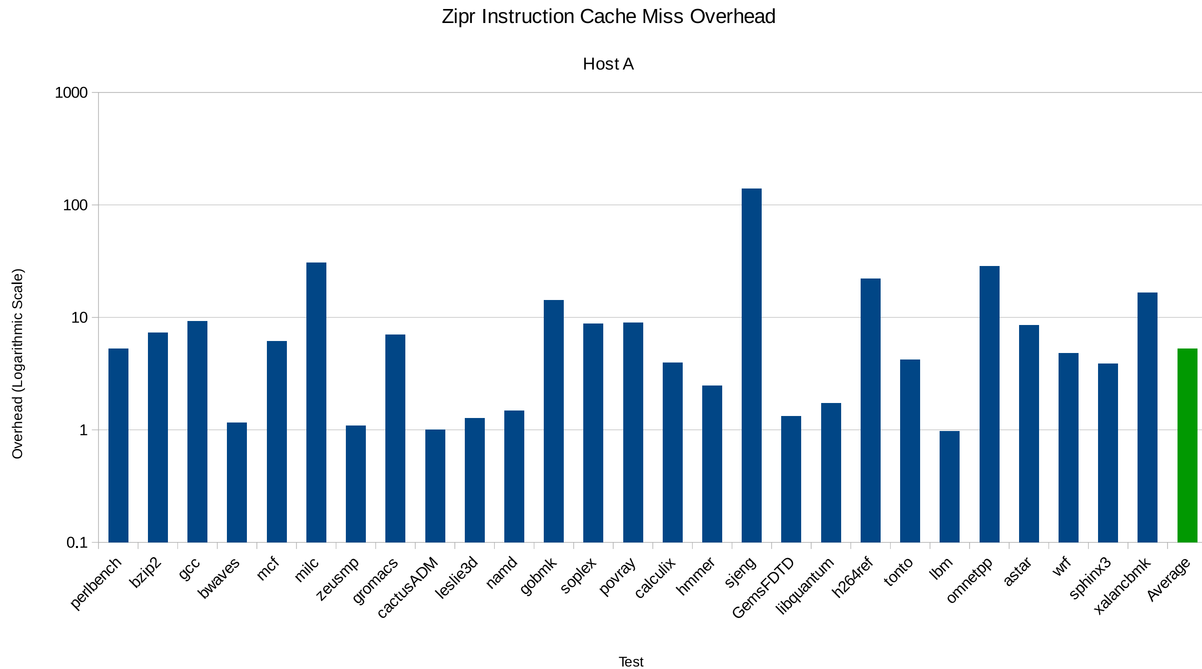

Figure 5.10: The instruction cache miss overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host A.

times of the cache hierarchy. The counters provide the data necessary to quantify the effects of the algorithms of the Reassembly phase on the instruction cache miss rate.

Figures 5.10 and 5.10 show the instruction cache miss rate overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. For Hosts A and B, the statically rewritten programs incur 5.25x and 5.61x as many instruction cache misses as their native counterparts, respectively. Because of the small size of the level one instruction caches on Hosts A and B (32K per core), the number of cache misses for the native version of the programs in the SPEC benchmark suite is tremendous. For example, in an execution of `gobmk` on Host B there are more than 18,000,000,000 first level instruction cache misses. Nevertheless, an increase by a factor of five in the miss rate is not good for performance. The algorithms discussed in Part III are designed to address this overhead.

5.5.4 Filesize Overhead

One of the primary goals set forth in Part I is for the architecture and algorithms of the static binary rewriter to generate rewritten programs/libraries with minimal on-disk overhead. If the algorithms and architecture produce statically rewritten programs and libraries that are much larger than the original program, it may be impossible to deploy those programs/libraries on systems with minimal storage capacity. Figures 5.12

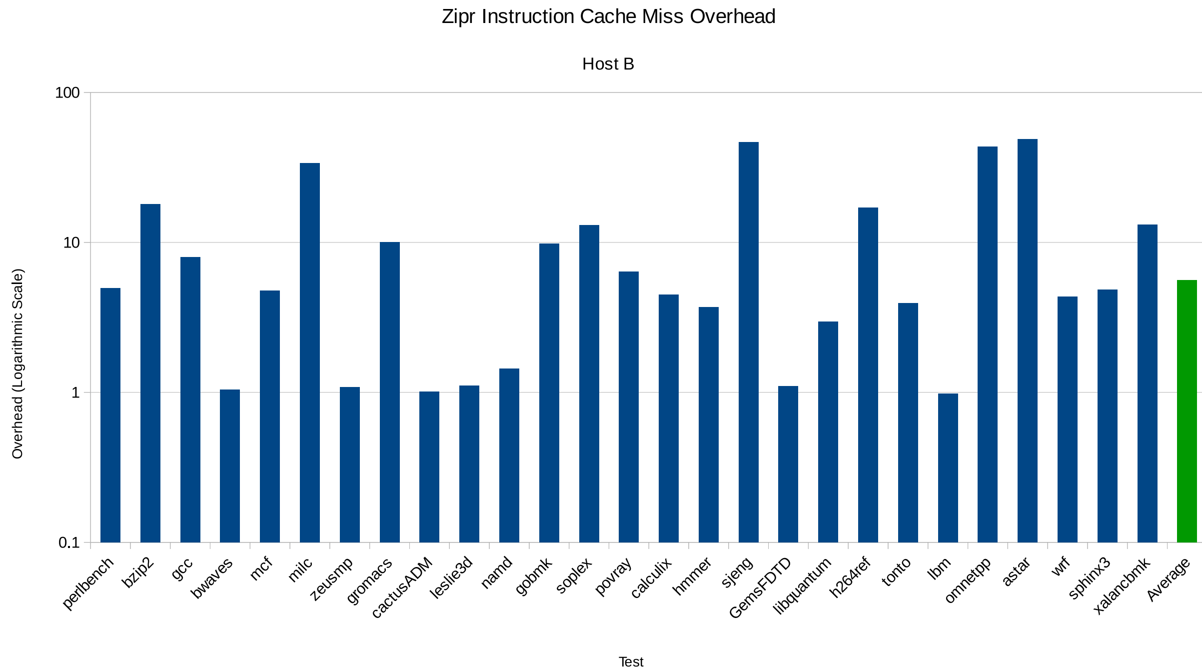

Figure 5.11: The instruction cache miss overhead for the applications of the SPEC benchmark suite when reassembled using the default algorithms of the Reassembly phase. These results are for Host B.

and 5.13 show the filesize overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. For Host A, the average filesize overhead of the applications of the SPEC benchmark suite was 9.40%. For Host B, the overhead was 9.05%. In both cases, the filesize overhead is minimal which validates the ability of the architecture and algorithms of the static binary rewriter as implemented by Zipr to produce statically rewritten programs with low on-disk overhead. Figures 5.12 and 5.13 show the optimizations described in Part III will also reduce on-disk overhead.

5.6 CGC

In 2016, DARPA organized the CGC [230]. While CGC was structured as an autonomous hacking competition, the overall goal was to demonstrate the feasibility of identifying vulnerabilities in programs and patching those flaws without human intervention. Competitors were only given access programs in binary form and the binaries were stripped of all debugging and metadata information. A successful entrant in the competition had to "[r]everse engineer unknown software" (i.e., software without source code or other metadata) and "[h]eal weaknesses without sacrificing [f]unctionality, [or] [c]orrectness [or] [p]erformance" [230]. The tools built for the administrators of the CGC are ideal for testing whether the algorithms and architecture of the static

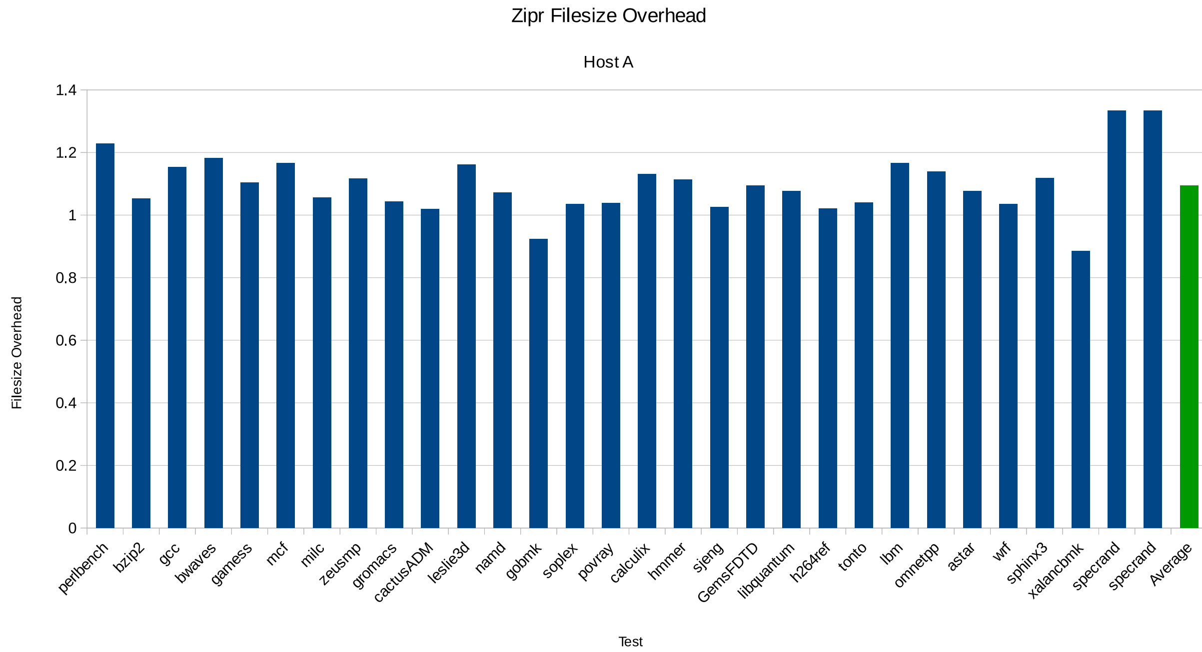

Figure 5.12: Filesize overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host A.

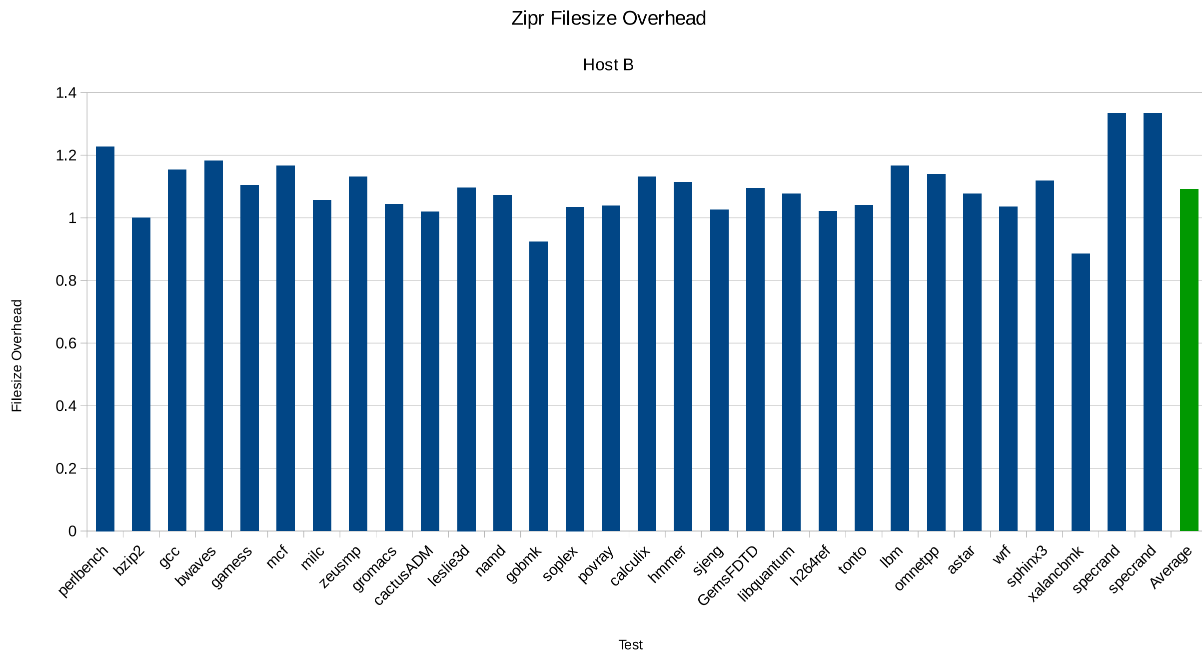

Figure 5.13: Filesize overhead for SPEC benchmark suite programs when reassembled using the default algorithms of the Reassembly phase. These results are for Host B.

binary rewriter described herein can produce rewritten binaries that have acceptable filesize and performance overhead with respect to the original program.

To understand the CGC tools and their utility in measuring overhead, it is necessary to understand the structure of the challenge. Each competitor in the CGC built a cyber reasoning system (CRS) whose job it was to protect a set of previously unseen, statically-linked, x86, 32-bit binary programs (known as challenge binaries or CBs) during an autonomous game of capture the flag (CTF). The CRS had access to only the binary CBs – no source code, debugging information or other metadata was available to the CRS for analysis. The CBs were designed from scratch for the competition by various so-called performers.

A CRS defended a CB by either a) adding additional security or reliability or b) identifying vulnerabilities and repairing them and producing an augmented CB called a replacement challenge binary (RCB).[4] Vulnerabilities in the CBs could be validated by demonstrating successful control-flow hijacking and/or information disclosure attacks.

For the CTF game, CRSs were scored on the performance characteristics of the RCBs. Those scores were compiled and the team with the best score at the end of the game was declared the winner of the challenge final event (CFE). The characteristics that determined the score of an RCB were calculated relative to the performance of the CB. To deterministically measure the baseline performance, CGC employed a) a specialized operating environment known as the DARPA Experimental Cyber Research Evaluation Environment (DECREE) and b) sample inputs/outputs, known as pollers, for each CB.

DECREE is a simplified, restricted Linux-based OS (e.g., it supports only seven system calls, imposes limitations on interprocess communication and provides no filesystem or network access) [230].

A CB poller is an input/output to a CB. "Service polls are intended to validate the performance and functionality of challenge binaries in order to test the efficacy and impact of reformulation performed by competitors. Similar in ideals to unit tests, service polls should be written to validate the interactivity and complex state machines implemented by CBs" [18]. Because the correct behavior of a RCB with respect to the pollers determined the competitors' scores, it can be assumed that the pollers exhaustively test all the valid execution paths of the CB. In fact, DARPA required that the pollers "...provide coverage for all *major or significant* aspects of binary functionality. Code segments that are not exercised by any poll may effectively never be executed and may therefore be considered superfluous" [21]. For testing prior to the final competition, performers were required to have 1,000 pollers for each CB. For the final competition, performers were required to have 1,000,000 pollers for each CB. The practical value of having 1,000 or even a million inputs for each CB is that an evaluation using those inputs accounts for 1,000 or a million executions – enough

[4] Alternatively, a CRS could also deploy a network filter.

to account for the unintentional introduction measurement bias from too few executions on a restricted number of invocations.

Aside from the practical benefit that the CGC tools provides for calculating statistics about the RCBs, the poller's comprehensiveness is *the* reason that CGC tools are useful for evaluating the entire design and architecture of the static binary rewriter proposed in this dissertation. It is especially important for the development and evaluation of the Profile Layout algorithm (see Chapter 8). In combination, the DECREE environment and the pollers make it possible to take deterministic and reproducible measurements (e.g., execution time, file size, runtime memory usage,[5] and functionality) of each CB and RCB. Each of the characteristics is calculated for each of the poller inputs/outputs for each of the CBs and RCBs.

Calculations for performance and on-disk and memory overhead are done using traditional metrics like RSS and execution time. For every poller input, these metrics are gathered. The RCB's overall performance and on-disk and memory overhead are based on an average of those metrics across all of the pollers. The functionality overhead is calculated based upon the number of pollers for each CB whose input matches its output when given to the RCB. Each poller input is given to each RCB and if the poller output matches the actual output from the RCB, then the functionality score for the poller on that RCB is 1. If there is a mismatch, the functionality score for the poller on that RCB is 0. The RCB's overall functionality overhead is based on an average of those scores across all of the pollers.

DARPA had strict targets for each of these characteristics: 20% overhead for file size, 5% overhead for performance and memory and limited loss of functionality. Any overhead beyond those thresholds negatively impacted the *availability* score of an RCB with respect to the original CB. The availability score is a good proxy for the functionality, performance and on-disk and runtime memory overhead of the RCBs and serves as a good shorthand for evaluating the ability of the architecture and algorithms described herein to efficiently rewrite a program/library.

Based on DARPA's performance targets, each RCB can be labeled with a *dominator*. A dominator is the characteristic (execution time, file size, runtime memory usage or functionality) that contributes the most to the decrease in the availability score of the RCB. Determining the dominators for each RCB is a way to evaluate the ability of the architecture and algorithms described herein to efficiently rewrite a program/library and determine where to focus optimization efforts. For instance, if a majority of the RCBs with low availability scores are dominated by file size overhead, it is reasonable to focus optimization efforts on how the architecture and algorithms of the static binary rewriter affects the on-disk layout of a rewritten program/library.

[5]Note that the calculation of runtime memory usage done in the DECREE environment is not subject to the limitations discussed in Section 5.5.2.

DARPA released a set of 145 CBs for competitors to use for testing prior to the competition. The CGC dataset used for this evaluation (and the related CGC evaluations performed in subsequent experiments) contained 143 of those. Two of the CBs were omitted because they could not be rewritten appropriately. One of the two CBs contains self-check code that performs runtime validation that its code is identical to how it was originally compiled.[6] The second of the two CBs contains obfuscated function pointers for "almost all of the function calls" [116]. The "function pointers are obfuscated at compile time by adding a constant value to the pointers", a situation that makes it infeasible for the algorithms of the static binary rewriter's IR Construction phase to discover the addresses of those functions and handle them correctly.[7]

To evaluate the performance of the architecture and algorithms described in the preceding chapters, an RCB for each of the CBs in the CGC dataset was generated using the prototype implementation.[8] Figure 5.14 shows the availability scores for each of the 143 RCBs for the CBs in the CGC dataset. The average availability score for the entire dataset is approximately 83%.[9] Because the determination of whether a particular characteristic of the execution of an RCB is dominated and, therefore, lowers its availability score is based upon limits set by DARPA, the availability score itself is not good or bad. Each of the characteristics is already allowed a certain amount of overhead before it affects the availability score so the objective is for this value to be as close to 100% as possible.

Understanding which of the characteristics dominate the RCBs that negatively impact the overall availability score will help direct optimization efforts. The distribution of dominators for the 143 RCBs for the CBs is shown in Table 5.4. One of the primary goals set for the architecture and algorithms of the static binary rewriter is to create rewritten programs that have a very small amount of on-disk overhead (see Part I). The dominator results from the CGC data set indicate that the algorithms and architecture described so far accomplish that goal. The dominator results also indicate that the execution overhead of programs statically rewritten using the architecture and algorithms described so far are small. Where the dominator results direct attention for optimization is the memory overhead. More than 82% of the RCBs with less-than-perfect availability scores are dominated by memory overhead. The optimizations described in Part III are designed to address this shortcoming.

5.7 Dense Pinned Addresses and Sled Usage

As mentioned in Section 4.4.2, special techniques are required to handle situations where pinned addresses are too close together to use the link technique. Sleds effectively solve the problem but add runtime overhead.

[6] Programs/libraries that contain obfuscated code are explicitly out of scope for this work. See Chapter 3.
[7] Programs/libraries that contain obfuscated code are explicitly out of scope for this work. See Chapter 3.
[8] The only transformation used when generating these RCBs was the Null Transformation.
[9] All calculations of the average availability score throughout this dissertation are done using the arithmetic mean.

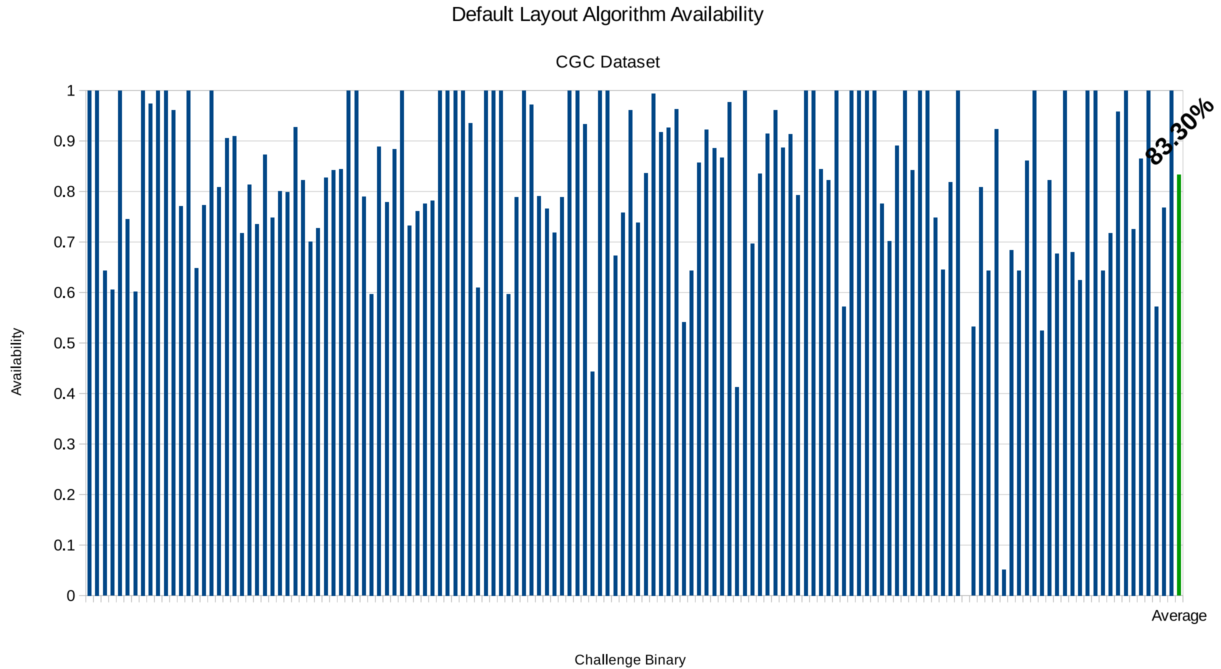

Figure 5.14: Availability scores for the RCBs in the CGC dataset when reassembled using the default algorithms of the Reassembly phase.

Category	Count
None	42
Performance	16
Filesize	1
Memory	83
Functionality	1

Table 5.4: Distribution of dominators for the RCBs in the CGC dataset.

Because of the added overhead caused by this solution, it is important to understand how often sleds are used.

For the entire set of test programs in the SPEC2006 benchmark suite, no sleds are required. Nor are sleds required for any of the programs in the set of coreutils. No sleds were required to successfully rewrite *glibc*.

5.8 Conclusion

This chapter described the results of experiments with statically rewriting programs and libraries to validate the thesis that creating a robust static binary rewriter is possible. The programs and libraries were actually transformed for experimental evaluation using Zipr, a prototype implementation of the algorithms and architecture described in the preceding three chapters. The results indicate that it is possible to statically

rewrite real-world programs and libraries like the complex computationally bounded applications in the SPEC benchmark suite, the performance-sensitive open source implementation of the C standard library, the widely deployed Apache webserver, a popular runtime virtual machine for Java and more than 140 applications developed for the DARPA CGC. Yet more evaluation is needed.

To understand whether the tool is useful to system and security designers and architects, applications for this technology must be explored. A description of three potential applications of the static binary rewriter are presented in Part IV to demonstrate its ability to transform SOUP to add security.

And, although the architecture and fundamental algorithms of the static binary rewriter presented in this section can be used as the basis for a functional tool, there exists an opportunity to refine that design so that statically rewritten programs are more efficient and meet the goals outlined in Part I. The optimizations are presented in Part III.

Chapter 6

Related Work

6.1 Introduction

While there are myriad tools that rewrite software to improve its security and reliability, there are only a few that operate on SOUP. Only those that meet the criteria and constraints described in Part I are described in this chapter. Rewriters that operate on software stripped of debugging information and metadata and whose source code is unavailable can be divided in two different classes: static and dynamic. Dynamic binary rewriters modify the program online and static rewriters modify the program offline.

Dynamic binary rewriting, also known as software dynamic translation (SDT), is a useful tool for system architects to modify legacy software to add state-of-the-art security mechanisms, rearrange instructions in a program to improve branch prediction and cache localization or retarget software from one platform to another. SDT requires a runtime engine to translate and execute the modified program. Although some critical software does operate in a context that allows additional runtime and performance overhead, the fact that a significant number of these safety-critical software systems operate on embedded platforms sensitive to strict performance requirements disqualifies SDT as a technology for modifying SOUP to add additional security and reliability in the context presented in Part I.

The goal of static rewriters is to give the system engineer all the functionality of SDT without requiring a runtime translation engine. Modern static binary rewriting systems can be used not only for instrumenting a binary but also for optimizing it. The rewritten program may be optimized for speed, security, reliability etc. Each of these optimization targets adds additional challenges for the design of the static binary rewriter.

Several static binary rewriters exist, each with its own strengths and weaknesses. Some of the more prominent binary rewriters are described in the remainder of this section. The description of each includes a

Year	Tool
1994	ATOM [212]
1997	Etch [183]
2001	Vulcan [69]
2005	DIABLO [225]
2010	SecondWrite [164]
2014	Work begins on the architecture and algorithms described herein.
2016	UROBOROS [234]
2017	Ramblr [232]
2017	Zipr [94]

Table 6.1: Timeline of introduction of related work.

discussion of the ways that it is inappropriate for the context described in Part I.

Table 6.1 presents a timeline of the publication of the work described in this chapter with respect to the beginning of the work described in this dissertation.

6.2 ATOM

Analysis Tools with OM (ATOM) is a very early static binary rewriter written by researchers at the Digital Equipment Western Research Laboratory in the mid-1990s [212]. ATOM provides its user with the tools to build a "wide range of customized program analysis tools" [212] for monitoring dynamic memory allocation or instruction profiling [73]. ATOM relies on OM [213] to perform the underlying analysis of the program to be rewritten. OM operates on the object code and not the final program which means that ATOM cannot be used to rewrite SOUP.

6.3 Etch

Etch is "a general purpose tool [from 1997] for rewriting arbitrary Win32/x86 binaries without requiring source code" [183] that allows developers the opportunity to instrument/modify a Windows program. "Instrumentation code can add, remove, or modify instructions, or add procedure calls at any point in the executable" [183]. The original motivation for Etch is to create a tool that rewrites programs "for both measurement and optimization" [183] and does not provide any information on the overhead of the rewriting technique itself. The system does not provide any high-level API for the transformation writer. He or she is responsible for writing a transformation that is accurate with respect to indirect memory accesses and indirect control flow. Because Etch is limited to Windows programs and requires knowledge of the control flow of the program under transformation to operate correctly, it is inapplicable in this context.

6.4 Vulcan

Vulcan is a combination static and dynamic binary program rewriting framework from Microsoft Research in 2001 [69]. Vulcan supports static and dynamic binary rewriting in both online and offline modes and can be used for software analysis and measurement, optimization, management and reliability. Vulcan allows its user to rewrite programs that are compiled into machine code, intermediate languages like MSIL or a combination thereof. The Vulcan authors argue that its ability to rewrite hybrid applications makes it well suited to modern developers who have to contend with applications that simultaneously target multiple architectures where "final optimization and code generation [is] performed at run time" [69]. In other words, Vulcan is designed for "research and development in the new Internet environment" [69]. Vulcan requires a program's metadata in order to rewrite so it is not applicable to SOUP.

6.5 DIABLO

DIABLO is a static rewriter first described in 2005 whose main goal is to support whole program optimization [225]. DIABLO is "reliable" [225] because it is a link-time rewriter that has access to relocation information that removes ambiguity about memory accesses and indirect targets. Whereas a compiler can only optimize within the scope of a single compilation unit (and cannot, for instance, optimize libraries used by the program), DIABLO uses knowledge of the entire program to perform optimizations. Although the goal of DIABLO is whole program optimization, the tool allows its user to perform arbitrary program modification. Because DIABLO requires metadata that is not retained after linking, it cannot be used to statically modify SOUP.

6.6 SecondWrite

SecondWrite is "an advanced binary rewriter that operates without need for debugging information or other [assistance]" [164]. The system is developed as a collaboration between the University of Maryland and Columbia University and was first described publicly in 2010. Their system is novel insofar as it represents new techniques for decompiling programs into a high-level IR and handling indirect targets.

SecondWrite disassembles a given program binary into LLVM[1] IR format and then passes control to the compiler to perform standard optimizations and generate a rewritten binary. In this respect, the disassembly phase of SecondWrite is like those of Ramblr (Section 6.7) and UROBOROS (Section 6.8) that attempt to

[1] LLVM was originally an acronym for low-level virtual machine but no longer stands for anything[125]. However, it is still written typographically in all capital letters[219].

create a representation of the program that can be given directly to one of the tools in a compiler pipeline that can, in turn, compile that into an executable program or linkable library.

SecondWrite splits the original program stack into individual frames, splits those frames into individual variables and, finally, converts constants and variable memory accesses into symbols. Based on that analysis, SecondWrite constructs an IR from the original program that can be analyzed by LLVM. SecondWrite applies LLVM's built-in optimizations to this IR and uses LLVM's code generation algorithms to reconstruct the modified program.

Their rewriting process resembles that of UROBOROS and Ramblr. Whereas UROBOROS and Ramblr generate a type of assembly code that can be fed into an off-the-shelf assembler, SecondWrite generates a type of code that can be fed into an off-the-shelf compiler.

Unfortunately, SecondWrite's analysis is not guaranteed to be fully accurate. Depending upon the program, their analysis may not recognize all IBTs. To handle these cases, the rewritten binary contains a copy of the original binary program. In this way, SecondWrite is like a replica-based rewriter (Section 6.8, [233]). which means the rewritten version is at least 100% larger than the original. In fact, for the benchmarks in the SPEC 2006 suite, SecondWrite's filesize overhead is 6.3x [205]. Given the disk size constraints of many of the embedded hardware platforms that run critical systems, this overhead is not acceptable.

6.7 Ramblr

Ramblr is a static binary rewriter developed by researchers at the University of California, Santa Barbara in 2017 [232]. Ramblr extends the research results of UROBOROS (see Section 6.8 for more information) [234]. Ramblr's overall process for rewriting an original program is straightforward: disassemble the original program, analyze the disassembly output and reassemble a statically rewritten binary.

Ramblr's analysis phase creates reassembleable disassembly [233] from the disassembler's output which is fed into a traditional, off-the-shelf assembler that performs the reassembly and generates the statically rewritten binary program. Ramblr's use of novel static analysis techniques in the analysis phase to create this reassembleable disassembly from the output of a standard assembler is what advances the state of the art.

Reassembleable disassembly is different from the disassembly generated by traditional disassemblers like objdump or IDA Pro. The output from traditional disassemblers cannot be given to an assembler to produce a program. In reassembleable disassembly, references to memory locations (whether those locations held code or data) are done symbolically. Assemblers rely on symbolized assembly code as input so that they can place the targets of those references freely throughout the output program. If the assembly code given to

	Assumption
1	Pointers are all machine word aligned.
2	The reassembled program will have data segments in the same place as the original program.
3	Pointers to code embedded in the original program's data segments are only either pointers to functions or pointers into jump tables.

Table 6.2: UROBOROS's assumptions that Ramblr authors believe do not hold for a non-trivial number of real-world programs.

the assembler used absolute addresses instead of symbolic addresses, code or data movement done by the assembler would affect the program's functionality.

A disassembled program contains two types of *immediates*, constant values embedded in the object code and data, that are really symbols. First, the disassembly output may contain instructions that use the immediates as symbol addresses. Second, the disassembled program may have immediates embedded in its data segments that are really symbol addresses used by indirect program control transfers.

A novel method of static analysis is the key to Ramblr's ability to identify the immediate values in the program's instructions that really are symbols. As mentioned previously, the development of these static analysis algorithms is based on the author's analysis of the assumptions and shortcomings of UROBOROS (see Section 6.8).

UROBOROS and traditional rewriters identify immediates that are potentially the address of symbols used in indirect program control transfers through a simple, linear scan of the disassembled program's data segments. UROBOROS' recognition algorithms make several assumptions about the structure of input programs that are inaccurate and result in incorrect symbolization. In particular, UROBOROS makes the three assumptions listed in Table 6.2 that do not hold for a non-trivial number of programs.

The first assumption is violated by the presence of packed data structures whose individual elements are not machine-word aligned. The second assumption makes it impossible to move and/or update the size and contents of data segments, something that an assembler must be free to do if it is to generate a reassembled binary from symbolized assembly code.

The UROBOROS authors relied on the third assumption to filter the potential code pointers in the original program's data segments. By their reasoning, if a potential pointer in the data segment does not point to a jump table or the beginning of an identified function, then the UROBOROS reassembleable disassembler assumes that it is not a symbolizable immediate. The problem is with the underlying assumption that identification of the start addresses of every function in every program is possible. The fact that every function's start address cannot be properly identified means that the UROBOROS system would incorrectly skip symbolization of some immediates

The Ramblr researchers identify several other reasons why traditional techniques for generating reassembleable disassembly may fail. First, compiler optimizations trigger false negatives. There are situations where compiler optimizations will add/subtract a value to/from an immediate before using it as a pointer at runtime. These constants are skipped by the symbolizer because the immediate base value does not fall within a memory region that can ever be the target of a code or data pointer. However, after modification up/down before and before being dereferenced, the immediate ends up pointing to an address that does fall within a memory region that can contain a pointer to code. Immediates that meet these criteria should be symbolized. Second, programs may implement abnormal behavior to manipulate the value of pointers. In particular, programs often employ schemes to encrypt pointers. Even though their true use is obscured by other operations, these values should be symbolized. Third, and finally, there are collisions where an immediate that is not actually a symbol (e.g., the value is being used for a mathematical operation) but its value falls within a region of memory that *could* be the target of a code or data pointer. These immediates should not be symbolized.

To overcome the invalid assumptions and shortcomings of UROBOROS' and other traditional static rewriters' techniques for identifying values to be symbolized, Ramblr performs static analysis on the traditional disassembler's output.

Ramblr first performs a scan of the program's disassembled instructions and its data section(s) to identify prospective immediates that might be symbol addresses. This prefilter step trims out all the values that cannot point to valid areas of memory (either code or data) but it is not strict; it allows immediates that *approximately* point to valid addresses. Without the slight expansion of valid target regions, Ramblr could not handle the compiler optimizations described above where the immediate addresses an invalid area of memory but, because of subsequent modification before dereference, actually point to a symbol.

For each of the prospective immediates that might be symbol addresses, Ramblr statically analyzes the basic blocks preceding its usage as a pointer. The result of the analysis is a set of potential values that pointers may take. Ramblr assumes that the targets of those potential values are addresses of labels and should be symbolized.

Ramblr uses static analysis in other ways, too. Their static analysis yields information that Ramblr uses to perform data identification and type analysis. Ramblr is able to analyze many primitive types (e.g., integers, pointers, strings, jump tables and arrays). By identifying immediates as integers, floating point numbers, etc, the analysis overcomes misclassifications when those constants, by chance, look like symbolizable values.

The static analysis does add a certain amount of runtime overhead to the operation of Ramblr. In cases where this overhead is unacceptable, Ramblr can employ a fast data type analysis. In this fast analysis case, Ramblr looks for *obvious* uses of data that indicate the program is using the value as an integer, floating

point number, etc. The fast analysis also detects consecutive machine-width values that point to pre-defined regions of memory which are considered to be an array of pointers. They also look for singly- and doubly-null terminated sequences of ASCII and Unicode characters to identify strings.

The authors evaluated Ramblr extensively on its own and in comparison with UROBOROS using programs from the CGC and coreutils. Overall, the results show that a Ramblr rewritten program has zero performance overhead and may actually decrease the on-disk size.

The authors used GCC to build the programs of the coreutils package submitted to Ramblr and UROBOROS for rewriting. They experiment on versions of the tools compiled and optimized at levels `0`, `1`, `2`, `3`, `s`, and `fast`. Ramblr maintains the functionality of all the programs in the package no matter the level of optimization. Ramblr does fail to maintain functionality in one case when it is invoked to rewrite the coreutils in fast analysis mode. UROBOROS, on the other hand, breaks functionality in more than 20% of the cases when optimization is disabled and breaks functionality in more than 50% of the cases when even minimal optimization (`1`) is enabled.

The Ramblr designers also use the binaries provided to the competitors of CGC for further evaluation. These programs are compiled with Clang and optimized at levels `0`, `1`, `2`, `3`, `s`, and `fast`. Ramblr never breaks functionality when the programs are compiled without optimization. Ramblr maintains the functionality of more than 98% of the programs even when they are compiled with the most aggressive optimizations.

Because the Ramblr authors relied heavily on the (invalid) assumptions of UROBOROS, they attempted to recreate UROBOROS' results for comparison with their tool. In order to recreate the experiments, the Ramblr authors had to work through several inconsistencies that compromised their ability to make comparisons between the two tools. For instance, the Ramblr authors had to disable several of UROBOROS' assumptions (Table 6.2), use an older version of the compiler for test programs, limit comparisons to 32-bit versions of the test programs. Further, the Ramblr authors could not compare the tools' symbolization correctness because the UROBOROS tool does not produce data that contains this information. See [232] for full details of the Ramblr's authors comparison with UROBOROS.

6.8 UROBOROS

UROBOROS is a static binary rewriter developed by researchers at the Pennsylvania State University College of Information Sciences and Technology in 2016 [234]. UROBOROS allows the user to perform arbitrary transformations on binary programs without access to the program's debugging information or relocation symbols. Like most static binary rewriters, UROBOROS works in three phases: the analysis phase, the transformation phase and the rewriting phase.

UROBOROS pioneered the use of reassembleable disassembly in static binary rewriters. To generate their reassembleable disassembly, they use algorithms their authors previously developed and documented [233]. By using reassembleable disassembly as the currency of the rewriter, the transformations can be implemented without the additional overhead that comes with patch- and replica-based rewriting.

In patch-based rewriting each instrumented instruction is replaced with a `jmp` to the transformed version of the instruction. At the end of the transformed (sequence of) instruction(s), the program `jmp`s back to the instruction that succeeded the original instruction. There are several problems with patch-based rewriting. First, it adds the overhead of two program control transfers for each rewritten sequence of instructions. Second, there are problems with instrumenting instructions that are smaller than the 5-byte `jmp` that transfers program control to the transformed sequence of instructions. Third, and finally, the transformed instructions may not be relocatable and may have different semantics when they are moved to a different location in the rewritten program.

A program rewritten using replica-based rewriting contains two copies of the rewritten program. One copy, the transformed copy, contains the version of the original program rewritten according to the User-specified transformations. The transformed version of the original program is rewritten with best effort. This means that execution will *likely* stay under the control of the transformed code. Because the transformed version of the program is not perfect, execution may escape the rewritten version. When execution escapes the transformed version, it will find its way to the second copy, a specially modified copy of the original program. This copy of the original code contains special modifications at every one of the IBTs. The modified copy has jumps at those IBTs that are used to send the program execution back to the transformed version of the code. Like patch-based rewriting, this type of rewriting incurs overhead. First, the on-disk size of the rewritten program is twice as large as the original. Second, the more often program execution escapes the transformed version of the program, the more often additional program control transfers are necessary to return execution to the transformed version.

The authors of UROBOROS also argue that programs rewritten using patch- and replica-based techniques produce binaries that are unsuitable for further analysis (i.e., traditional disassemblers cannot meaningfully operate on them). In their opinion, this makes the rewriting tool unsuitable for use in a pipeline of tools designed to operate on binary programs.

As long as the reassembleable disassembly generation is accurate, UROBOROS can transform and rewrite binary programs without incurring those penalties. The process of generating reassemble disassembly begins with a basic disassembly and UROBOROS uses the methods outlined by the authors of BinCFI (See Section 12.3.1) to perform this prerequisite step [251]. “To make the disassembled output relocatable, the main challenge is to identify memory addresses from all the data found in the disassembled output.” “An

in-depth study shows that with all the methods applied, almost no false positive or negative is found on broad sets of real world binaries."

The evaluation of UROBOROS measures the performance of the implementation of their reassembleable disassembly, transformation and reassembly techniques. At the time of writing, the implementation supported ELF binaries on 32- and 64-bit platforms but did not support C++ programs. They performed two sets of tests.

In the first, they measured the performance overhead of untransformed rewritten programs. This is equivalent to the Null Transformation tests of Zipr reported in Chapter 5. The authors reported "at most 1% execution slowdown and size expansion, when evaluating a broad set of real world programs."

In the second, they measured the speed of a UROBOROS-transformed program against a DynInst-transformed program using the SPEC2006 benchmark. Because UROBOROS did not support C++ programs at the time of writing, their tests were limited to the programs from the SPEC2006 benchmark suite implemented in the C programming language. The programs from the benchmark were transformed using UROBOROS and DynInst to count every basic block and every function executed at runtime. They chose to compare their performance against DynInst because it is "widely used." "For both tests, DynInst increases binary sizes to more than twice of the original (131.2% increase for basic block counting and 119.1% for function counting), while UROBOROS only brings in less than 40% of size expansion for basic block counting and 2.0% for function counting." "... [F]or function level instrumentation, the average execution slowdown of DynInst (2.94%) is as good as UROBOROS (2.93%). For the basic block counting task, binaries instrumented by UROBOROS is slower than DynInst (93.38% compared to 76.56%)."

6.9 Conclusion

Work on the design and architecture of the static binary rewriter described in this dissertation began in late 2014 after the publication and description of SecondWrite but before the research of UROBOROS and Ramblr. Based on a survey of related work at that time, it was an open question whether a static binary rewriter could operate on SOUP and reliably generate a rewritten program/library.

Furthermore, at that time it was an open question whether a static binary rewriter could rewrite SOUP without preserving a copy of the original code alongside the modified version in the statically rewritten program. Nor had it been determined that it was possible to build a static binary rewriter for SOUP that could build an output program/library that executes with a reasonable amount of runtime overhead. The work presented herein proved it was possible to reliably generate statically rewritten programs that do not preserve a copy of the original program and execute with an acceptable amount of runtime overhead.

Part III

Optimizations

As mentioned in the description of the evaluation of the fundamental design, architecture and algorithms, there are myriad reasons why the algorithms of the IR Construction and Reassembly phases produce statically rewritten programs that are inefficient (in terms of performance, runtime memory usage or on-disk size). Because of the criteria for the static binary rewriter described in Part I, producing the most efficient statically rewritten binaries is vital. This section describes and evaluates three different algorithms to augment the Reassembly phase that improves the efficiency of the statically rewritten binary programs.

Like Zipr was created for the purposes of evaluating the underlying design, architecture and algorithms of the static binary rewriter, each of these optimizations is actually implemented for the purpose of evaluation. One of the features of Zipr is a plugin system and API that allows customizations of the default algorithms of the Reassembly phase. A plugin implements a function named `AddressDollop` that is invoked by the Reassembly phase and allows the plugin to override the Reassembly phase's default behavior. The pseudo-code for the optimizations described in this part of the dissertation will all make reference to that function.

Chapter 7

Locality Layout

7.1 Introduction

The Locality Layout algorithm is an improvement to the basic algorithms of the Reassembly phase (described in Section 4.4). The goal of the Locality Layout algorithm is to improve both runtime performance with respect to CPU utilization and memory overhead of statically rewritten programs/libraries. The main focus, however, is on reducing the memory overhead.

In most major OSes, a program's code is paged into memory on-demand when the instructions on that page are executed for the first time and is part of the OS's lazy memory initialization scheme. The benefit for the user is that a large program with many instructions will only load the parts of the program into memory that are actually executed. If a program is statically linked to many extraneous libraries, that code will not take up memory at runtime because it is never actually executed and, therefore, is never loaded. The OS is limited by the physical memory of the host in the amount of program code that can exist simultaneously in memory. Minimizing the size of each program's resident memory at runtime supports the OS's objective of running as many processes as possible in parallel before having to swap out the memory allocated to idle programs.

At a program's startup, none of its code is in memory and the PC is transferred to the program's entry point, e. Because the code is not in memory, when the CPU attempts to transfer control to the instruction at e, the CPU emits a page fault. The OS handles this fault by invoking a page fault handler that loads the page of code around e from disk into memory. The process is commonly referred to as *paging in* from disk. As the name suggests, the page fault handler in the OS loads an entire *page* of memory when it handles a fault. By reading in an entire page rather than just the single instruction at e, when the CPU executes the

next instruction (assuming the instruction at e is not a `jmp`, `call`, etc.) that code is already in memory. This insight had a tremendous impact on the design of the Locality Layout algorithm.

As a program is built and optimized, the compiler does its best to keep instructions that execute as a unit together in memory. This locality is one of the ways that the compiler builds spatial locality into a program in order to improve performance of instruction fetching. When a segment of memory is paged in to execute a particular instruction i at the start of basic block b, there is a benefit to having the subsequent instructions in b execute from the same page in memory. This compiler behavior maximizes cache performance and minimizes the amount of resident memory used by the program as it executes. There are several traditional ways that a compiler builds a program in order to accomplish this objective – see Section 8.6 for an explanation of those techniques.

In order to improve the memory locality of a statically rewritten program, the Locality Layout algorithm attempts to keep interconnected dollops nearby in memory. Ideally, a unit of interconnected dollops would exist entirely within the same page to minimize the number of page faults.

7.2 The Problem

The basic algorithms of the Reassembly phase does not take locality into account. By not considering locality, the statically rewritten binary program exhibits very poor runtime memory overhead. In addition to increasing the runtime memory overhead, the time that the CPU spends handling the extra page faults in the inefficient statically rewritten binary program decreases the program's execution speed.

Understanding the reason for the poor performance depends upon an integrated understanding of the OS's runtime memory management algorithms (outlined above) and the static rewriter's use of pinned addresses. Recall pinned addresses, first discussed in Section 4.2.2. A pinned address is a location in the original program that may be the target of an indirect program control transfer. In the statically rewritten program, a link is placed at the pinned address to associate it with the target code, the (possibly modified by a user transformation) code from the original program at the pinned address. The basic algorithm in the Reassembly phase places the target code arbitrarily in the rewritten version of the program. Oftentimes the target dollop is placed on a different page than the pinned address that links to it.

To illustrate the problem, consider the program p and its statically rewritten version p'. For the purposes of this example, assume that

1. p's entry point is a pinned address and
2. the target code from p's entry point is placed on a separate page from the pinned entry point in p'.

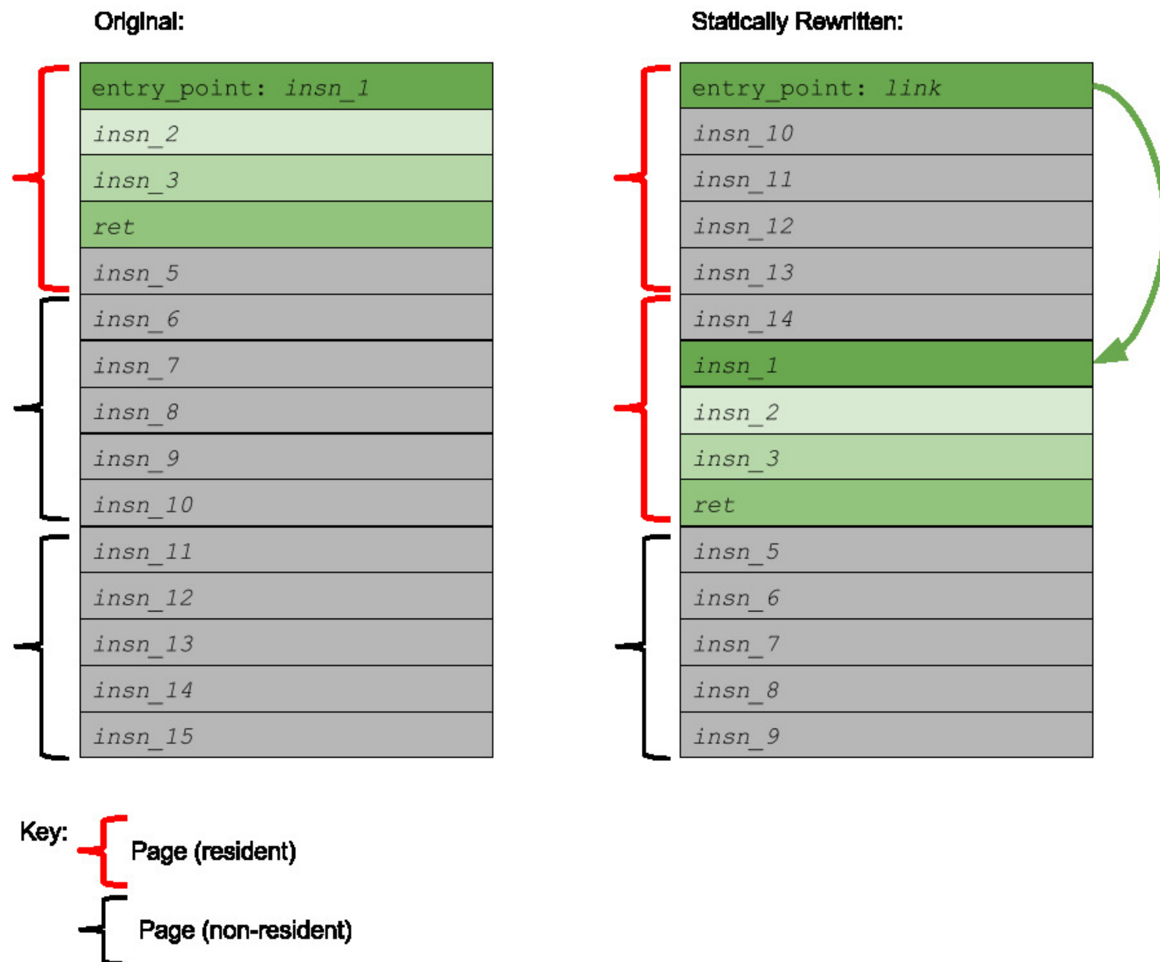

Figure 7.1: How the basic algorithms of the Reassembly phase cause excessive runtime memory usage because it does not respect program locality.

See Figure 7.1 for the visual representation of the following description.

When p starts,

1. there is no code present in memory
2. the PC is redirected to the program's entry point
3. a page fault occurs
4. the program loads that page into memory
5. the code at the entry point executes.

Before the first instruction of p executes, there has been one page fault, the OS's page fault handler has been invoked once, and p's RSS is a single page.

When p' starts,

1. there is no code present in memory
2. the PC is redirected to the program's entry point
3. a page fault occurs
4. the OS loads that page into memory
5. the instruction linking to the target code is executed.
6. the PC is redirected to the target dollop
7. because the target dollop is on a separate page (an assumption for this example), another page fault occurs
8. the OS loads that page into memory
9. the target code executes.

Before the first useful instruction of p' executes, the OS's page fault handler has been invoked twice and its RSS is two pages.

As the program continues to execute, the problem compounds.

7.3 The Solution

To combat this problem, the Locality Layout algorithm changes the static binary rewriter's default reassembly algorithm in the following way:

1. If the dollop to be placed is the target of a link at a pinned address, the dollop is placed on the same page as the pinned address.
2. If the dollop to be placed is the target of a link from a previously placed dollop, the dollop is placed on the same page as the targeting dollop.
3. If space is not available on the desired page, the dollop to be placed is placed on the infinite page (Section 4.4.1).

Pseudo code for the Locality Layout algorithm is presented in Algorithm 4. The AddressDollop function is invoked with two parameters: the dollop to be placed and the address of the link to that dollop in the address space of the statically rewritten binary. For a dollop that is the target of a link at a pinned address, that value is the pinned address. For a dollop that is the target of a link from another dollop, that value is the address of that link. Function PageOf is a simple function that returns the page address for a specific memory location. Function IsValidAddress checks the return value from GetSpaceOnPage to determine whether GetSpaceOnPage succeeded in finding an acceptable range. Function GetSpaceOnPage is provided by the Reassembly phase of the static binary rewriter and finds, if possible, available memory from

a page in the address space of the statically rewritten program/library. The function takes three parameters: the page where memory is to be reserved, the minimum amount of available space for the request to be considered successful and an out parameter that is assigned the first address of the reserved memory. Finally, Function SizeOfDollop, also provided by the Reassembly phase of the static binary rewriter, calculates the size, in bytes, of the dollop.

GetSpaceOnPage runs in constant time. The function scans the addresses of the fixed-size page passed as an input parameter and looks for an appropriately sized range of free memory. Each invocation of AddressDollop invokes GetSpaceOnPage at most twice. Assuming that there are D dollops in the program/library being statically rewritten, AddressDollop is invoked D times. Therefore, the running time of the Locality Layout algorithm is $O(D)$.

Algorithm 4 Override the default behavior of the Reassembly phase to place a dollop according to the Locality Layout algorithm.

```
1: function AddressDollop(Dollop tp, Address source)
2:     place ← 0
3:     if not IsValidAddress(GetSpaceOnPage(PageOf(source), SizeOf(tp), source)) then
4:         GetSpaceOnPage(InfinitePage, SizeOfDollop(tp), source )
5:     end if
6:     return place
7: end function
```

Figure 7.2 is a visual representation of the Locality Layout algorithm in action as it reassembles a program. When p'', the statically rewritten binary generated with the Locality Layout algorithm, starts,

1. there is no code present in memory
2. the PC is redirected to the program's entry point
3. a page fault occurs
4. the OS loads that page into memory
5. the instruction linking to the target code is executed.
6. the PC is redirected to the target dollop
7. because the target dollop is on the same page, the target code executes.

Before the first actual instruction of p'' executes, the OS's page fault handler has been invoked once and the program's RSS is one page. At this point in the execution of p'', the program has generated the same number of page faults and has the same RSS as the original program p.

There are two reasons why the Locality Layout algorithm, though seemingly universally applicable, cannot be used. Both are related to size constraints in the statically rewritten binary program. First, there may not be room for the target dollop on the same page as its link. A page may be full for two reasons. Recall the

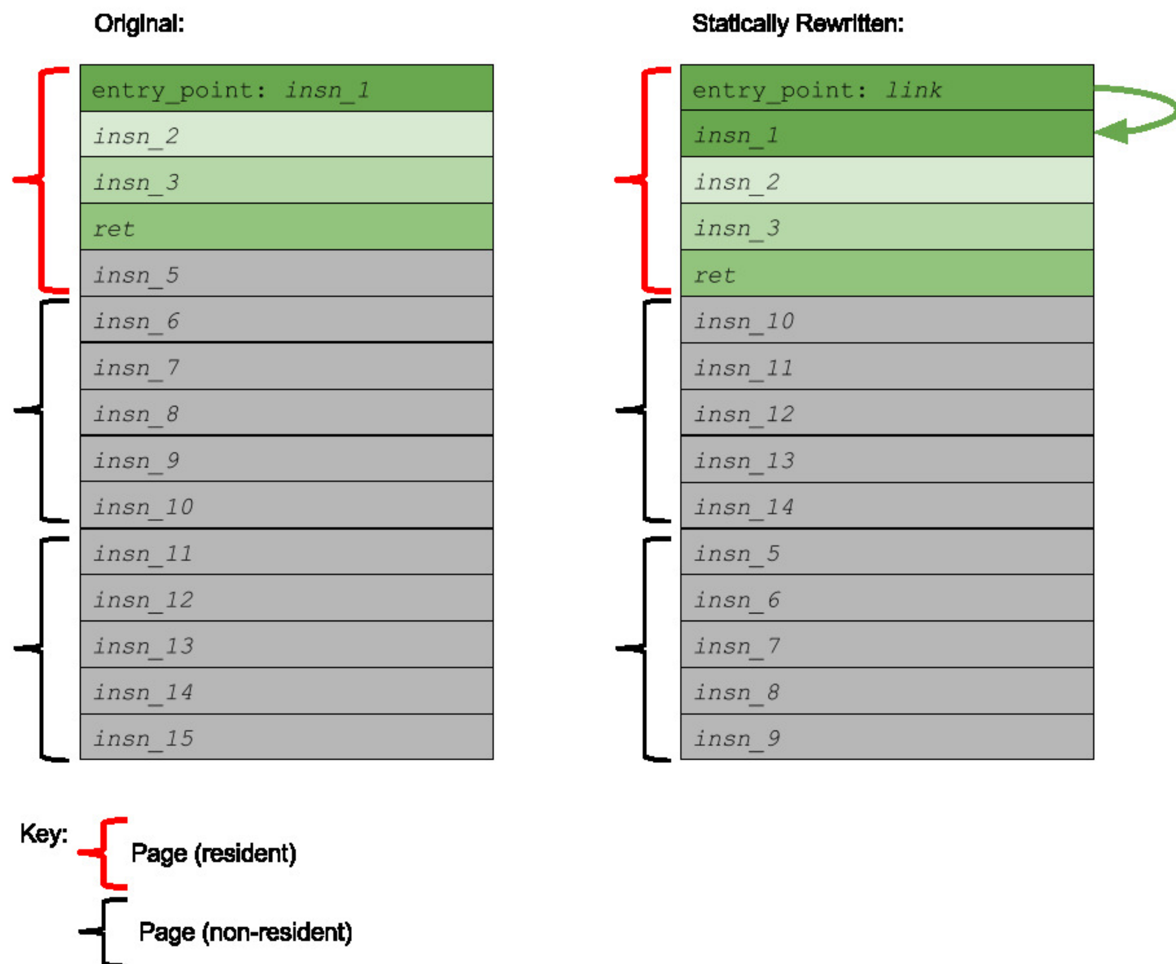

Figure 7.2: An example of the Locality Layout algorithm reassembling a program to take advantage of program locality. Compare the reassembly with the reassembly in Figure 7.1

basic algorithms of the Reassembly phase already discussed. The statically rewritten program p' or p'' is necessarily larger than p. This increase in size was discussed in Section 4.4.6 and shown in Figure 4.6. A page of the statically rewritten binary program that holds code between address a and address b effectively shrinks by the size of the instruction used to implement the link – either 2 or 5 bytes on the x86 platform – for every pinned address in that range. This problem is compounded by the fact that every pinned address essentially fragments the available memory and makes it increasingly unlikely that there will be target code dollops that perfectly and completely fill those fragments. Figures 7.3 illustrates the problem of the link implementation increasing the memory usage of a statically rewritten program by a page. Figure 7.4 illustrates the problem of a statically rewritten binary program having so many pinned addresses that links and their target dollops cannot fit on the same page. The instructions highlighted in red `should` both be at the same address. The

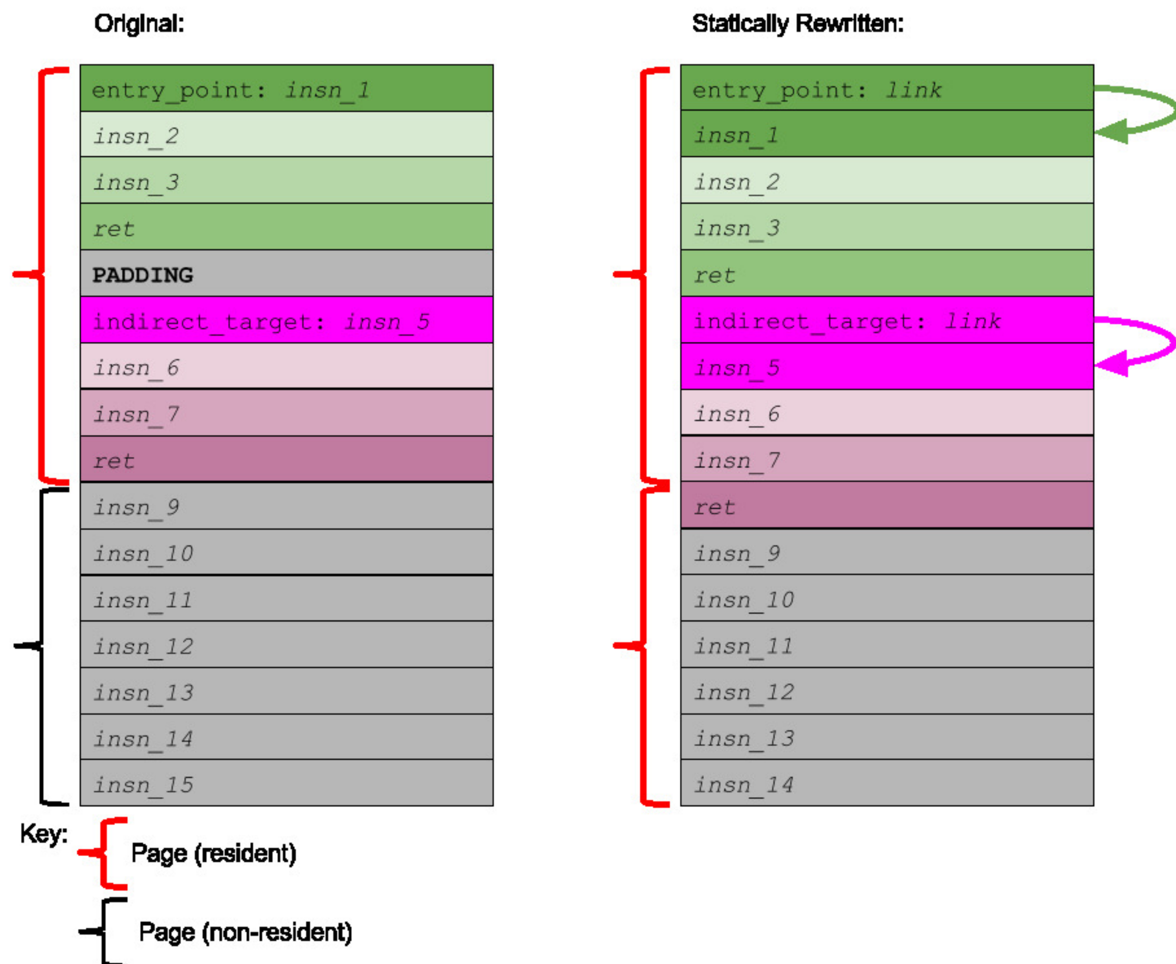

Figure 7.3: An example of how the size and density of links on a single page affects whether links and target dollops can fit on the same page.

target dollop containing instructions `insn_1`, `insn_2`, `insn_3`, and `insn_4`, will have to yield and be placed on another page.

Second, a target dollop may not fit on the same page as its link because a user transform adds additional instructions to the target dollop. See Figure 7.5. The original target dollop of the link at the program's entry point contained four instructions and fit on a single page. A user transformation added three new instructions to that dollop. The dollop placed in the statically rewritten binary is now bigger than a single page and must extend onto a second page. For programs that are bigger than a single page where a) at least some of those pages are less than completely filled with instructions from dollops and b) those pages are already resident in most executions, it may be possible to split the extended dollop and place the pieces on the not-completely-filled, usually resident pages. Splitting dollops will maintain parity of the maximum RSS

Figure 7.4: An example of how the size and density of links on the same page limits the ability to place dollops in the gaps between pinned addresses.

in the common case. This extension was considered for the Locality Layout algorithm but not implemented. It is implemented in the other optimization algorithms described in Part III.

The first of these constraints on the applicability of the Locality Layout algorithm – the space wasted implementing the links means that target dollops no longer fit on the same – can be mitigated, however. The solution constitutes an extension to the basic Locality Layout algorithm known as Overwriting Pinned Addresses. The extension does what the name implies: where possible, the target dollop is placed at the pinned address in the statically rewritten program which removes the requirement for using a link at all.

Assume that pinned address a points to target dollop d and that d consists of instructions i_1 through i_n. Without the Overwriting Pinned Addresses extension, the Locality Layout algorithm would place d's instructions i_1 through i_n on the same page as a. The link at a would target i_1 in d. The Overwriting Pinned

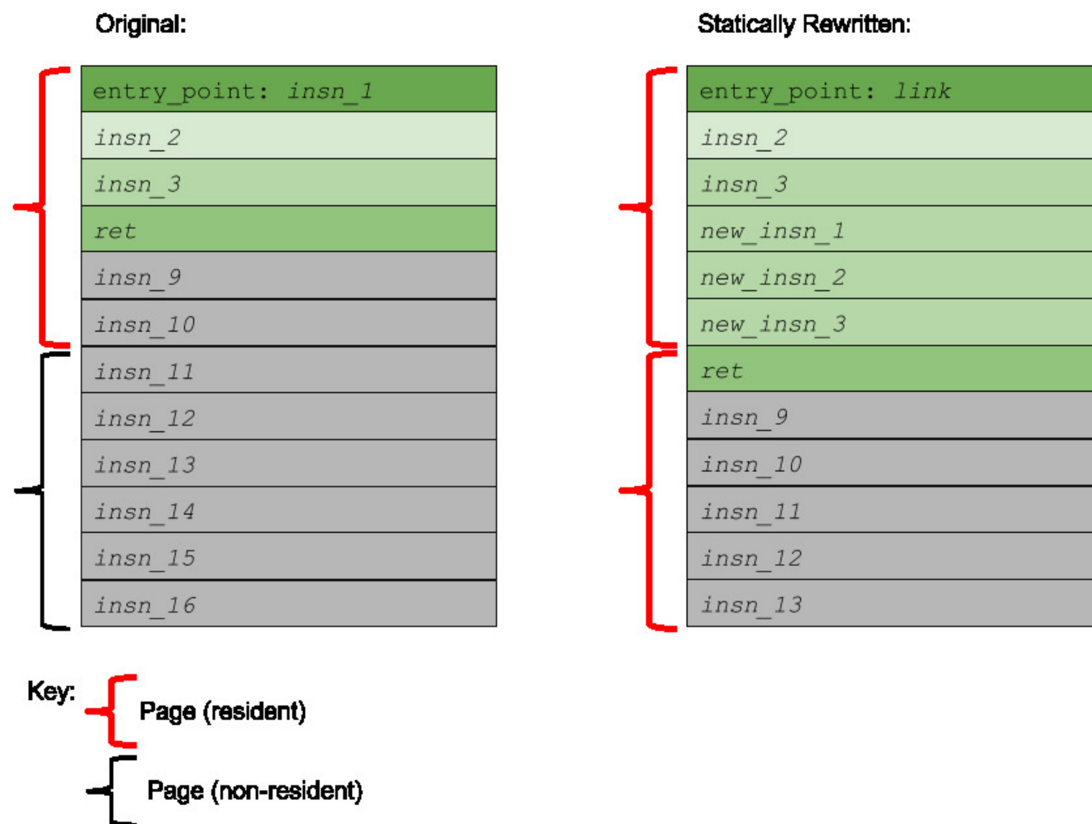

Figure 7.5: An example of how a user transformation to a dollop may make it so that a link and its target dollop cannot fit on the same page.

Addresses extension places d at a so that instruction i_1 is at a. When the PC is redirected to pinned address a, the code at the target dollop begins executing immediately without going through the link. There are three benefits. First, there is no wasted space for the implementation of the link – again, two or five bytes for the x86 architecture. Second, there is one fewer IB in the CPU's branch predictor. Third, and finally, there is one less branch that the CPU must take.

Comparing the statically rewritten binary in Figure 7.6 and the statically rewritten binary in Figure 7.3, the statically rewritten binary using the Overwriting Pinned Addresses extension reduces the RSS and decreases the number of page faults that occur when executing code in the two dollops shown in the figures.

Although this extension seems obvious and universally applicable, like the Locality Layout algorithm itself, it cannot always be employed. It is inapplicable in cases where the target dollop is extended by a user transformation and, because of its bigger size, does not fit in open space around its link address in the statically rewritten binary. See Figure 7.7 for visual representation of this scenario. Notice, particularly, the dollop placement highlighted in red. The final instruction of the target dollop at the program's entry overlaps

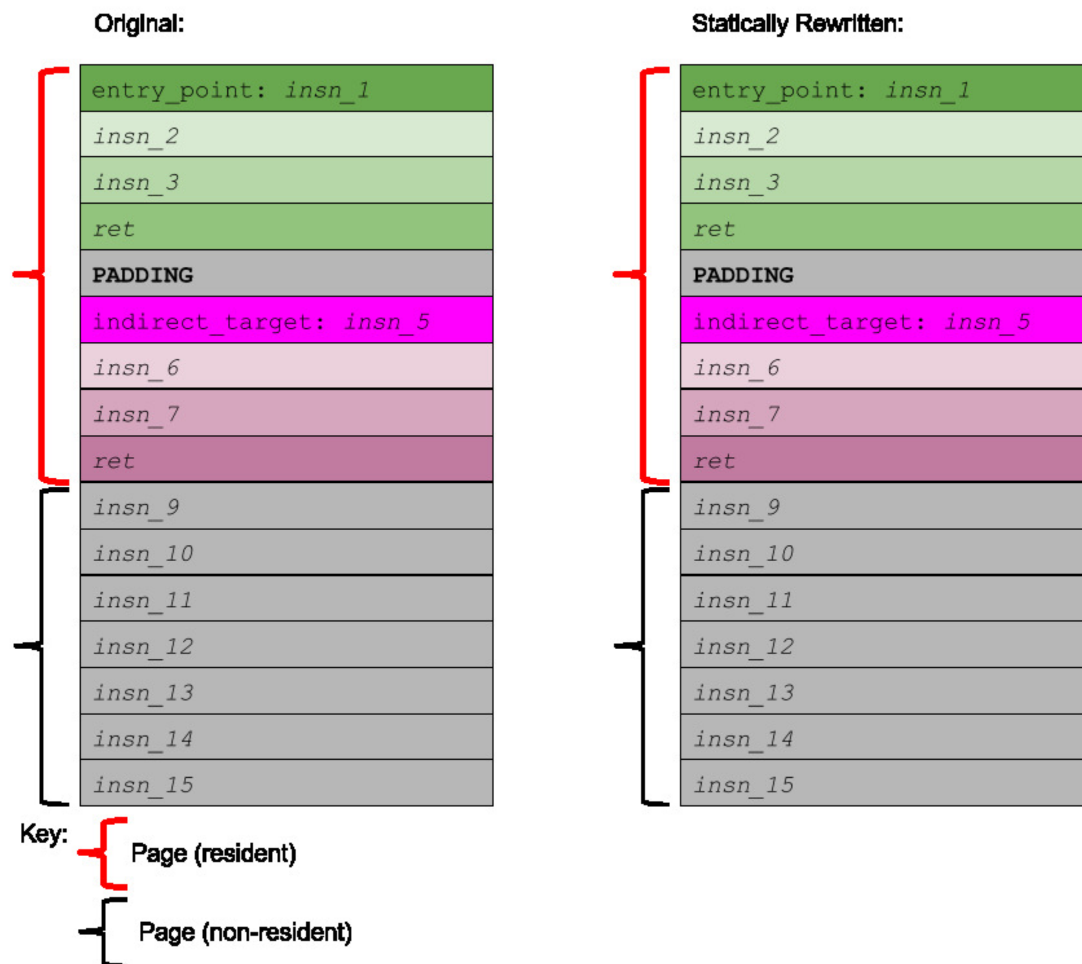

Figure 7.6: An example of how the Overwriting Pinned Addresses extension results in a smaller maximum RSS and fewer page faults.

pinned address *b*. This constraint on the extension's applicability parallels the second of the two reasons why the Locality Layout algorithm itself cannot be employed.

For historical reasons, this extension is actually implemented along with the Profile Layout algorithm (see Chapter 8). And, although the target dollop cannot always be placed in open space around its link address, when it is applicable, the extension always improves program behavior. For this reason, it is the default behavior of the Profile Layout algorithm.

Finally, even when the Locality Layout algorithm is available, its choice of pages for dollops may not always be ideal. This is especially true when the algorithm is placing dollops that are the targets of other dollops. Recall from the description of the default algorithms of the Reassembly phase that dollops are actualized on a first-come-first-served basis depending upon the order in which dollops are placed that target

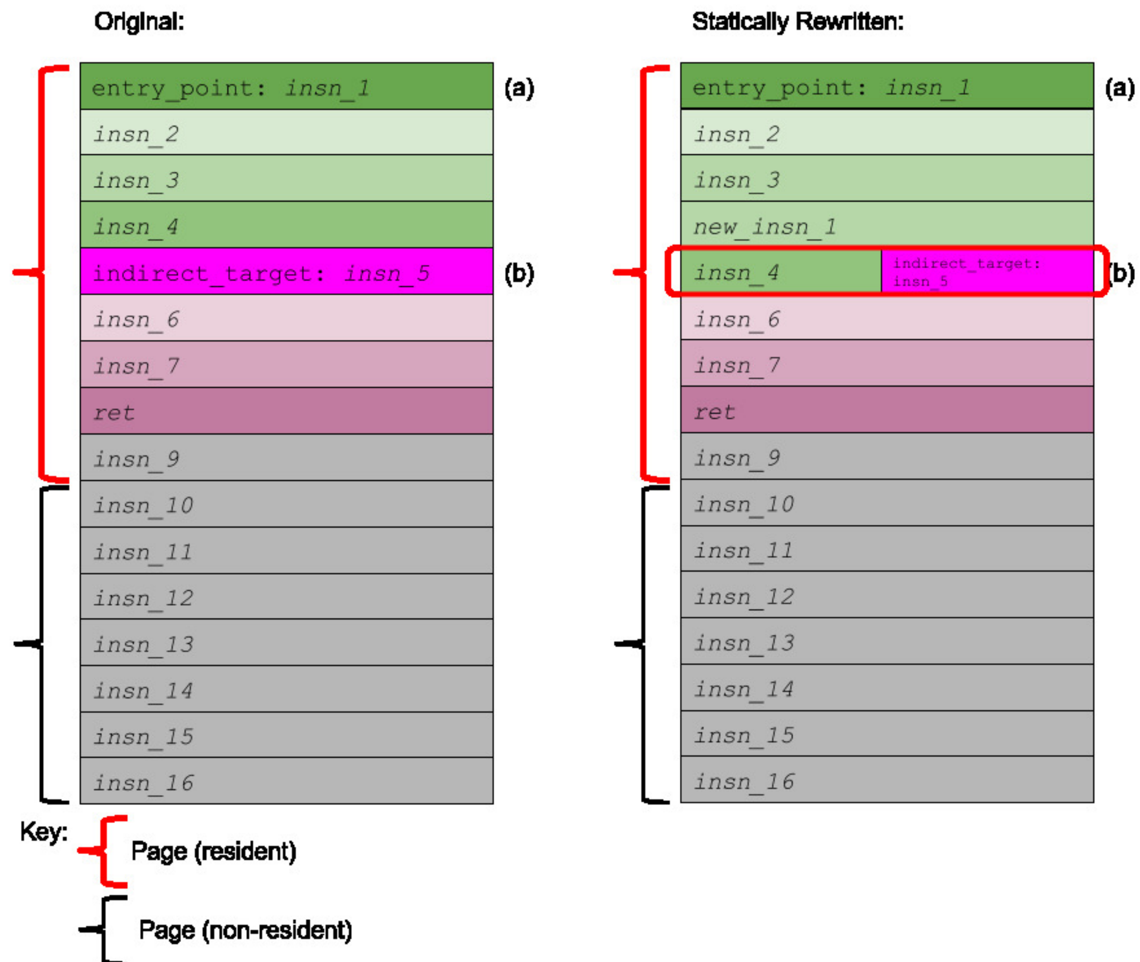

Figure 7.7: An example of how a user transformation that extends a dollop makes it impossible to apply the Overwriting Pinned Addresses extension.

that dollop (see Section 4.4). Dollops may be the target of more than one dollop. For example, assume that Dollops a and b both target Dollop c. If Dollop a is executed far more commonly than Dollop b but Dollop b is actualized first, the reassembled program will jump between different pages when Dollop b passes control to Dollop c. The transfer between pages will result in unnecessary page faults that increase overhead. Subsequent optimizations address this shortcoming.

7.4 Evaluation

The applications of the SPEC benchmark suite and the CBs of the CGC dataset were evaluated in order to assess whether the Locality Layout algorithm achieves the goal of lowering the memory overhead of

statically rewritten programs and, therefore, improves the overall performance of those programs. The detailed assessment and description of the experiments is presented in Appendix A. This section discusses the overall impact of the results on the path toward developing additional optimizations to the architecture and algorithms of the static binary rewriter.

7.4.1 SPEC

The SPEC benchmark suite provides a way to evaluate the performance impact of the Locality Layout algorithm on programs that are computationally bound.

The results for the memory usage of the applications of the SPEC benchmark suite reassembled using the Locality Layout algorithm compared to the memory usage of the applications reassembled using the default algorithms of the Reassembly phase were mixed. On Host A there was an improvement of just under a fifth of a percent and on Host B there was actually an impairment of less than a quarter of a percent.

The results for the page fault overhead of the applications of the SPEC benchmark suite reassembled using the Locality Layout algorithm compared to the memory usage of the applications reassembled using the default algorithms of the Reassembly phase were positive. On Hosts A and B there was an improvement of 0.85% and 0.78%, respectively.

The results for instruction cache miss overhead of the applications of the SPEC benchmark suite reassembled using the Locality Layout algorithm compared to the memory usage of the applications reassembled using the default algorithms of the Reassembly phase were very positive. On Hosts A and B there were improvements of more than 17%.

The improvements and impairments to these characteristics of the execution of the programs of the SPEC benchmark suite contribute to the overall performance. Results indicate that performance improves for the applications of the SPEC benchmark suite when reassembled using the Locality Layout algorithm compared to the performance of the applications reassembled using the default algorithms of the Reassembly phase. On Hosts A and B there was a 2.666% and 2.476% improvement, respectively.

The results for the key characteristic for evaluating the "offline" overhead, filesize overhead, were mixed. On Host A there was an improvement and on Host B there was an impairment but both were within a percent.

Taken together, the results for Hosts A and B demonstrate that the Locality Layout algorithm did improve the overall performance of the computationally bound applications of the SPEC benchmark suite. The original hypothesis was that this optimized reassembly algorithm would improve the RSS, page fault and instruction cache overhead by optimized the placement of code. The results show that this was not the case,

however. In the cases where the Locality Layout algorithm did improve those overheads, the improvements were slight.

7.4.2 CGC

The more than 140 CBs in the CGC dataset provide another way to assess whether the Locality Layout algorithm achieves its stated goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs. See Section 5.6 for a complete discussion of the CGC dataset and the terms used in the description of the evaluation. The CBs in the CGC dataset are more interactive than the computationally bound applications of the SPEC benchmark suite. Comparing the difference between the impact of the Locality Layout algorithm on interactive and computationally bound programs guarantees that the algorithm does not improve one type of application at the expense of another.

The average availability scores for each of the RCBs when reassembled using the Locality Layout algorithm instead of the default algorithms of the Reassembly phase[1] was 87.09%. This results compares well with the average availability score for the RCBs generated by Zipr using the default algorithms of the Reassembly phase (83.42%) – a 4.4% increase.

The distribution of dominators for the RCBs in the CGC dataset generated using the Locality Layout algorithm shows a notable improvement in memory usage. Overall there are 54 RCBs with a perfect availability score compared with only 42 RCBs with a perfect availability score when statically rewritten using the default algorithms of the Reassembly phase. There is decrease in the number of RCBs that have a lower availability score because of memory overhead. Of the RCBs that have less-than-perfect availability, more than 83% are most negatively impacted by their memory overhead. In absolute terms, there is an 10.84% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the default algorithms of the Reassembly phase.

7.5 Conclusion

Unfortunately, the Locality Layout algorithm does not solve the locality problem entirely. The results show, however, that targeting improvements in the way that the rewritten programs are reassembled is key to improving performance of the statically rewritten programs. The next two chapters present different reassembly algorithms that take advantage of the insights gained during the development and evaluation of the Locality Layout algorithm and further improve the performance of the statically rewritten programs/libraries.

[1]The Null Transformation was the only transformation applied to the CBs.

Chapter 8

Profile Layout

8.1 Introduction

The Profile Layout algorithm is an improvement to the basic algorithms of the Reassembly phase (described in Section 4.4) using the insight gained by implementing, testing and analyzing the Locality Layout algorithm (see Chapter 7). As with the Locality Layout algorithm, the goal of the Profile Layout algorithm is to improve runtime performance with respect to CPU utilization and memory overhead.

8.2 The Problem

Recall the brief discussion at the end of the previous chapter. After the Locality Layout algorithm has attempted to place all the target dollops on the same page as the pinned addresses that link to them, it attempts to use the remaining free space to place dollops on the same page in p' as they are in p. At first glance, this seems like a reasonable policy.

Consider the case where the static binary rewriter is rewriting a program p using the Locality Layout algorithm. p has three dollops and a single pinned address at the program's entry point (Figure 8.1). The pinned address *entry_point* targets dollop a (shaded in green). Dollop a is conditionally connected to dollop b (shaded in pink). Dollop b implements a loop and part of the body of that loop is implemented in dollop c (shaded in blue). Because it is only part of the loop body, dollop c always returns control to dollop b. In the original program, all three dollops are on the same page. A user transformation increases the size of dollop a.

Assume that, because of the increase in the size of dollop a due to the user transformation, dollop b no longer fits on the same page in p' as it did in p. Therefore the Locality Layout algorithm places it on a second page. However, dollop c can still fit on that page in p' so the Locality Layout algorithm places it there.

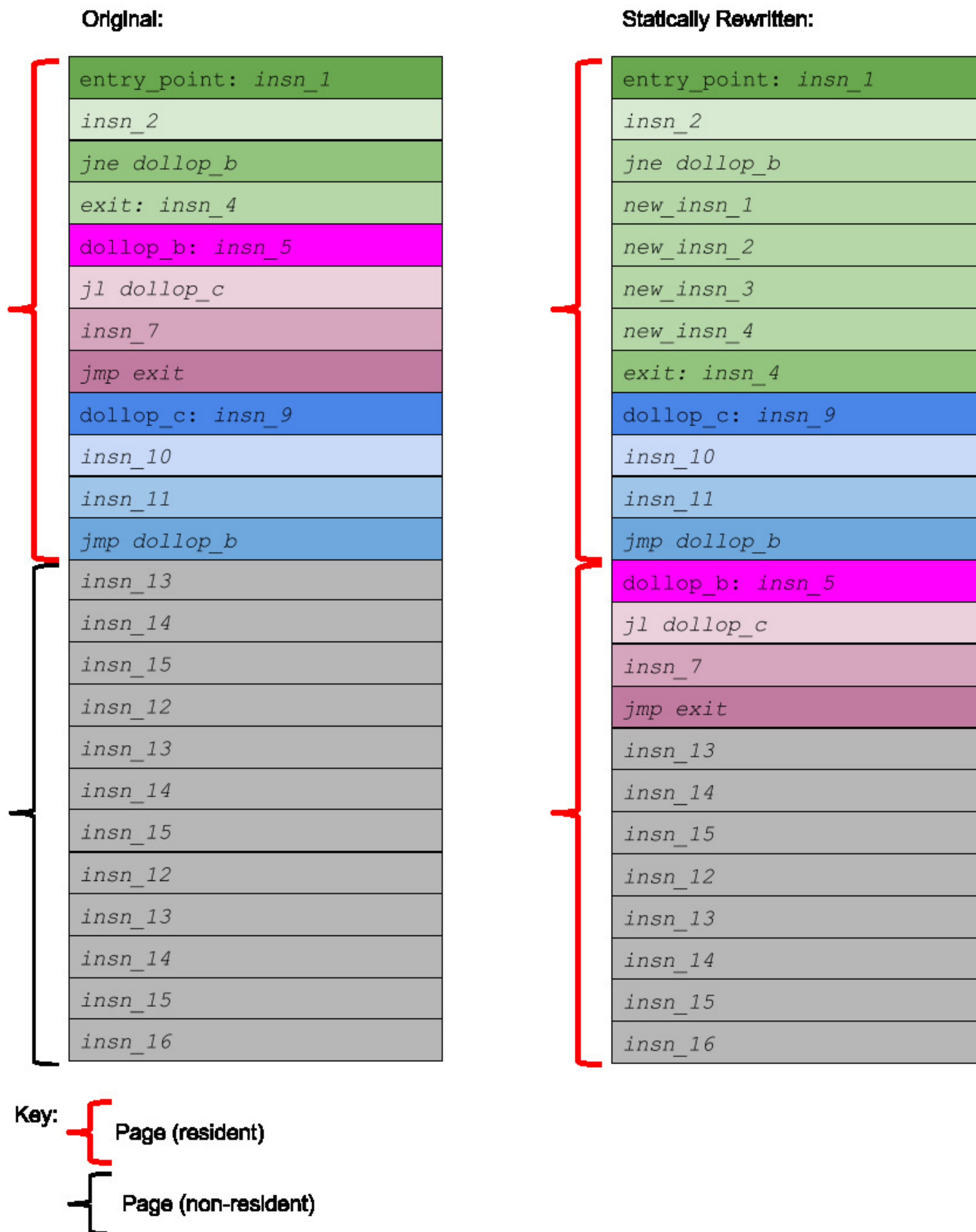

Figure 8.1: An example of a case where the Locality Layout algorithm places a dollop on the same page in the statically rewritten binary that it was on in the original program which results in excessive paging.

The result is perverse and significantly affects the runtime behavior of the program. Assume that the loop embedded in dollops *b* and *c* is executed for many iterations. In the original program, the code for the

entire loop is on the same page. In the statically rewritten program, control will alternate between executing code on two different pages. For optimal performance, the CPU has to keep both pages in the page table cache. The small scale of this example might not betray the consequences of this problem. However, on a low-memory, embedded system or even a powerful server that is running many applications at the same time, keeping these entries in the page table cache simultaneously may not be possible.

Problems with the Locality Layout algorithm also arise in cases where a program's CFG dictates the best code layout. Consider an original program p with three dollops a, b, and c (Figure 8.2). The dollops constitute the entirety of the program and form a single loop. In p, the three dollops all fit on the same page. The user modified dollop a with a user transformation that increased the size of the dollop. As a result, only b *or* c can fit on the same page as a in statically rewritten program p'. If program control transferred equally from dollop a to either b or c, then there is no disadvantage to placing b on page a instead of c, or vice versa. However, if the program executes 100 iterations of the loop and 99 of those iterations go through dollop b instead of c, then placing c on page a is, obviously, not ideal. The proper choice, in this scenario, is to place the *hot* dollop b on the same page as dollop a.

It is not just loops where this problem is found. Empirical results gathered from the analysis of a set of open source Java programs each between 1,000 and 1,600,000 lines of code shows that error handling accounts for as much as 46% of code [240]. However, program performance is independent of the amount of code dedicated to error handling [45]. In other words, if the layout algorithm assumes that a program was equally likely to invoke error handling code as it was to invoke code written for the nominal case, the final program layout will be less than ideal.

8.3 The Solution

The Profile Layout algorithm is based on the analysis of these problematic situations and is an improvement of the Locality Layout algorithm. The improvement is based on the insight that program *profiles* provide information about which instructions are executed most often and in what order. The use of program profiles to improve a compiler's output is not new. In 1990, Pettis and Hanson used program profiles to optimize the code layout of a compiled program [172]. Their work is discussed in depth below in Section 8.6.

8.3.1 Profiles

The Profile Layout algorithm analyzes program profiles to drive placement of code in the statically rewritten binary. The layout algorithm itself is discussed in Section 8.3.3. Before describing the Profile Layout algorithm, though, it is important to understand what a profile reveals about the execution of a program. Profiles

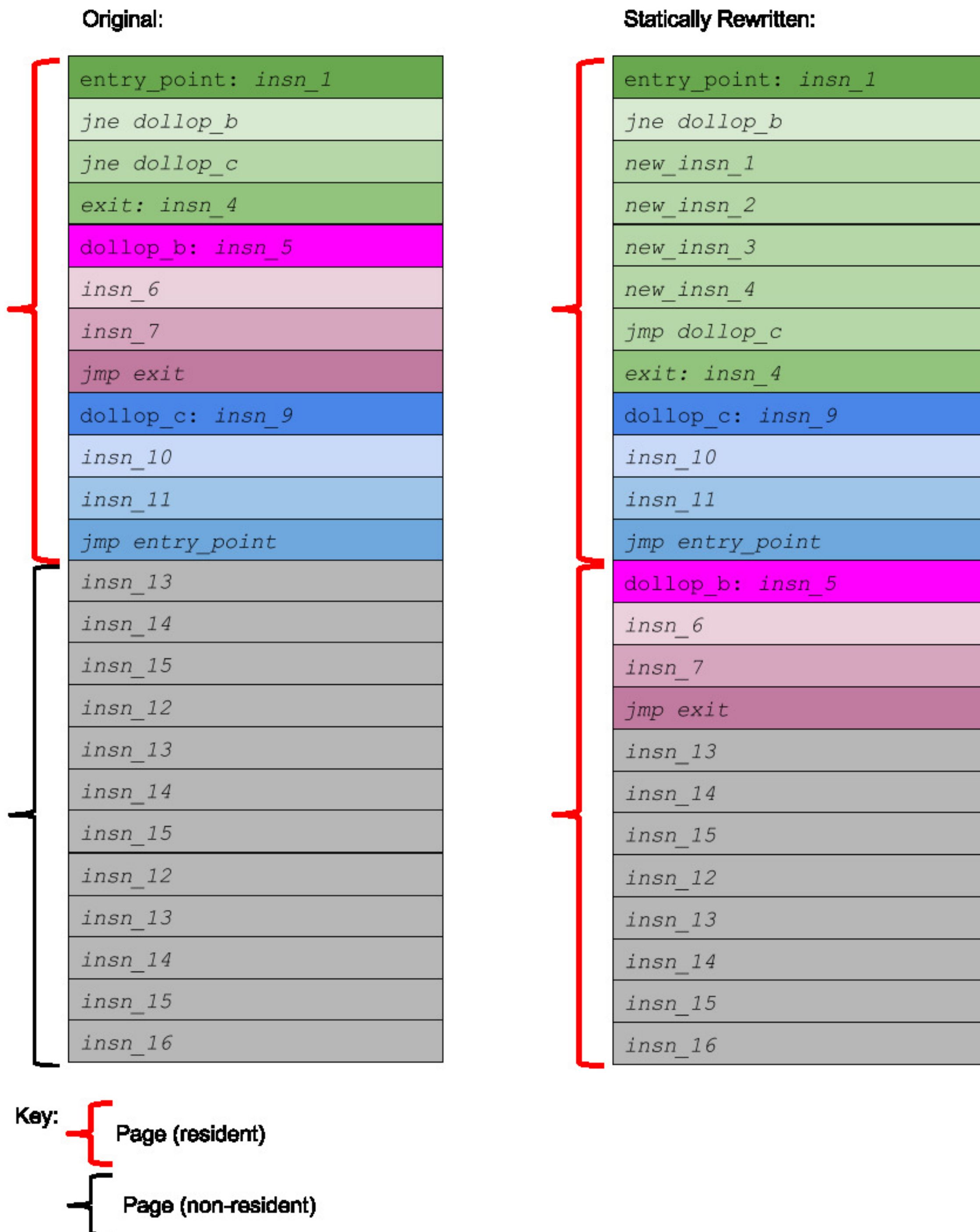

Figure 8.2: An example of a case where the Locality Layout algorithm cannot take advantage of hints from the original program's CFG to place the *hot* connected dollops on the same page.

are commonly classified into one of three categories [31]. There are vertex profiles (sometimes called node profiles [209]), edge profiles and path profiles. Vertex profiles store counts of the number of times that each

basic block is executed. Edge profiles store counts of the number of times that transitions between basic blocks, edges, are taken. Finally, path profiles store counts of the number of times that particular sequences of edges, paths, are executed.

The profiles used by the Profile Layout algorithm do not fit neatly into a single category. The profiles used by the Profile Layout algorithm store two pieces of information for every instruction. First, it stores a count of the number of times that instruction executed. Second, for the program's branches, it stores the number of times the branch is taken. Storing the former data point makes the profile similar to a vertex profile; storing the latter data makes the profile similar to an edge profile.

Although a single profile of a program provides useful information, an aggregation of profiles from multiple executions of a program provides more information. Information from multiple executions masks the program's idiosyncratic behavior and allows the program's dominant behavior to become evident. That said, the inputs that drive the repeated executions that generate the aggregate profiles must be representative of the program's input in real world conditions. If the executions are not representative of the way that a user runs the program, an algorithm that builds a statically rewritten binary program based on those profiles will generate an inefficient layout. See Section 8.4 for more discussion about the importance of using profiles that representative of program behavior in the nominal case.

The accumulated profiles generated from many program executions are combined into a single aggregate profile. This aggregate profile conveys several pieces of information that the Profile Layout algorithm uses to reassemble an efficient statically rewritten binary.

First, it tells the Profile Layout algorithm which instructions, if any, are not executed at all, a strong signal to the algorithm that these instructions can be placed anywhere in the statically rewritten binary without affecting performance. By separating cold code into a separate section of the output binary, more space is available to optimally place hot code. This prioritization of hot code over cold code decreases the likelihood that code that should go on a particular page will be placed on a different page because there is no room one of the reasons that a Locality Layout placement may fail (See Section 7.3, above).

Second, it tells the algorithm which instructions are executed most often, an indication to the algorithm that these instructions need to be placed very carefully because the speed and efficiency of their execution will dictate the performance of the statically rewritten program. The profile information is very helpful for the cases described above where the Locality Layout algorithm lacks the information necessary to make an informed placement decision. The profile information also solves the second of the two problems outlined at the beginning of this section where only a subset of a loop's code can be colocated on a single page and not every block of code implementing that loop is executed equally. See Figure 8.2.

Finally, the profile is able to convey to the algorithm the affinity between connected dollops. In other words, the profile information tells the Profile Layout algorithm which dollops transfer control to other dollops and how often they perform this transfer. The more that two dollops invoke one another, the more important it is that those blocks of code be placed on the same page in order to promote efficient use of caches. This solves the first problem outlined at the beginning of this section where two interrelated dollops are placed on different pages. See Figure 8.1.

8.3.2 Collecting and Storing Profiles

Profiles are generated by a custom tool implemented for the Pin Dynamic Binary Instrumentation ("Pin") Tool [133] and stored in a binary protocol on disk. A library and tools are available for merging multiple profiles into aggregate profiles and analyzing those profiles.

Collected profiles are stored on disk in a binary format in order to save space. Saving the profile files in a canonical, human readable format is impossible given the need to support profiles of programs that have a considerable amount of code and are long running. Support tools exist for converting the binary format to a human readable format to assist debugging and analysis.

The library makes it possible for programs written in C++ to manipulate and access stored profiles. Access to the profiles in third-party applications makes it possible to separate profile generation from profile usage. These libraries have been used beyond this static binary rewriter to implement other algorithms that rely on program profiles.

The most important part of generating a profile is observing a program as it executes and recording information about that execution. Profiles are generated by a Pin Tool, a plugin to Pin. The profile-generating Pin Tool implements a callback function that Pin invokes for every executed instruction. The instruction's address is passed as a parameter to the callback function and is used to accumulate a count of the number of times that instruction is executed. The profile generating Pin Tool also contains a function that Pin invokes for every branch instruction. The parameters to this callback function are the instruction address and a flag indicating whether or not the branch was taken. The profile generating Pin Tool maintains a counter for each branch instruction that is incremented every time the branch is taken.

8.3.3 Algorithm

The Profile Layout algorithm (see Algorithm 5) uses an aggregate profile to make informed decisions about how to reassemble the statically rewritten program. The Profile Layout algorithm operates like an auction as

a means to replicate the Pettis and Hanson profile guided code position algorithm [172]. Conceptually, the algorithm is very simple to understand – the key to its effectiveness.

Algorithm 5 Override the default behavior of the Reassembly phase to place a dollop according to the Profile Layout algorithm.

```
1: bid_list ← ∅
2: function CreateBids(DollopList dl, Profile p)
3:     for all d ∈ dl do
4:         for all i ∈ d do
5:             if IsPinned(i) then
6:                 if ProfileExists(p) then
7:                     bid ← NewPinnedAddressBid(OriginalAddress(i), d, ProfileCount(p, i))
8:                 else
9:                     bid ← NewPinnedAddressBid(OriginalAddress(i), d, DefaultPinnedBidValue)
10:                end if
11:            else
12:                if ProfileExists(p) then
13:                    bid ← NewBid(d, TargetDollop(i), ProfileBranchTakenCount(p, i))
14:                else
15:                    bid ← NewBid(d, TargetDollop(i), DefaultBidValue)
16:                end if
17:            end if
18:            Insert(bid_list, bid)
19:        end for
20:        if HasFallthroughDollop(d) then
21:            fallthrough ← FallthroughDollop(d)
22:            fallthrough_ins ← FirstInstruction(d)
23:            Insert(bid_list, NewBid(d, fallthrough, ProfileCount(p, fallthrough_ins)))
24:        end if
25:    end for
26: end function

27: function AwardBids(BidList bl)
28:    while not IsEmpty(bl) do
29:        while b ∈ bid_list do
30:            awarded_page ← null
31:            if not IsPinnedBid(b) then
32:                awarded_page ← PageOf(b.PinnedAddress)
33:            else if IsPreplaced(b.Bidder) then
34:                awarded_page ← PageOf(PrePlacement(b.Bidder))
35:            end if
36:            if awarded_page not null then
37:                success ← false
38:                if IsPinnedBid(b) then
39:                    success ← PrePlaceAtAddress(b.Target, b.PinnedAddress)
40:                end if
41:                if not success then
42:                    success ← PrePlaceOnPage(b.Target, awarded_page)
43:                end if
44:                if not success then
45:                    success ← PrePlaceOnEmptiestPage(b.Target)
46:                end if
47:                if not success then
```

```
48:                 success ← PrePlaceOnInfinitePage(b.Target)
49:             end if
50:             DeleteBid(b)
51:             break
52:         end if
53:     end while
54:   end while
55: end function

56: function AddressDollop(Dollop tp, Address source)
57:   return Preplacement(tp)
58: end function
```

Description

Function CreateBids populates a list of bids, stored in the set *bid_list*. There are two types of bids, the *dollop bid* and the *pinned address bid*. The two share in common a bidder, a target and bid value. Each is described in more detail below. The set *bid_list* is sorted according to the value of each bid. If a new bid is inserted in *bid_list* with a bidder and a target that match an existing bid, the values of the two bids are combined. Function AwardBids acts on the bids in the *bid_list* and it preplaces each bid's target according to the preference of the bidder, a preference that varies depending whether the bid is a dollop bid or a pinned address bid.

Functions CreateBids and AwardBids are executed before the Reassembly phase begins. When the Reassembly phase invokes this algorithm's AddressDollop function, the Profile Layout algorithm responds by simply looking at its existing list of preplacements.

Again, the algorithm is driven by a list of *dollop bids*, a triple of a source dollop, a target dollop and a value, sorted in descending order based on that value. The dollop bid is the way the source dollop informs the Profile Layout algorithm about its interest in having the target dollop placed on the same page. Upon consideration of each bid, if the source dollop is already placed (Line 33 in Algorithm 5, the algorithm places the target dollop on the same page as the source. If the source dollop is not already placed, the next most valuable bid is considered (Line 29 in 5). This process continues until a bid is selected whose source dollop is already placed. Once the algorithm places the target dollop, that dollop submits a series of dollop bids where it is the source dollop for each bid and its connected dollops are the targets. The values of these new dollop bids are based on the number of times that the aggregate profile indicates that the branch with that target is taken. Should a single dollop contain multiple branches to the same connected dollop, the number of times those branches are taken are combined to determine the bid value. When there is no execution frequency data for a source/target pair in a profile, the Profile Layout algorithm assigns the bid a value of 5.

For example, dollops a and b have both been placed on page g. There is some room left on page g but not enough to accommodate every connected dollop of dollops a and b. Dollop a has two branch instructions that link it to dollop c and d, respectively. Dollop b has two branch instructions that connect it to dollops e and f, respectively. The aggregate profile indicates that dollop a branches to dollop c 5 times and dollop d 4 times. The aggregate profile indicates that dollop b branches to dollop e 6 times and dollop f 3 times. Dollops e and f only branch back to a and b, respectively, 10 and 2 times, again, respectively. The ordered list of dollop bids looks like this:

1. dollop b bids 6 for dollop e
2. dollop a bids 5 for dollop c
3. dollop a bids 4 for dollop d
4. dollop b bids 3 for dollop f

When the algorithm chooses the next dollop to place, it will select dollop b's dollop bid for dollop e. The result is that dollop e will be placed on the same page as dollop b. Once dollop e is placed, it will submit dollop bids for pages it targets. The ordered list of dollop bids now looks like this:

1. dollop e bids 10 for dollop a
2. dollop a bids 5 for dollop c
3. dollop a bids 4 for dollop d
4. dollop b bids 3 for dollop f

The algorithm examines the highest dollop bid, dollop e's bid for dollop a. Because dollop a is already placed, the bid is simply removed from the list. The ordered list of bids now looks like this:

1. dollop a bids 5 for dollop c
2. dollop a bids 4 for dollop d
3. dollop b bids 3 for dollop f

The algorithm examines the next highest dollop bid, dollop a's bid for dollop c. The result is that dollop c will be placed on page g.

Figure 8.3 shows the layout of the code in memory at this point in the process of statically rewriting the original program.

This example demonstrates how the Profile Layout algorithm uses the information from an aggregate profile to solve the problems with the basic algorithms in the Reassembly phase and the Locality Placement algorithm.

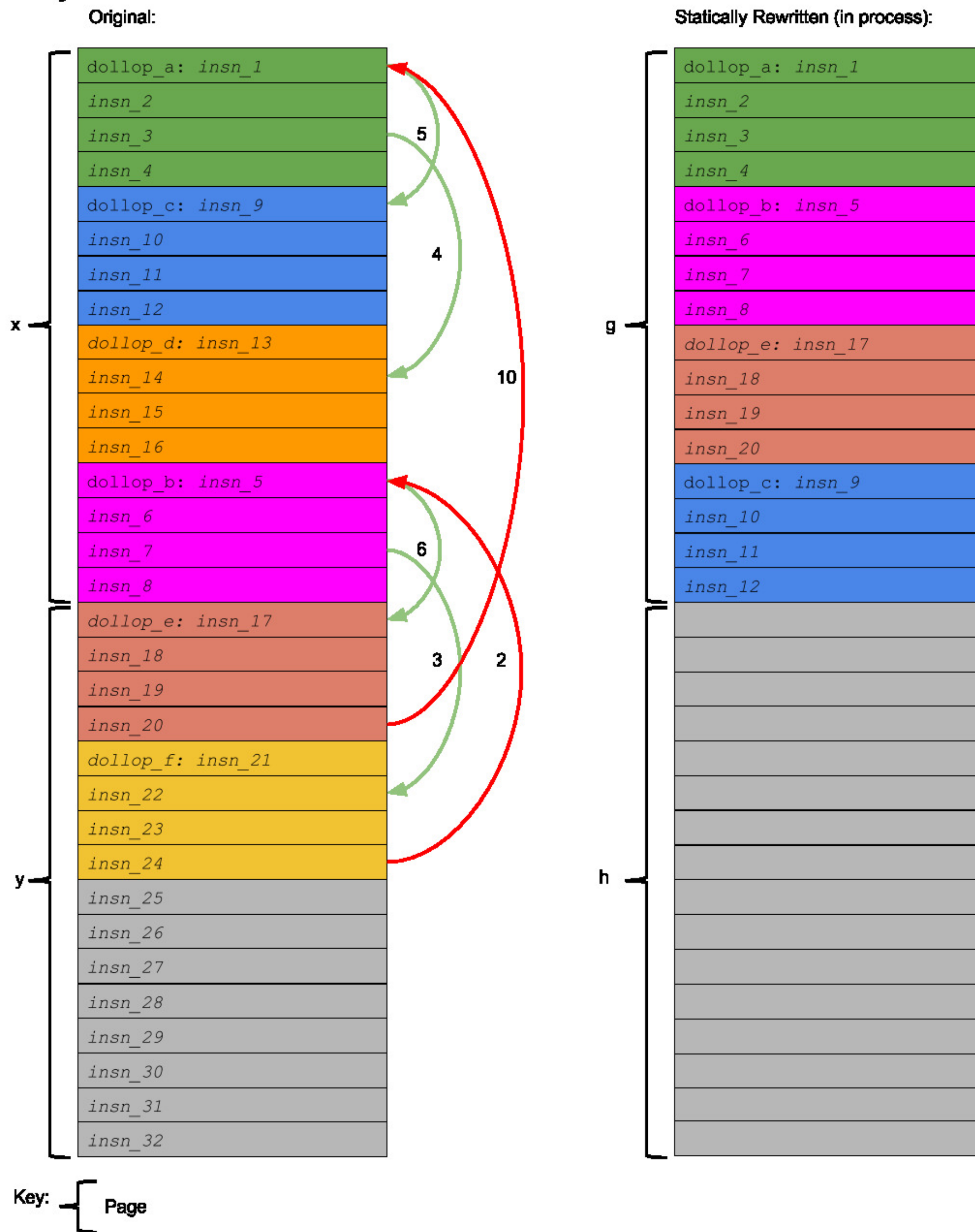

Figure 8.3: An example of how information from a profile guides the Profile Placement algorithm in its decision of where to place dollops.

While this example and the explanation so far demonstrates the algorithm's behavior as it performs placements based on existing placements of dollops, the *placement* phase, it does not explain how the algorithm makes choices about where to initially place dollops. In other words, it does not illustrate the

Profile Layout algorithm's *bootstrap* phase. The Profile Layout algorithm's bootstrap process resembles the function of the Locality Layout algorithm but in terms of the Profile Layout algorithm's bid mechanism.

The Profile Layout algorithm's ordered list of bids is initially populated with a special type of bid known as a *pinned address bid*, a triple of a source pinned address, a target dollop and a value. The pinned address bid conveys to the Profile Layout algorithm the frequency that a pinned address is reached during program execution. The insight used to build the Locality Layout algorithm was that it is advantageous for program performance if a pinned address' target dollop is placed on the same page as the pinned address itself. The pinned address bid does exactly this *and* prioritizes those placements based on the frequency of a dollop's execution. If there is no actual data for the frequency of execution of a dollop targeted by a pinned address, the Profile Layout algorithm assigns a default pinned address bid of 25. Assigning a higher default value for pinned address bids than for a dollop's bid for a target prioritizes the pinned address bids in order to maintain the benefit to program performance if a pinned address' target dollop is placed on the same page as the pinned address itself.

Consider the case where a page g contains two pinned addresses a and b that target dollop c and d, respectively. Because of user transformations that expand dollops c and d only one will fit on page p. The aggregate profile indicates that pinned address a is reached 10 times and pinned address b is reached 100 times. The Locality Layout algorithm would choose one of the two dollops arbitrarily and place it on page g. Obviously, however, dollop b should be given priority because it is executed an order of magnitude more often than dollop a. In this case, when the Profile Layout algorithm begins, the ordered list of bids would look like this:

1. pinned address a bids 100 for dollop c
2. pinned address b bids 10 for dollop b

As the Profile Layout algorithm executes, it treats the two types of bids equally and simply responds to the highest bidder at each iteration. In this case, the algorithm would evaluate the top bid and place dollop c on the same page as pinned address a. This is the ideal behavior and solves the problems indicated above.

The final remaining aspect of the Profile Layout algorithm to explain is how it handles a bid from, for example, source dollop a for target dollop b when there is not enough space on dollop a's page to fit dollop b.

By default, the Profile Layout algorithm will place such a dollop on the emptiest page at that point in the construction of the statically rewritten program. The effect of this decision is to keep the relatively crowded pages as free as long as possible in order to leave room on those pages for dollops that do fit and are targeted by dollops already placed. There are, however, other options. One other option is to find the page in the program as it currently exists in construction of the statically rewritten program that offers the best fit. The

policy of choosing the page with the best fit has the effect of decreasing fragmentation of the program's code segment when smaller blocks of free space cannot be used because the dollops are too big. However, this tactic may prevent the algorithm from being able to satisfy future bidders.

Analysis

The above exposition describes the Profile Placement algorithm as if bidding and placement take place concurrently. In the actual implementation of the algorithm, bidding and placement are done serially. CreateBids creates the entire list of bids before AwardBids begins to handle them.

As mentioned above, the list of bids is a list sorted by the bid values in descending order. There are a number of ways to actually implement this type of a list. This analysis assumes that the underlying data structure used to maintain the sorted bid list maintains a pointer to a single element and supports the following methods:

ListInsert Insert an item in L_i time.

ListDelete Delete the item under the list's pointer in L_d time.

ListPeek Read the element under the list's pointer in L_p time.

ListWalk Step the list pointer to the next element in the list in L_w time.

ListFind Find an element in the list and set the pointer to that element in L_f time.

As each bid is handled, its target dollop is placed on a page. There are a number of data structures that can be used to store such a dollop-to-page mapping. This analysis assumes that the underlying data structure mapping between dollops and their pages supports the following operations:

MapInsert Insert an element in M_i time.

MapLoopkup Lookup and element in M_l time.

For the purposes of this analysis, assume that there are L links, P pins and F dollops with fallthrough dollops in the program/library being statically rewritten. Further, assume that after the bootstrap and building process, there are B bids in the list. Note that $B \neq (L + F + P)$. Rather, $P \leq B \leq (L + F + P)$ because bids can be combined.

Analyzing the runtime behavior of the bid creation process means quantifying the number of times that Insert is invoked. Insert, however, does not simply put a bid into the bid list. Recall that a bid with source dollop s and target dollop t of value v will be combined with any existing bid with the same source and

target dollop. Therefore, each INSERT operation is a combination of a $ListFind$, $ListDelete$ (potentially), and $ListInsert$. Therefore, INSERT takes $O(L_f + L_d + L_i)$ time.

In the bootstrap phase of the Profile Layout algorithm, INSERT is called P times, once for each of the pinned addresses in the program. In the process of creating the remaining bids, the INSERT is called L and F times, once for each of the links and dollops with fallthroughs, respectively. Therefore, the runtime of the bid creation process is

$$\begin{aligned} O(L*(L_f + L_d + L_i) + F*(L_f + L_d + L_i) + P*(L_f + L_d + L_i)) = \\ O((L + F + P)*(L_f + L_d + L_i)) \leq \\ O(L_f + L_d + L_i) \end{aligned} \tag{8.1}$$

Analyzing the runtime of the process of awarding bids is more complicated. The AWARDBIDS function operates by iterating over the list of bids until it is empty. Each iteration of this outer loop acts upon one bid although it is not always the first one in the bid list. In the worst case, the inner loop that selects the bid upon which to act has to walk the entire bid list. Walking the entire bid list and peeking at each element to determine whether the bid should be handled takes $O(B*(L_w + L_p))$ time. After the winning bid is chosen and handled, it must be removed from the bid list and its placement must be recorded. In total, each iteration of the inner loop takes $O(B*(L_w + L_p) + L_d + M_i)$ time. Combining the runtime of the inner and the outer loop, AWARDBIDS takes $O(B*(B*(L_w + L_p) + L_d + M_i))$ time.

Finally, as with the Locality Layout algorithm, ADDRESSDOLLOP is invoked D times, where D is the number of dollops in the program/library being statically rewritten. The ADDRESSDOLLOP routine simply finds the location of the dollop in the data structure used to hold the dollop-to-page mappings. Therefore, ADDRESSDOLLOP runs in $O(D*M_l)$ time.

In total, the runtime of the Profile Layout algorithm is

$$O(L_f + L_d + L_i) + O(B*(B*(L_w + L_p) + L_d + M_i)) + O(D*M_l). \tag{8.2}$$

Needless to say, the implementation of the Profile Placement algorithm could be improved. However, this algorithm executes as the original program/library is being statically rewritten and not while it is executing. Only if/when the implementation of this algorithm becomes a bottleneck of the static rewriting process will time be wisely spent on improving its performance.

8.4 The Importance of Profile Data

As discussed in Section 8.3.1, profile-driven optimizations require access to profiles that accurately represent the behavior of the target program in the nominal case. If the profile does not encode the common behavior of the program, the algorithm's layout of the statically rewritten binary will not be ideal with respect to the performance of the target program in the way that it is usually invoked.

The problem is building representative profiles and there is extensive and ongoing research into this topic. The process of building representative profiles can be divided into two different categories (offline and online) and each has its own benefits and drawbacks [111].

In offline profile generation and optimization, the process of generating representative profiles and optimizing the target software according to those profiles is done before the program is deployed to end users. In online profile generation and optimization, the process of generating representative profiles and optimizing the target software according to those profiles is done as the end user interacts with the software after it has been deployed.

Offline techniques for profile generation are limited by the availability of representative program inputs. Representative inputs are required to drive executions of the target program that correlate with the way the end user will ultimately operate the software. Buse et al. put it this way: "Unfortunately, profiling is severely limited by practical concerns: the frequent lack of appropriate workloads for programs, the questionable degree to which they are indicative of actual usage, and the inability of such tools evaluate program modules or individual paths in isolation" [45]. In order to overcome these limitations, Buse et al. developed a technique that analyzes the target program's source code in order to identify the commonly executed paths through the program and generate profiles based on that information. Their technique is able to successfully identify "the 5% of paths that account for over half of a program's total runtime" [45]. While a great advance in generating representative profiles, for the situation described in Part I, such a technique is inapplicable because the static binary rewriter must operate equally as efficiently no matter whether a program's source code is available or not.

Online techniques excel in this respect because profiles are generated at the same time as *actual* users *actually* use the target software. The more the user uses the target software, the more precise the profile becomes. With each execution, the optimization takes the updated profile and reoptimizes the target program. Online profile-driven optimization techniques are in use today in the widely used Chrome browser and have been shown to improve browser start-up time by more than 16% [137]. However, in a case study, Kistler et al. found that "[i]n many situations, the costs of dynamic optimizations outweigh their benefit, so that no break-even point is ever reached" [119]. The impact of the overhead of profiling and reoptimizing at runtime

is especially acute on embedded platforms such as those described in Part I.

Given the limitations to profile-driven optimizations, this layout optimization may not be as effective as it could be. A comparison between the performance of statically rewritten programs using the Profile Layout algorithm seeded with actual, representative profiles and the performance of statically rewritten programs using the Profile Layout algorithm without actual, representative profiles will demonstrate the impact of profiles on the performance of the statically rewritten program. It turns out that seeding the Profile Layout optimization with actual, representative does *not* significantly improve the performance of the statically rewritten program.

Overall, there is a gap between the *ideal* benefits of profile-driven optimizations and the *actual* impact of those optimizations on program performance. Jantz et al. have studied and quantified this difference [111]. To reiterate, for profile-driven optimizations to produce an ideally optimized output program, it is necessary but not sufficient to have a set of tests/inputs that correlate with the way the end user will ultimately operate the software, something that is not easily accomplished for most software.

Inputs/tests are available for the 145 different CBs developed for the DARPA CGC that drive each program in a representative manner (see Section 5.6 for a complete description of the DARPA CGC, CBs, pollers and the testing environment). Therefore, if the profile-driven Profile Layout algorithm cannot produce more efficient statically rewritten versions of the CBs when actual profiles are available, then it is reasonable to conclude that the algorithm operates just as efficiently using the default bid values (as specified in Section 8.3.3) a significant benefit because actual profiles are difficult to find or generate, as described extensively above.

Because generating profiles for all 145 of the CBs in the test set was prohibitively expensive, a subset of 50 of those CBs were selected for profiling and evaluating. The 50 CBs were selected randomly but 6 were excluded. The excluded CBs could not be profiled because the CB did not actually have any pollers (CROMU_00041) or a) produced bad output (KPRCA_00008, KPRCA_00009) or b) failed to execute (KPRCA_00024, KPRCA_00039) in the profiling environment.

Figure 8.4 shows the Availabilities for each of the 46 CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue. The average availability score for the CBs statically rewritten with the Profile Layout algorithm without profiles was 94.72%. The average availability score for the CBs statically rewritten with the Profile Layout algorithm with profiles was 95.57%, a difference of less than 1%.

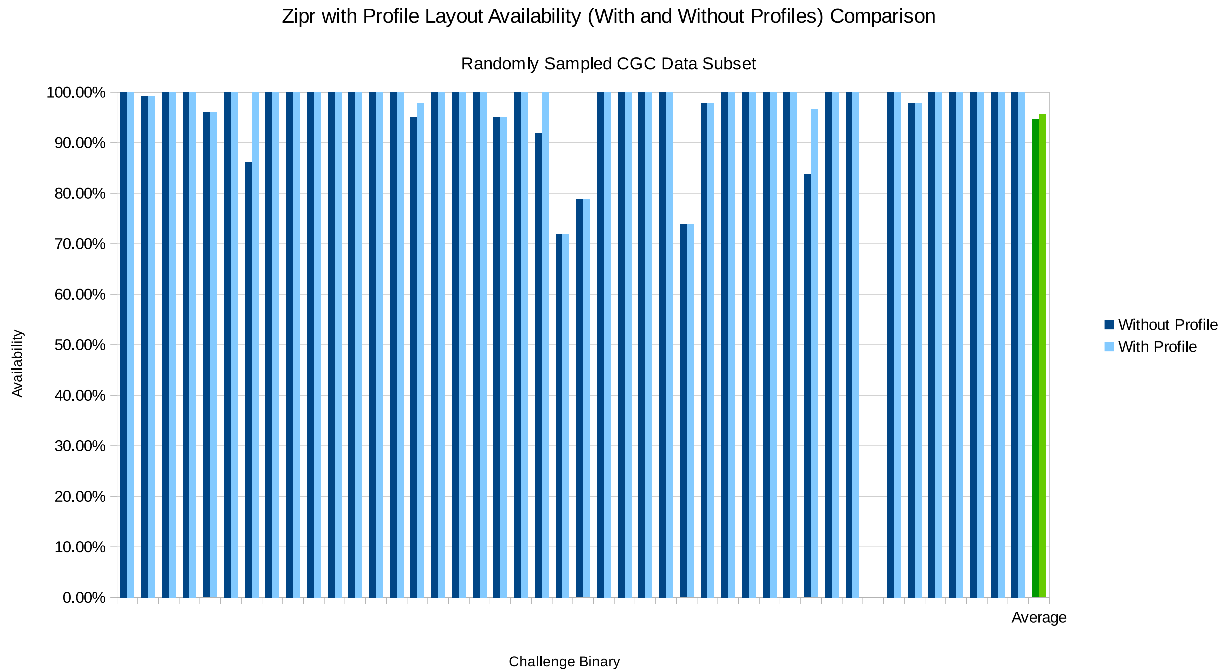

Figure 8.4: Comparison of availability scores for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The availability scores for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green.

Figure 8.5 shows a comparison of the percentage of time the target dollop could be placed on the winning bidder's preferred page for each of the 46 CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue. The average placement ratio for the CBs statically rewritten with the Profile Layout algorithm without profiles was 89.07%. The average availability score for the CBs statically rewritten with the Profile Layout algorithm with profiles was 86.72%. It is noteworthy that the average placement ratio is 2.71% *higher* for the CBs statically rewritten using the Profile Layout algorithm without profiles than with profiles. This is not surprising, however. The profile data, by representing the real interconnectedness of dollops, produces more pressure on the layout algorithm to place more dollops on the same pages in memory. The auction algorithm ensures that the most interconnected dollops end up on the same page, but cannot guarantee that each interconnected dollop ends up on the same page. On the other hand, the default data that the Profile Layout algorithm uses to quantify the interconnectedness and produce the bids does not produce as much pressure which increases the likelihood that the layout algorithm can place interconnected dollops on the same page.

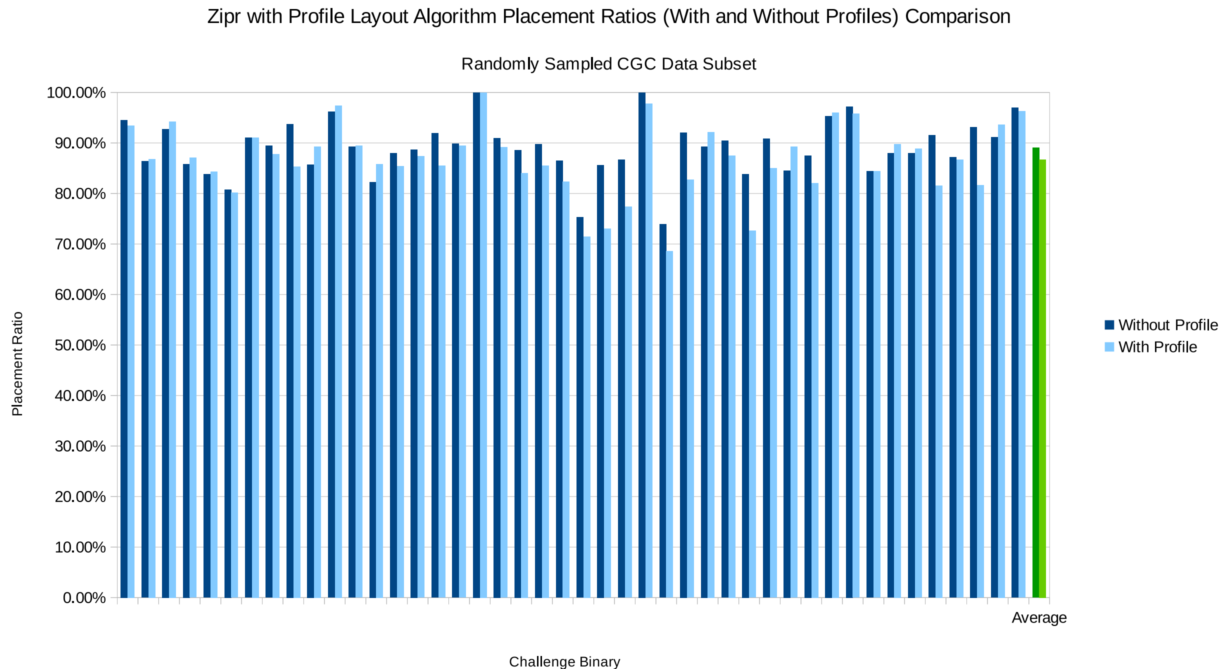

Figure 8.5: Comparison of the percentage of time the target dollop could be placed on the winning bidder's preferred page for each 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The placement ratios for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green.

Figure 8.6 shows a comparison of the execution overhead for each of the 46 CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue. The average execution overhead for the CBs statically rewritten with the Profile Layout algorithm without profiles was 0.98%. The average execution overhead for the CBs statically rewritten with the Profile Layout algorithm with profiles was 0.30%. That is a difference of almost 70% but, in absolute terms, the execution overhead is less than 1% in both cases.

Figure 8.7 shows a comparison of the memory overhead for each of the 46 CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue. The average memory overhead for the CBs statically rewritten with the Profile Layout algorithm without profiles was 5.81%. The average execution overhead for the CBs statically

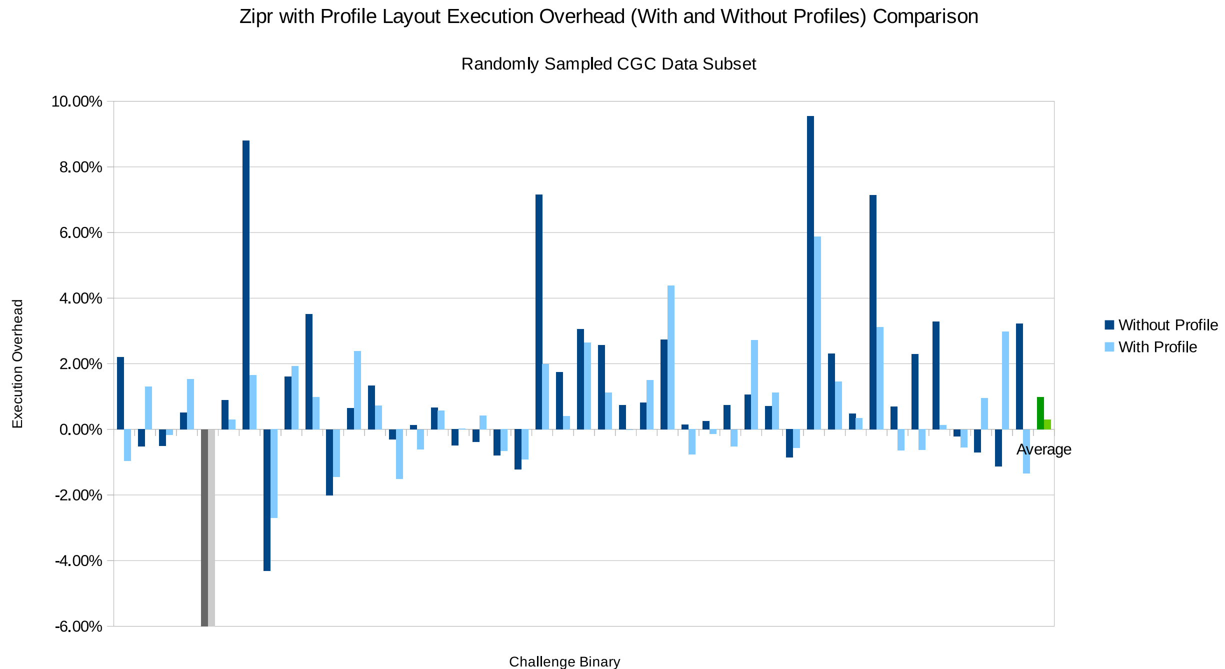

Figure 8.6: Comparison of execution overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The execution overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph.

rewritten with the Profile Layout algorithm with profiles was 5.75%. That is a difference of 1.03%.

Figure 8.8 shows a comparison of the filesize overhead for each of the 46 CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue. The average filesize overhead for the CBs statically rewritten with the Profile Layout algorithm without profiles was 4.19%. The average execution overhead for the CBs statically rewritten with the Profile Layout algorithm with profiles was 4.30%. That is a difference of 2.56%.

The pollers, CBs and measurement tools of the DARPA CGC infrastructure makes it possible to measure whether actual profile data improves the performance of the Profile Layout algorithm.[1] The evidence, the availability scores, the execution overhead, the memory overhead and the filesize overhead, shows that the Profile Layout algorithm does not operate significantly better with profile data than it operates without profile data.

[1] An evaluation of the performance of the Profile Layout algorithm with respect to the Locality Layout algorithm and the default algorithms of the Reassembly phase is discussed in Section 8.5.

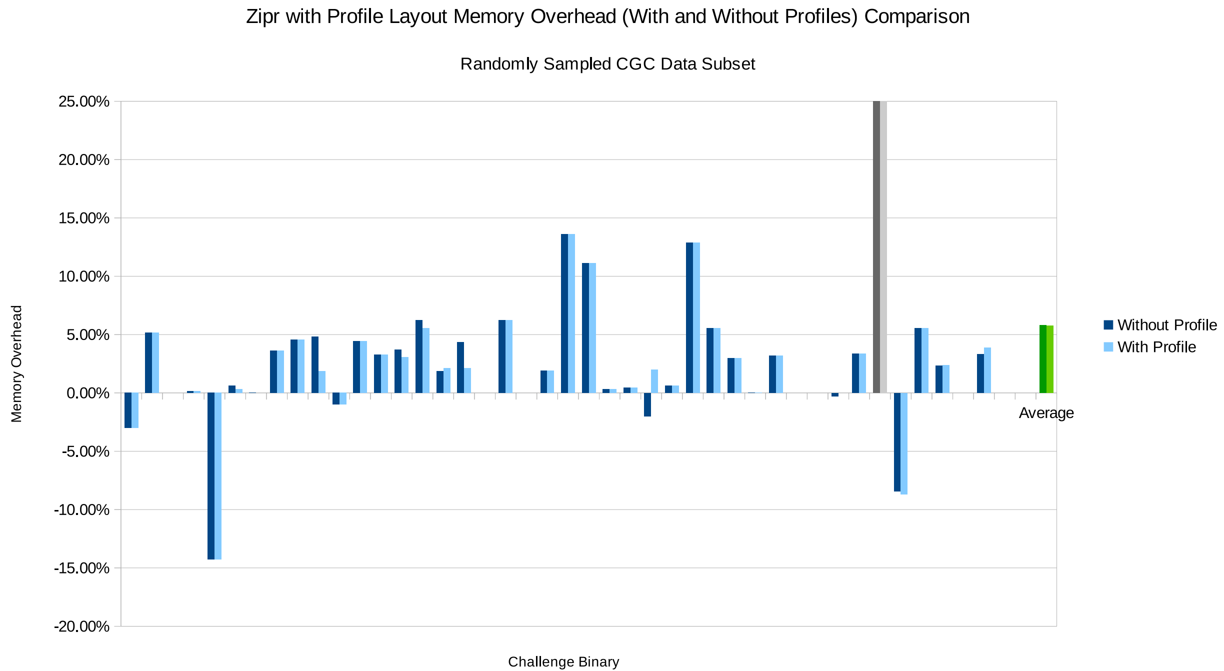

Figure 8.7: Comparison of memory overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The memory overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph.

As discussed above, actual representative profiles are difficult to find/generate. Given the power of the Profile Layout algorithm to improve the efficiency of statically rewritten programs/libraries (see Section 8.5), the fact that actual, representative profiles are not needed to gain the efficiency is a significant advantage for the algorithm. Because there is no difference in performance, in the remainder of this dissertation, all results for tests involving the Profile Layout algorithm will be done without actual profiles.

8.5 Evaluation

The applications of the SPEC benchmark suite and the CBs of the CGC dataset were evaluated in order to assess whether the Profile Layout algorithm achieves the goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs.

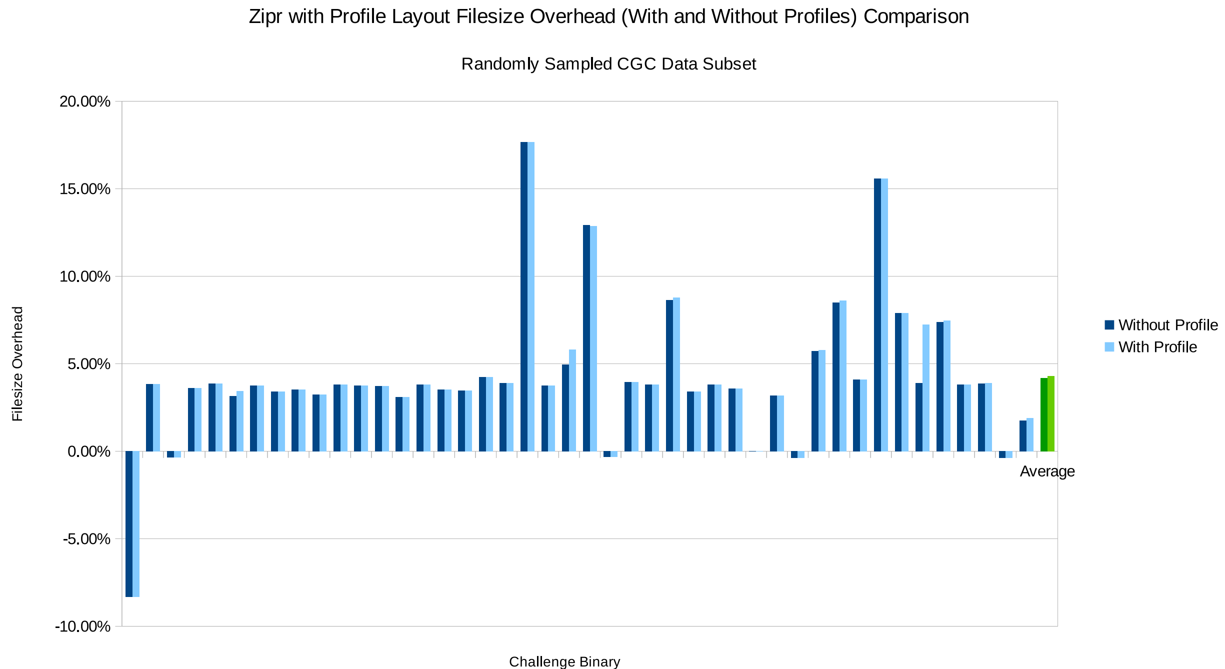

Figure 8.8: Comparison of filesize overhead for the 46 randomly selected CBs when statically rewritten using the Profile Layout algorithm with and without profiles. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm without profiles are shown on the left, in dark blue and dark green. The filesize overhead for the CBs when statically rewritten using the Profile Layout algorithm with profiles are shown on the right, in light blue and light green. Those values shown in (light and dark) gray are clipped to maintain the scale of the graph.

8.5.1 Performance

Figures 8.9 and 8.10 show the difference in performance for each of the applications of the SPEC benchmark suite when those programs are rewritten using the Profile Layout algorithm as opposed to the *Locality Layout algorithm* described in Chapter 7.

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm on Host A was 1.221x. This performance overhead compares favorably (a 5.230% improvement) to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm (1.288x) which itself was an improvement on the average overhead of the applications statically rewritten using the default algorithms of the Reassembly phase (1.323x).

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm on Host B was 1.198%. This performance overhead compares favorably (a 4.412% improvement) to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm (1.253x) which itself was an improvement on

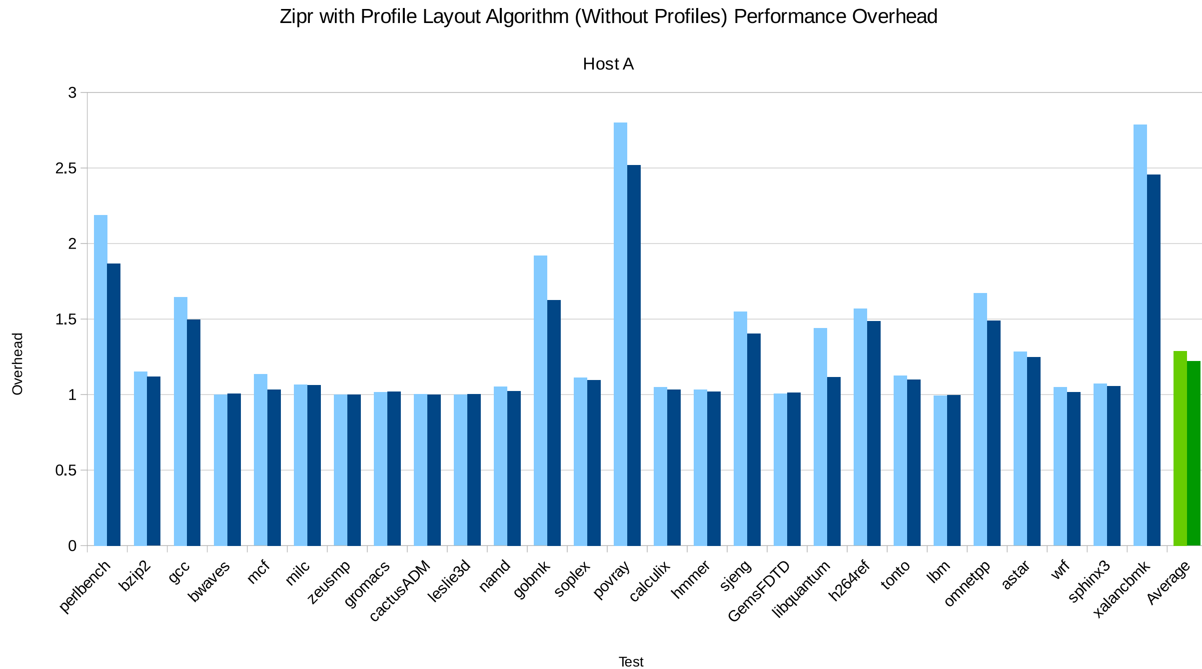

Figure 8.9: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

the average overhead of the applications statically rewritten using the default algorithms of the Reassembly phase (1.285x).

8.5.2 Memory Usage

Figures 8.11 and 8.11 show the difference in the RSS overhead for each of the applications of the SPEC benchmark suite when those programs are rewritten using the Profile Layout algorithm as opposed to the Locality Layout algorithm.

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm on Host A was 1.008%. This is roughly equivalent to the average RSS overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm (1.010%). It is within less than a quarter of a percent.

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm on Host A was 1.011%. This is roughly equivalent to the average RSS

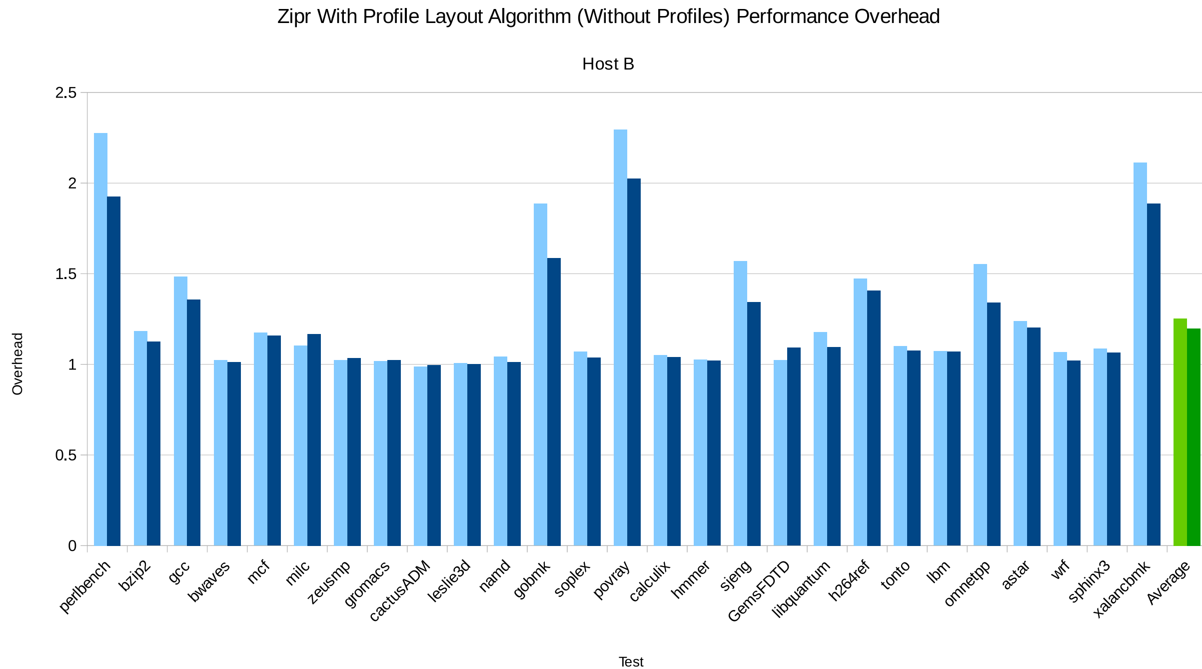

Figure 8.10: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

overhead of the applications of the SPEC2006 benchmark suite statically rewritten with Locality Layout algorithm (1.014%). It is slightly less than three tenths of a percent.[2]

Figures 8.13, and 8.14 show the page fault overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. On Host A, the overhead ranged from 1.136% to 309.630%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.293x more page faults than the native versions. That is comparable (a -0.15% change) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using the Locality Layout algorithm (1.291x). On Host B, the overhead ranged from less than 1.0638% to 289.482%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.284x more page faults than the native versions on Host B. That is comparable (a -0.078% change) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using the Locality Layout algorithm (1.283x).

[2] A complete explanation of the cause of the individual differences between `cactusADM` and `sphinx3` in these results is found in Section 5.5.2

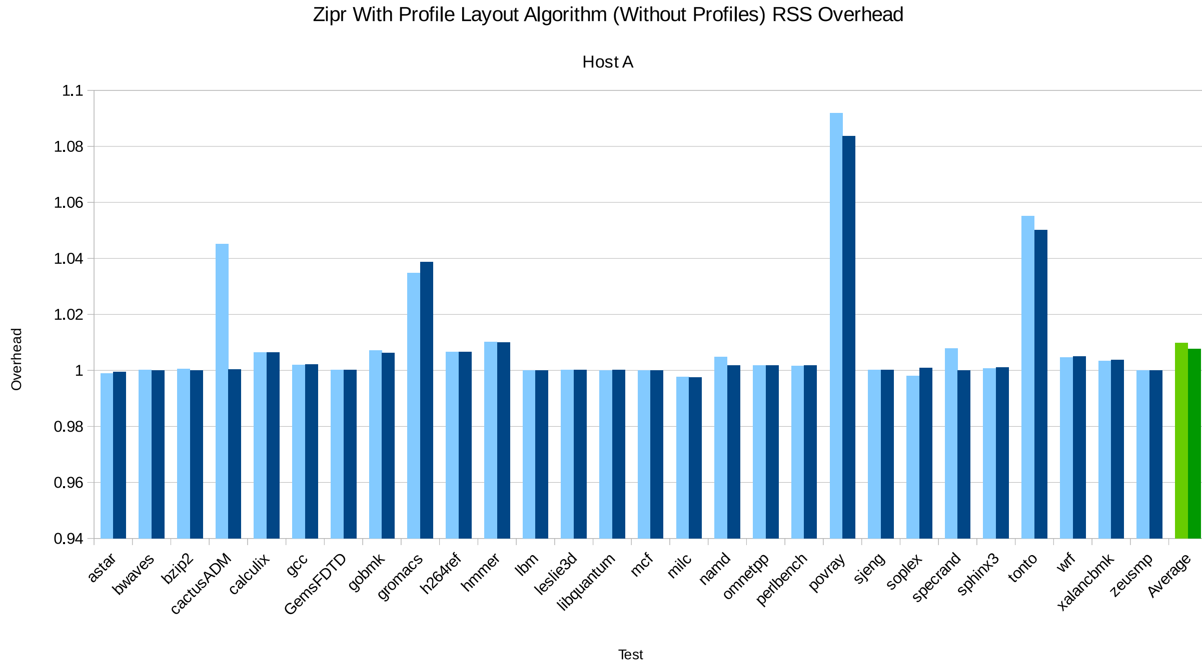

Figure 8.11: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

8.5.3 Instruction Cache Usage

As discussed in Section 5.5.3, the instruction cache is an important part of a host's hardware designed to improve performance. Using an algorithm like the Profile Layout algorithm that reassembles statically rewritten programs and libraries to take advantage of temporal locality will improve the statically rewritten program's use of the instruction cache.

Figures 8.15 and 8.16 show results that prove this assertion. The For Hosts A and B, the versions of the programs of the SPEC benchmark suite statically rewritten using the Layout algorithm incur 3.620x and 4.08x as many instruction cache misses as their native counterparts, respectively. Those overheads are improvements of more than 16% and 12% for Hosts A and B respectively when compared to the instruction cache miss overhead of the programs of the SPEC benchmark suite statically rewritten using the Locality Layout algorithm which itself was already an improvement over the default algorithms of the Reassembly phase. The improvements are significant and contribute to the overall performance improvement discussed earlier.

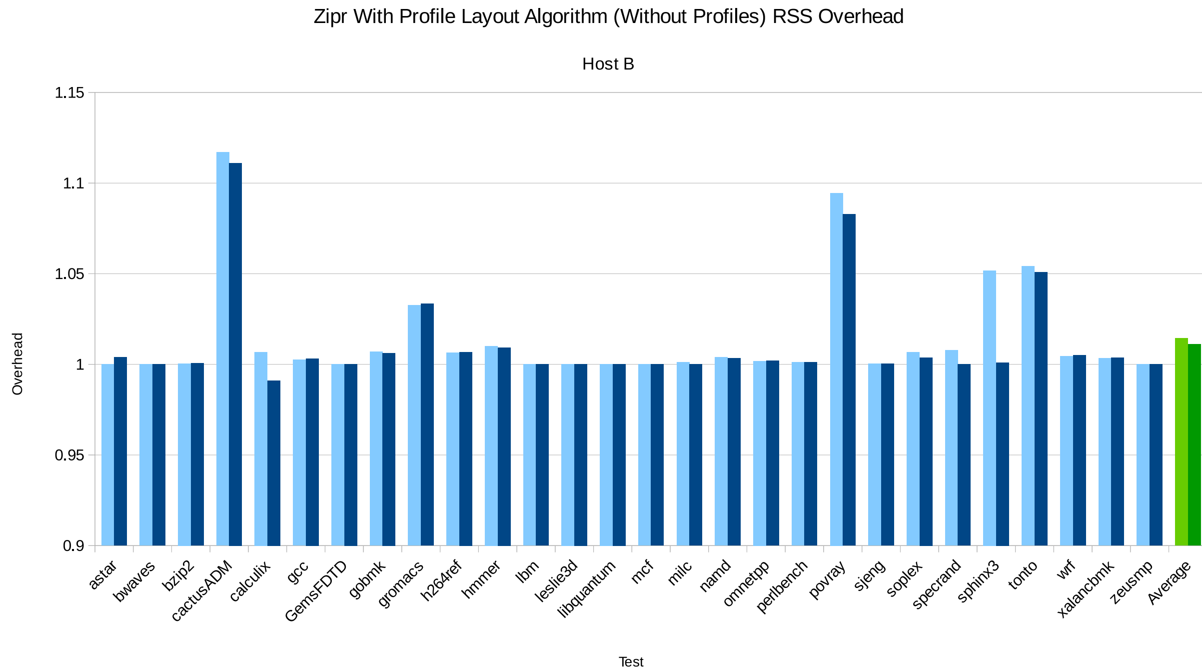

Figure 8.12: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

8.5.4 Filesize Overhead

Figures 8.17, and 8.18 show the filesize overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. For Host A, the average filesize overhead of the applications of the SPEC benchmark suite was 9.512%. For Host B, the overhead was 9.584%. In both cases, the filesize overhead is minimal which again validates the ability of the architecture and algorithms of the static binary rewriter as implemented by Zipr to produce statically rewritten programs with low on-disk overhead [216, 11].

8.5.5 Placement Ratios

As described in Section 8.3.3, there are scenarios under which the Profile Layout algorithm cannot place the winning bidder's target dollop on the preferred page. Results gathered during the process of statically rewriting the applications of the SPEC2006 benchmark suite when using the Profile Layout algorithm's placement ratio demonstrate that this is more than a hypothetical problem. Figure 8.19 shows the placement ratios for Hosts A and B. As with the placement ratio results for the Locality Layout algorithm, the results are similar for both hosts because the placement ratio is a function of the application being rewritten and

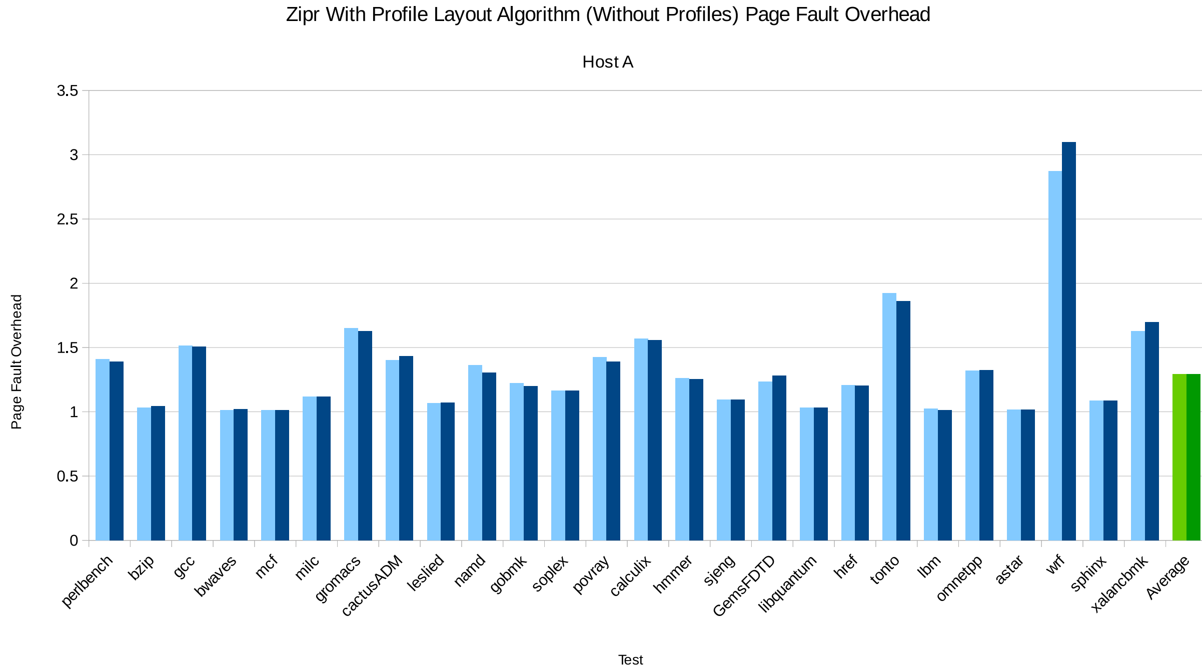

Figure 8.13: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

independent of the underlying host. Since the input applications for Hosts A and B are built from the same source code using the same compiler, it is expected that their placement ratios are similar. The placement ratios for Hosts A and B are 72.66% and 72.71%, respectively.

8.5.6 CGC

The more than 140 CBs in the CGC dataset provide another way to assess whether the Profile Layout algorithm achieves its stated goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs. See Section 5.6 for a complete discussion of the CGC dataset and the terms used in the description of the evaluation.

Figure 8.20 shows the availability scores for each of the RCBs generated by Zipr using the Profile Layout algorithm instead of either the default algorithms of the Reassembly phase or the Locality Layout algorithm.[3] The average availability score was 94.19%. This compares well with the average availability score for the RCBs generated by Zipr using the default algorithms of the Reassembly phase (83.42%) and the average

[3]The Null Transformation was the only transformation applied to the CBs.

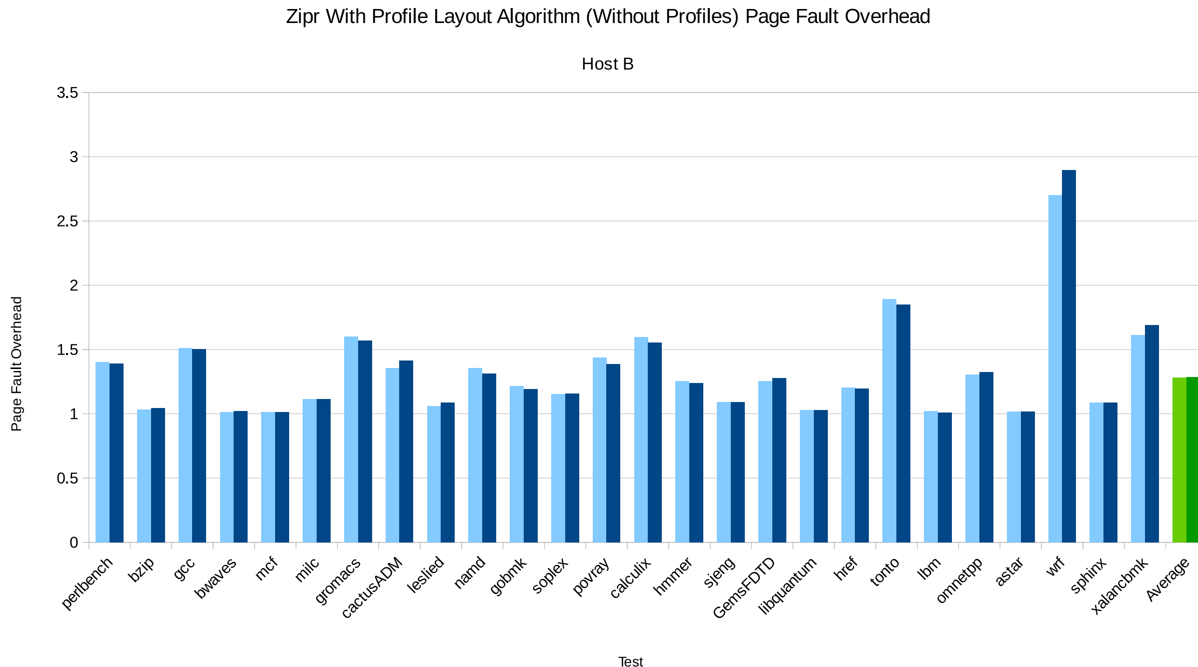

Figure 8.14: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

Category	Default	Locality	Profile
None	42	54	100
Performance	16	12	12
Filesize	1	2	0
Memory	83	74	29
Functionality	1	1	2

Table 8.1: Distribution of dominators for the RCBs in the CGC dataset generated using the Profile Layout algorithm.

availability score for the RCBs generated by Zipr using the Locality Layout algorithm (87.09%) – a 12.13% and 7.83% increase, respectively.

Table 8.1 shows the distribution of dominators for the RCBs in the CGC dataset generated using the Profile Layout algorithm. Overall there are 102 RCBs with a perfect availability score compared with only 47 RCBs with a perfect availability score when statically rewritten using the default algorithms of the Reassembly phase and 60 RCBs with a perfect availability score when statically rewritten using the Locality Layout algorithm. The distribution shows a notable decrease in the number of RCBs that have a lower availability score because of memory overhead. Of the RCBs that have less-than-perfect availability, more than 67% are most negatively impacted by their memory overhead. In absolute terms, there is a 96.43% improvement in the

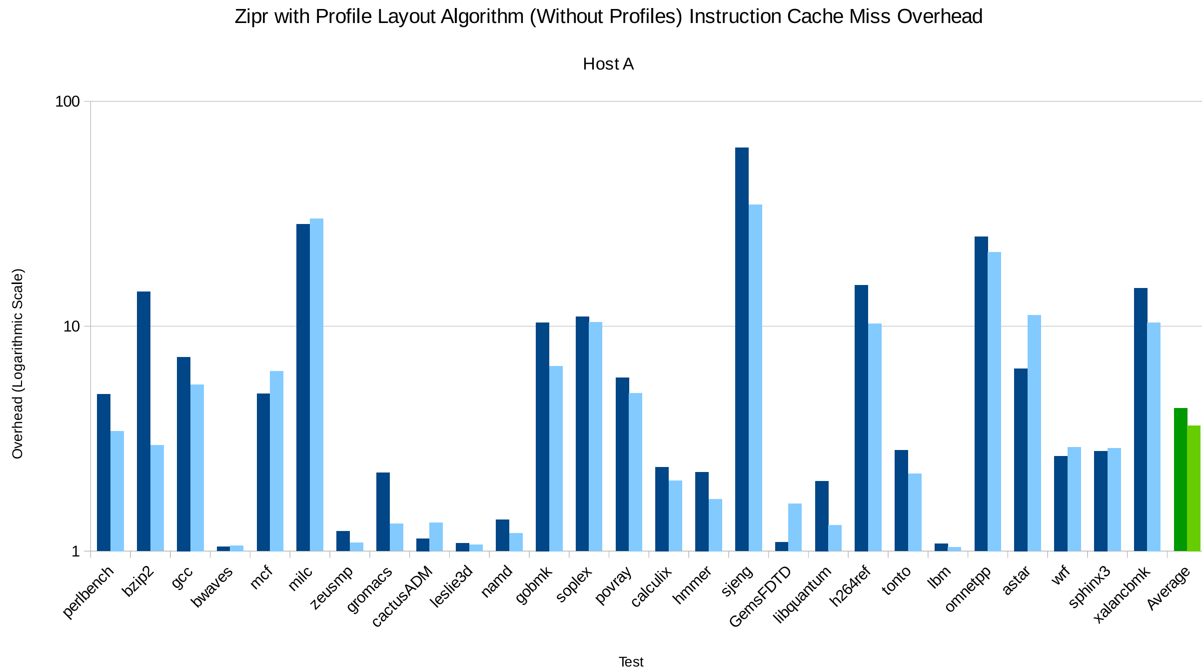

Figure 8.15: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm without profiles. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the default algorithms of the Reassembly phase and a 87.38% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the Locality Layout algorithm.

8.6 Related Work

In 1990, researchers Karl Pettis and Robert Hansen of Hewlett Packard explored the idea of using profiles to guide the code placement. The two eventually incorporated their algorithms into a production compiler that proved the utility of using profiles to guide code layout.

The researchers recognized the importance of optimizing a compiled program with respect to the instruction memory hierarchy. Even in 1990, they recognized that CPU performance was increasing faster than memory performance. In other words, CPUs were getting faster at executing instructions at a faster rate than memory controllers were getting better at fetching instructions from memory. To accommodate this growing gap,

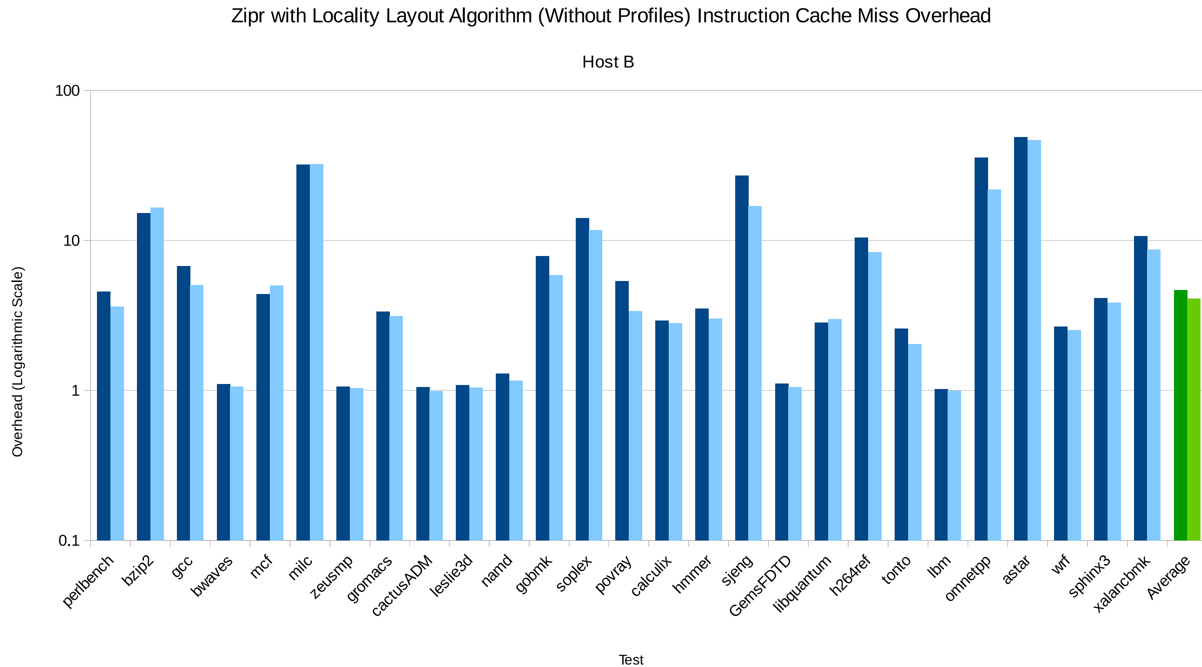

Figure 8.16: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Profile Layout algorithm without profiles. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

system architects began adding instruction caches and the researchers at HP recognized the importance of using these caches effectively to get the best system performance.

The gap is known as the Memory Wall, a term coined by University of Virginia scientist William Wulf, [245] and has not closed in the years since HP's work. In fact, there is an entire area of Computer Science dedicated to overcoming this gap in the context of the computation on big data sets. [199]

The HP researchers built two separate techniques to "reduce the overhead of the instruction memory hierarchy." The first is a technique that uses profiles to place interconnected procedures on the same page. The second is a technique that uses profiles to determine which basic blocks of a procedure are interconnected and rearranges the position of those inter-procedure blocks to maximally exploit locality.

The function placement technique in their work was the most influential on the Profile Layout algorithm presented here and the analysis that drove design of the Profile Layout algorithm parallels the HP researchers' reasoning for developing their techniques. Pettis et al. hypothesized that exploiting spatial locality among functions that "call[] [one] another frequently" will "reduc[e] the size of the working set . . . and should result in fewer page and TLB misses." They also point out that instruction caches are organized by pages which means that "placing two procedures next to one another minimizes the overlap in cache lines between them."

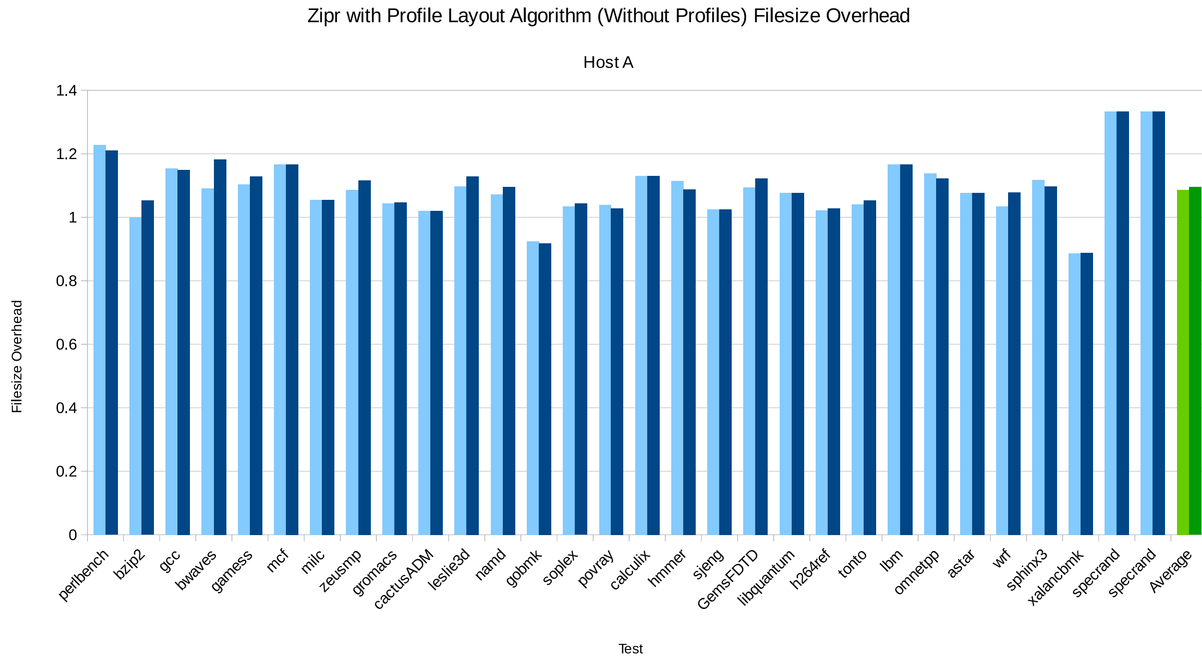

Figure 8.17: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

Their function placement technique is basically a graph algorithm. A graph is built whose nodes represent the functions and weighted edges represent calls between functions. The weights assigned to those edges are drawn from the profile information. Their algorithm loops over the graph and collapses nodes together until only unconnected aggregate nodes remain. At each iteration, the two nodes in the graph with the heaviest edge weight are selected for placement next to one another in the optimized program. For the remainder of the placement algorithm, those two nodes are considered one.

The researchers developed and tested two separate algorithms for basic block placement. The first algorithm is a bottom-up algorithm and the second algorithm is a top-down algorithm.

The bottom-up algorithm is very similar to the algorithm they use for function placement. The outcome of the algorithm are an ordered list of chains of basic blocks. The algorithm starts by considering each block as a chain of one block. The algorithm iterates through the weighted edges between chains. For the chains linked by the heaviest edge, the algorithm decides if the chains can be merged. Two chains can be merged if the tail of one chain links to the head of the other.

The algorithm uses the arcs between chains that cannot be merged to define the function that defines the ordering between chains. The algorithm uses the weights of these arcs to make sure that the chains are

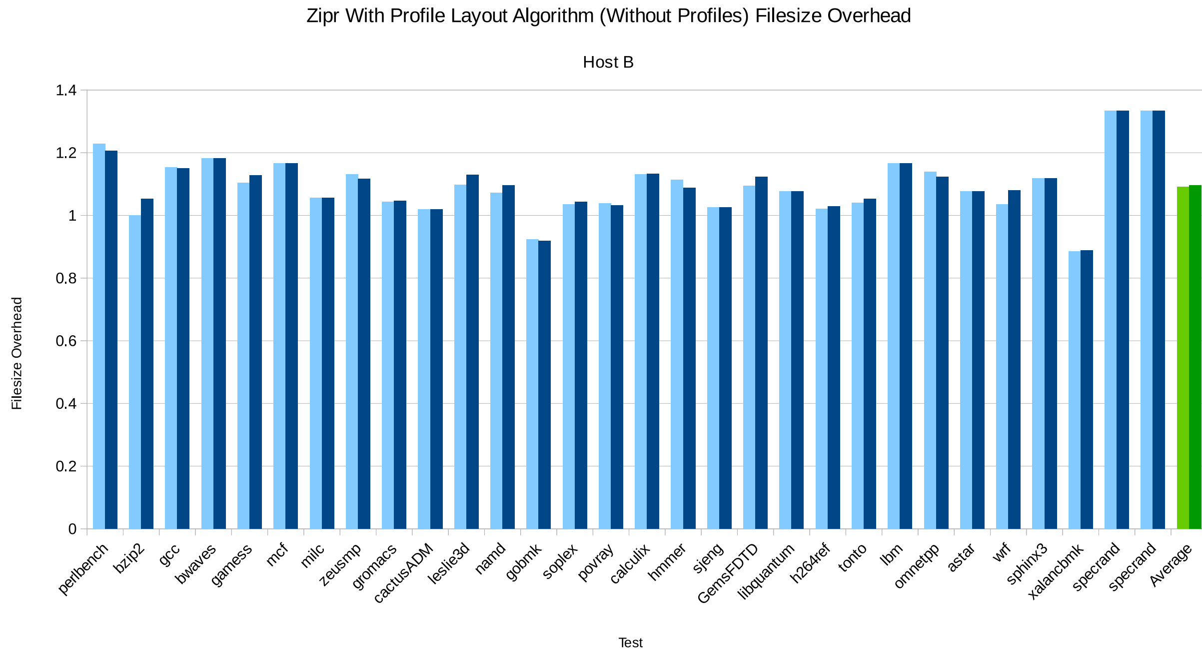

Figure 8.18: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Locality Layout algorithm are shown on the left, in light blue and light green.

ordered to cooperate with the hardware's assumption that forward branches are taken less often than reverse branches.

The top-down algorithm is based on an observation that most basic blocks have two successors and that the relative positioning of basic blocks should cooperate with the hardware branch predictor's predilection for backward branches. The algorithm builds an ordering of basic blocks one block at a time. The ordering begins with the procedure's entry block. At each iteration, the weights between the successors of the previously placed blocks are examined. The successor block connected to the most recently placed block with the heaviest edge is selected to be placed next. If that block is already placed, then the non-successor blocks are surveyed to find the one whose source is the heaviest edge that is already in the list.

The researchers chose to implement only the bottom-up algorithm and subsequently enhanced that algorithm with procedure splitting. The most often executed basic blocks of a procedure are put at the beginning of the resulting intra-procedure order of basic blocks. The basic blocks at the end of the list are called the function's *fluff*. The fluff basic blocks are then removed from the function and placed separately. Splitting the function like this means that, when combined with the function placement algorithm, only the hottest blocks of interconnected functions are colocated on the same page. This means that, overall, a higher

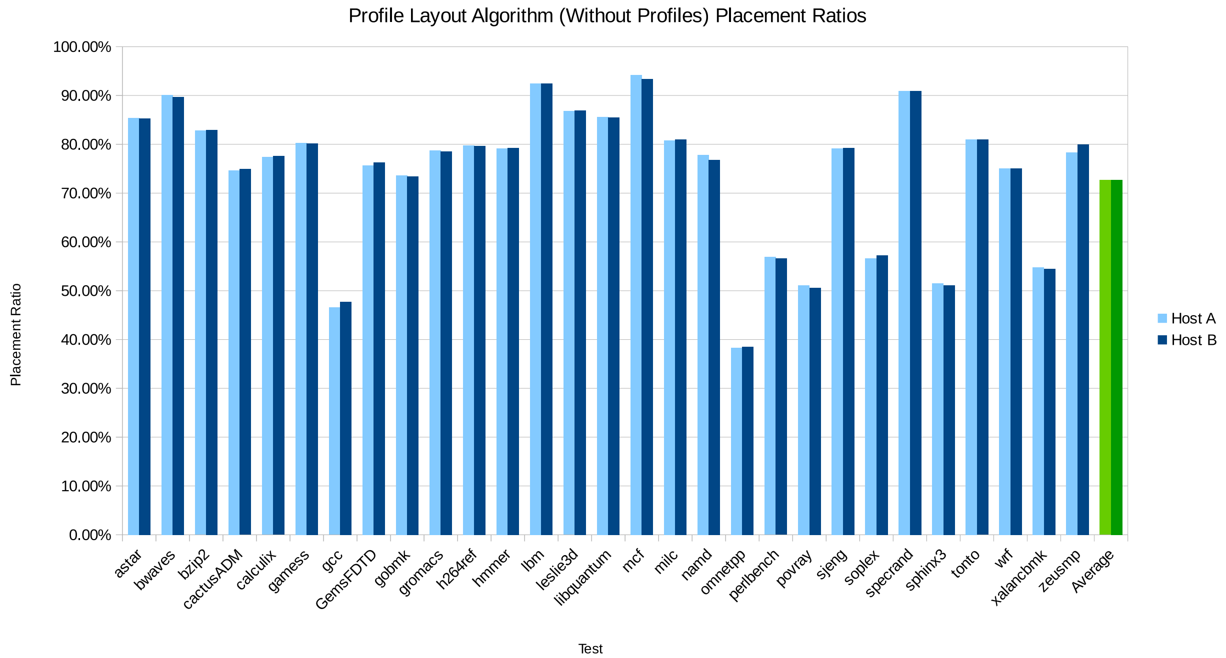

Figure 8.19: The percentage of the time target dollop could be placed on the winning bidder's preferred page. Host A's placement ratios are shown in light blue. Host B's placement ratios are shown in dark blue.

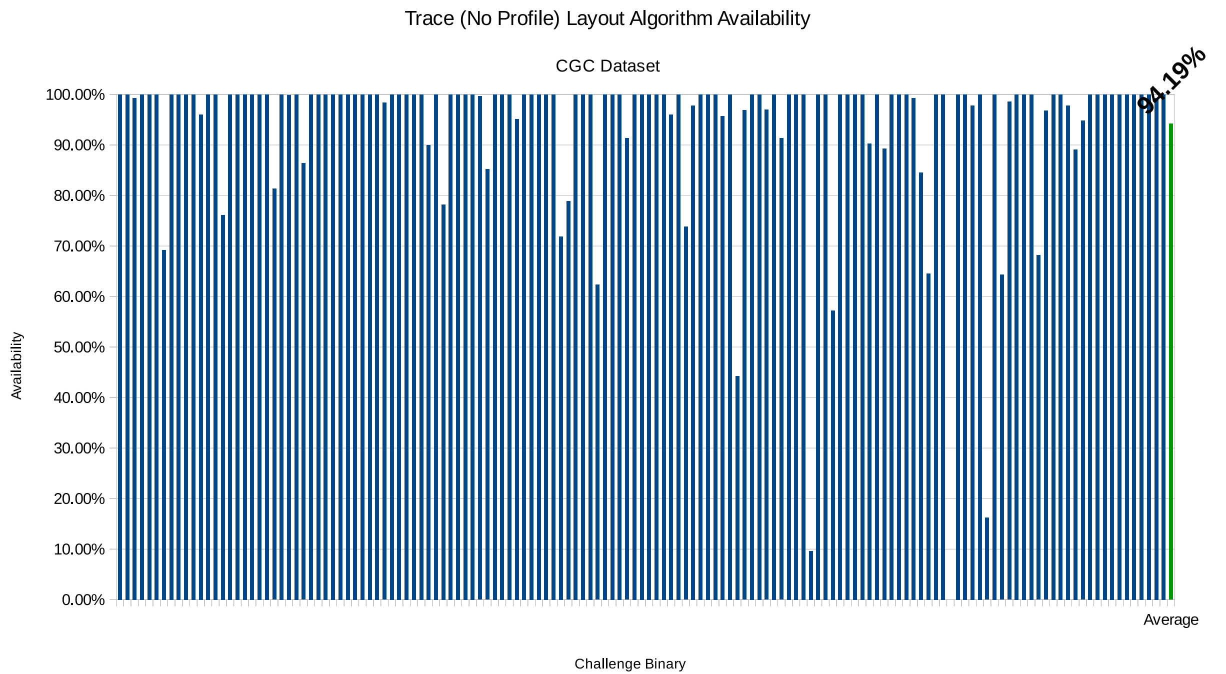

Figure 8.20: Availability scores for the RCBs in the CGC dataset when rewritten using the Profile Layout algorithm.

percentage of hot code can now be placed on the fewest possible pages. The benefit is a reduction in working set size and a reduction in the cache contention for the most common workloads.

Their experimental results show that their hypotheses were correct. Improvements in the runtime performance of their test applications ranged up to 26%. While part of the author's motivation was to reduce the working set size and the cache penalties, they do not measure these characteristics directly. They use overall runtime user performance improvement as a proxy for these values.

8.7 Conclusion

The Profile Layout algorithm is an optimized reassembly algorithm built on the insight gained by implementing, testing and analyzing the Locality Layout algorithm (see Chapter 7). As with the Locality Layout algorithm, the goal of the Profile Layout algorithm is to improve runtime performance with respect to CPU utilization and memory overhead.

The Profile Layout algorithm was originally developed to use profiles to guide program reassembly. Gathering representative execution profiles for real-world programs is difficult and time-consuming. The results indicate, however, that the Profile Layout algorithm produces statically rewritten programs/libraries without actual profiles that are as efficient as those produced with profiles. This removes a significant hurdle that users must cross before using this optimized reassembly algorithm.

The results indicate that the Profile Layout algorithm produces statically rewritten programs/libraries that are more efficient than those produced with the Locality Layout algorithm and the default algorithms of the Reassembly phase. The Profile Layout algorithm improves performance without sacrificing memory overhead or on-disk overhead and, for the interactive programs of the DARPA CGC dataset, improves those performance characteristics as well. The final reassembly optimization algorithm, the Relax Layout algorithm (see Chapter 9, extends the algorithms described in this section to produce statically rewritten programs/libraries that are as efficient and exhibit much less on-disk overhead.

Chapter 9

Relax Layout

9.1 Introduction

Thanks to the optimizations described in the Chapters 7, and 8, highly interconnected dollops are placed near one another in the final output binary. This characteristic makes it possible to implement a further optimization whose goal is to reduce the on-disk and in-memory size of the statically rewritten binary.

Recall from Section 4.4.4, that every link in the statically rewritten program is converted to an unconstrained link. Converting every link to an unconstrained link prior to placing dollops in the output binary means that there are no restrictions on the relative location of two interconnected dollops. This makes the implementation of the algorithms of the Reassembly phase easier but may result in wasted space.

When dollop a targets dollop b, the link between the two dollops must be unconstrained if those dollops are not close together.[1] However, when those dollops are placed near one another in the statically rewritten binary, using an unconstrained link is unnecessary. Let w be the difference in size of the architecture's implementation of a constrained link and the size of the architecture's implementation of an unconstrained link. On architectures with fixed width instructions, w will be 0. And, even on platforms where w is not 0, w is likely to be very small. Even though each instance of excess space is small, this difference is multiplied by the number of occurrences of this scenario throughout the statically rewritten binary and becomes significant. Let W be the total amount of wasted space for a statically rewritten binary program.

Minimizing W for a statically rewritten binary has two benefits. First, it reduces the on-disk size of the statically rewritten binary. This is important for systems where storage space is limited (e.g., embedded devices). Second, there will be more space on each page of the statically rewritten binary program to place

[1] Exactly how close depends on the target platform and the semantics of the instruction used to implement the link.

interconnected dollops. The benefits of colocated interconnected dollops on the same page were presented in the previous two chapters.

9.2 Savings

It is possible to calculate the exact value of W for every statically rewritten binary program. Assume that there are l links in the statically rewritten program. They are labeled L_1 through L_r. There is a function $far()$ that takes a link as a parameter and returns 1 if the target of that link is near the source of the link in the statically rewritten program; $far()$ returns 0 if that link is far from the source of the link in the statically rewritten program. Because $far()$ depends on the details of how a platform implements the link, what constitutes *near* and *far* depends on the architecture. In fact, on certain architectures like ARM, there may be no difference in size at all. In this case, $far()$ would always return 0 and $near()$ would always return 1. w is the difference between the size of unconstrained and constrained link. The total amount of wasted space from needlessly unconstrained links is

$$W = \sum_{i=1}^{l} far(L_i) * w$$

9.3 Algorithm

As mentioned at the beginning of this chapter, this optimization began as an enhancement of the Profile Layout algorithm whose benefits accrue from the characteristic of that algorithm to place interconnected dollops nearby in the statically rewritten program.

Unfortunately there is a technical hurdle that makes the Relax enhancement more than a straightforward extension to the Profile Layout algorithm. Recall that for the Profile Layout algorithm, when it acts on a bid of dollop a for dollop b, the algorithm attempts to place dollop b on the same page p on which dollop a was placed. If that is not possible, then dollop b is placed elsewhere. See Section 8.3.3 for the details. The hidden assumption is that dollop sizes are static throughout placement. If the Relax enhancement were a simple extension of that algorithm, however, the enhancement would not be as effective as it could otherwise be because the Relax enhancement invalidates the invariant of dollops' static sizes.

Consider the three dollops a, b and c and the two placements illustrated in Figure 9.1. Dollop a has two links to dollop b. By default those are unconstrained links and are shown that way on the left side of the illustration. Assume that the Profile Layout algorithm has placed dollops a and b on the same page p;

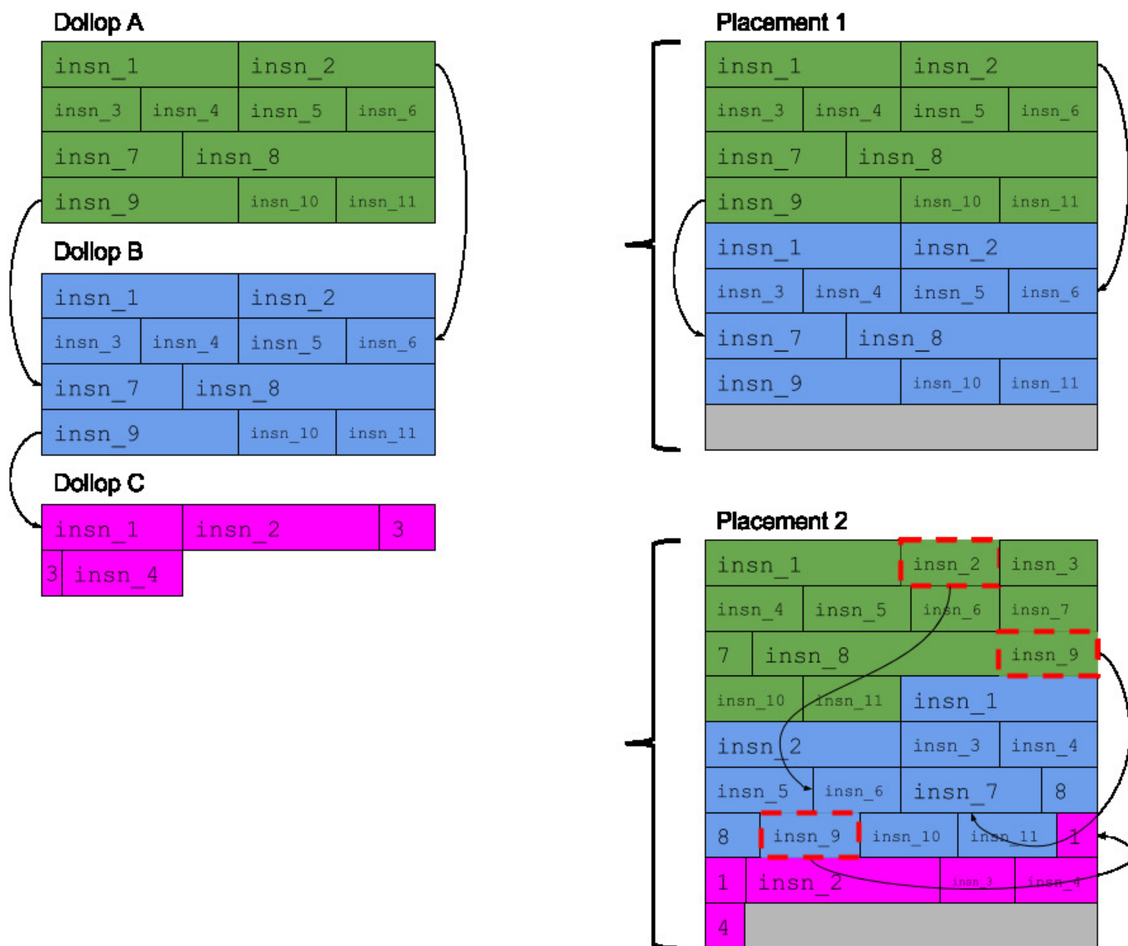

Figure 9.1: An example showing why the Relax Layout algorithm is not a simple extension of the Profile Layout optimization. Because the Relax Layout algorithm potentially changes the size of the implementation of links, the hidden assumption of the other layout algorithms that the dollop size is constant is violated.

Placement 1 shows this. The next highest bid inspected by the Profile Layout algorithm is dollop b's bid for dollop c. Unfortunately, there is not enough room on page p for dollop c so it is placed elsewhere.

The goal of the Relax Layout algorithm is to use constrained links wherever possible. Because dollops a and b are placed nearby in the statically rewritten binary, dollop a's links can be constrained. By actualizing dollops a and b according to dollop a's new size, dollop c can now fit on page p. More importantly, because dollop c is placed near dollop b, dollop b's link to dollop c can also be constrained. This is illustrated in Placement 2. Notice, especially, that whereas in Placement 1, dollop c could not fit, in Placement 2 dollop c fits *and* there is still space remaining on page p.

In other words, the Relax Layout algorithm assumes that dollop sizes are dynamic and their placements must shift throughout the entire course of the layout of the statically rewritten binary in order to make the

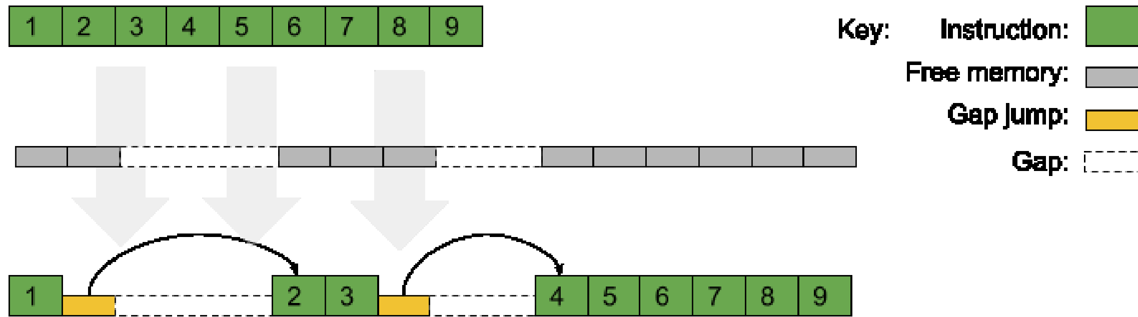

Figure 9.2: The Relax Layout algorithm *drapes* the dollops over the memory space of the statically rewritten binary and accommodates its *gaps*.

most effective use of space. As a result, the Relax Layout algorithm must work with a modified version of the Profile Layout algorithm for peak effectiveness.

9.3.1 The Modified Profile Layout Algorithm

The modified Profile Layout algorithm builds an ordered list of dollops according to their interconnectedness and considers them to be a single, contiguous list of dollops. Once that list is built through the auction process, the algorithm *drapes* the list over the memory space of the statically rewritten binary program. As explained in Section 4.4.1, the memory space of the statically rewritten binary program is never entirely free. There are *gaps* where instructions cannot be placed. In the process of draping the dollops over the memory space, the sequence is split where there are gaps and program control instructions are inserted to jump over those gaps. See Figure 9.2.

The fundamentals of the Modified Version of the Profile Layout algorithm still rely on the same auction mechanism to determine placement as the original version of the Profile Layout algorithm and, like the original Profile Layout algorithm, the modified Profile Layout algorithm operates in phases. Recall that in the original version of the algorithm there is a bootstrap phase and a placement phase. See Section 8.3.3.

The modified Profile Layout algorithm also has a bootstrap and placement phase in addition to a third phase, the *layout* phase. The original and modified versions of the Profile Layout algorithm behave identically in the bootstrap phase.

The two versions differ in the placement phase. See Algorithm 6 for the pseudocode for the modified version of the placement algorithm. Whereas the original Profile Layout algorithm makes deterministic placement of dollops on particular fixed-size pages, the modified Profile Layout algorithm makes dollop placements on particular infinitely-sized pages. *Infinite pages* allow the modified version of the algorithm to

make the ideal placements of dollops on pages. Unlike the original version of the algorithm where a target dollop may not fit on the same page as the bidding dollop because there is not enough space, infinite pages never run out of space.

Algorithm 6 The updated version of the AwardBids function for the placement phase of the Relax Layout algorithm.

```
1: function AwardBids(BidList bl)
2:     pl ← ∅
3:     while b ∈ bid_list do
4:         awarded_page ← null
5:         if not IsPinnedBid(b) then
6:             awarded_page ← PageOf(b.PinnedAddress)
7:         else if IsPreplaced(pl, b.Bidder) then
8:             awarded_page ← PageOf(PrePlacement(pl, b.Bidder))
9:         end if
10:        if awarded_page not null then
11:            success ← PrePlaceOnInfinitePage(pl, b.Target, awarded_page)
12:            DeleteBid(b)
13:        end if
14:    end while
15:    return pl
16: end function
```

The product of the placement phase of the modified Profile Layout algorithm is a list of pages and a list of dollops that should be placed on those pages (*pl* in AwardBids in Algorithm 6 is a map data structure used to hold this mapping). Within each infinite page, the dollops are ordered accorded to their interconnectedness.

The runtimes of the modified version of the Profile Layout algorithm used in the Relax Layout algorithm to build the list of infinite pages holding dollops are the same (Section 8.3.3 contains a complete description of the runtime of the Profile Layout algorithm).

Before beginning its final phase, the modified version of the algorithm calculates the gaps in the memory space of the statically rewritten binary program. See Function CreateGaps in Algorithm 7. Gaps are places in the memory space of the statically rewritten binary where dollops cannot be placed. The initial set of gaps is comprised of the original program/library's pinned addresses.[2] Gaps prevent dollops from being placed where a pinned address exists. See Section 4.4.1.

After creating the initial set of gaps, the algorithm *coalesces* those gaps where necessary. See Function CoalesceGaps in Algorithm 7. Two gaps a and b are coalesced into a single gap if there is not enough free space between them to place an instruction that transfers control over b. When this occurs, the two gaps are coalesced into a single gap c.

Algorithm 7 The algorithms to create and coalesce gaps.

```
1: function CreateGaps(PinnedAddressList pal)
```

[2] Unmapped address space between two executable sections of code in the original program/library can also be considered gaps. The algorithm presented here does not take this into account but this situation is handled by the prototype implementation.

```
2:      gap_list ← ∅
3:      for all p ∈ pal do
4:          gap ← ∅
5:          gap.StartAddress ← p.Address
6:          gap.EndAddress ← gap.StartAddress + p.Length
7:          Insert(gap_list, gap)
8:      end for
9:      return gap_list
10: end function

11: function CoalesceGaps(GapList gl)
12:     new_gap_list ← ∅
13:     current_gap ← Null
14:     previous_gap ← Null
15:     minimum_gap_space ← SizeOfShortestJump
16:     previous_gap_jump_space ← 0
17:     for all g ∈ gl do
18:         if not current_gap then
19:             current_gap ← g
20:             continue
21:         end if
22:         if SpaceBetween(current_gap, g) ≤ minimum_gap_space then
23:             current_gap ← Combine(current_gap, g)
24:             while current_gap.Start + ReachableLength(previous_gap_jump_space) ≤
                    current_gap.EndAddress do
25:                 Pop(new_gap_list)
26:                 current_gap ← Combine(previous_gap, current_gap)
27:                 previous_gap ← Top(new_gap_list)
28:                 previous_gap_jump_space ← SpaceBetween(previous_gap, current_gap)
29:             end while
30:         else
31:             Insert(new_gap_list, current_gap)
32:             current_gap ← g
33:             previous_gap ← Top(new_gap_list)
34:             previous_gap_jump_space ← SpaceBetween(previous_gap, current_gap)
35:         end if
36:     end for
37:     return new_gap_list
38: end function
```

In the example shown in Figure 9.3, assume that the algorithm is statically rewriting a binary for the x86 platform where the smallest instruction that can be used to implement jumps over gaps is two bytes ($SizeOfShortestJump$). However, there is only a single byte of space between the two gaps. Therefore, no program control instruction can fit between those two gaps and the two gaps are coalesced into a single gap, c.

Unless gap a started at the very beginning of the original program, by necessity there was a program control instruction i that implemented the jump over gap a. Now that a is subsumed into gap c which is, by necessity, bigger than a, i may need to be implemented using a different program control instruction i' that is bigger than i. If there is not enough space to put i' before a, a further coalescing is necessary. See Lines 24

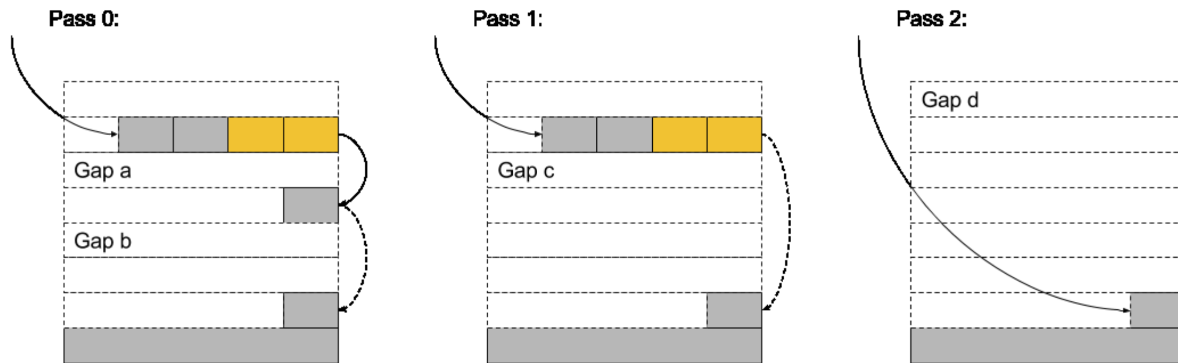

Figure 9.3: Coalescing gaps can have a cascading effect on the size of the program control instructions needed to jump over them.

through 29 in Algorithm 7.

Continuing the example in Figure 9.3, there are only four available bytes before *c*. However, the two bytes allocated for the program control instruction that implemented the jump over *a* are not enough to accommodate a larger instruction needed to implement the program control to jump over *c*. Therefore, gap *c* is coalesced with the preceding gap to form gap *d*.

The process continues until the list of gaps and the space between them required to hold program control instructions to implement the jumps over gaps stabilizes.

In future work, it may be possible to incorporate a heuristic into the process of determining whether to coalesce gaps. It is likely that, although an instruction can fit into a gap, using that gap could negatively impact the performance of the statically rewritten program/library. Transferring program control to a place where only two or three instructions are executed before another program control transfer is executed will put pressure on the branch prediction cache and the execution of those transfers will slow program execution. In cases like this, trading the space overhead inherent in leaving empty gaps for performance might be advantageous.

The final phase of the modified Profile Layout algorithm is layout (LayoutProgram in Algorithm 8). There is no such phase in the original version of the algorithm. LayoutAlgorithm in the layout phase operates on the results of previous phases: the program's set of infinite pages (*pl*) and the program's gaps (*gl*).

Before program layout *per se* begins, several supporting data structures must be initialized. First, the layout phase takes the program's list of infinite pages and merges them into one ordered list (see PopulateProgram in Algorithm 8). Second, the layout phase initializes a data structure that maps from instructions to their sizes (see InitializeSizes in Algorithm 8). By default, all link are assigned

the size of the smallest program control structure than can implement the link on the target architecture ($SizeOfShortestJump$). The choice to optimistically size every link initially as the platform's smallest instruction possible to implement a link means that, upon termination, the algorithm will have created a program layout that uses the least amount of space necessary to implement the program's links. If the instruction is not a link, its size is calculated using a helper function provided by the static binary rewriter itself (SizeOf).

Algorithm 8 The algorithms to perform the size-efficient placement of dollops around the gaps in the memory space of the statically rewritten program/library.

```
1: function PopulateProgram(PageList pl)
2:     program ← ∅
3:     for all p ∈ pl do
4:         for all d ∈ p.Dollops do
5:             Push(program, d)
6:         end for
7:     end for
8:     return program
9: end function

10: function InitializeSizes(Program p)
11:     insn_sizes ← ∅∅
12:     for all d ∈ p do
13:         for all i ∈ d do
14:             if not IsJump(i) then
15:                 instruction_sizes[i] ← SizeOf(i)
16:             else
17:                 instruction_sizes[i] ← SizeOfShortestJump
18:             end if
19:         end for
20:     end for
21:     return insn_sizes
22: end function

23: function Drape(Program p, GapList gl, InstructionSizes is)
24:     instruction_addresses ← ∅∅
25:     current_address ← p.StartAddress
26:     for all d ∈ p do
27:         for all i ∈ d do
28:             instruction_address[i] ← AddressNextInstruction(current_address, is[i], gl)
29:             current_address ← current_address + is[i]
30:         end for
31:     end for
32:     return instruction_addresses
33: end function

34: function Relax(Program p, InstructionAddresses ia, InstructionSizes is)
35:     changed ← false
36:     for all d ∈ p do
37:         for all i ∈ d do
38:             if IsJump(i) then
```

```
39:             if not TargetIsReachable(ia[i], is[i], Target(i)) then
40:                 is[i] ← SizeOfLargestJump
41:                 changed ← true
42:             end if
43:         end if
44:     end for
45:   end for
46:   return changed
47: end function

48: function LayoutProgram(PageList pl, GapList gl)
49:   program ← ∅
50:   program_layout ← ∅∅
51:   insn_sizes ← ∅∅
52:   insn_addresses ← ∅∅

53:   program ← PopulateProgram(pl)
54:   insn_sizes ← InitializeSizes(program)

55:   insn_addresses ← Drape(program, gl, insn_sizes)
56:   while Relax(program, insn_addresses, insn_sizes) do
57:       insn_addresses ← Drape(program, gl, insn_sizes)
58:   end while
59:   return insn_addresses
60: end function

61: function AddressDollop(Dollop d)
62:   return memorialized_insn_addresses[FirstInstruction(d)]
63: end function
```

Program layout itself is performed by DRAPE. DRAPE starts from the beginning of the statically rewritten program's memory space (*current_address*) and drapes the program's instructions over the gaps. Draping is done one instruction at a time by taking into account by calling ADDRESSNEXTINSTRUCTION that takes into account the current address in the memory space of the statically rewritten program/library (*current_address*), the instruction's size ($is[i]$) and the program's gaps (gl). The addresses assigned to each instruction are stored in a map data structure (*instruction_addresses*) that is returned by DRAPE.

The layout of instructions in addresses as held by the output from LAYOUTPROGRAM is tentative. It is not a final layout because it is possible that the sizes allocated to one or more of the link instructions is not big enough to hold a program control instruction that can reach its target. RELAX verifies the proposed mapping generated by DRAPE and corrects mistakes if necessary. RELAX uses TARGETISREACHABLE to determine whether instruction i's target (TARGET()) is reachable from i's tentative address ($ia[i]$) using a program control instruction of the tentative size ($is[i]$). The corrections are made directly to the instruction-to-size mapping and a boolean value is returned that indicates whether changes were made.

If the size of any link was changed during the relaxation process, draping is done again. Subsequent iterations of DRAPE use the updated instruction-to-size mapping created by RELAX. Once there is no longer

the need to change the size of any links, the program layout is stable and the instruction-to-address mapping is *memorialized*.

When the default layout algorithms of the Reassembly phase invoke the Relax Layout algorithm's AddressDollop function to inquire about preferential addresses for a dollop, the memorialized instruction-to-address mapping is consulted to find the address for the first instruction in that dollop. This address is returned and reassembly proceeds.

9.4 Evaluation

9.4.1 Performance

Figures 9.4 and 9.5 show the difference in performance for each of the applications of the SPEC benchmark suite when those programs are rewritten using the Relax Layout algorithm as opposed to the *Profile Layout algorithm* described in Chapter 8.

At the conclusion of the presentation of the results of the evaluation, there is a discussion about the cause for the surprising results.

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Relax Layout algorithm on Host A was 1.289%. This compares unfavorably (a 5.621% impairment) to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm (1.221%). It is comparable (a 0.09% impairment) to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten using the Locality Layout algorithm (1.288%) that was an improvement from the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten using the default algorithms of the Reassembly phase (1.323%).

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Relax Layout algorithm on Host B was 1.252%. This compares unfavorably (a 4.487% impairment) to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm (1.198%). It is comparable (a 0.122% improvement), however, to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten using the Locality Layout algorithm (1.253%) that was already an improvement from the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten using the default algorithms of the Reassembly phase (1.285%).

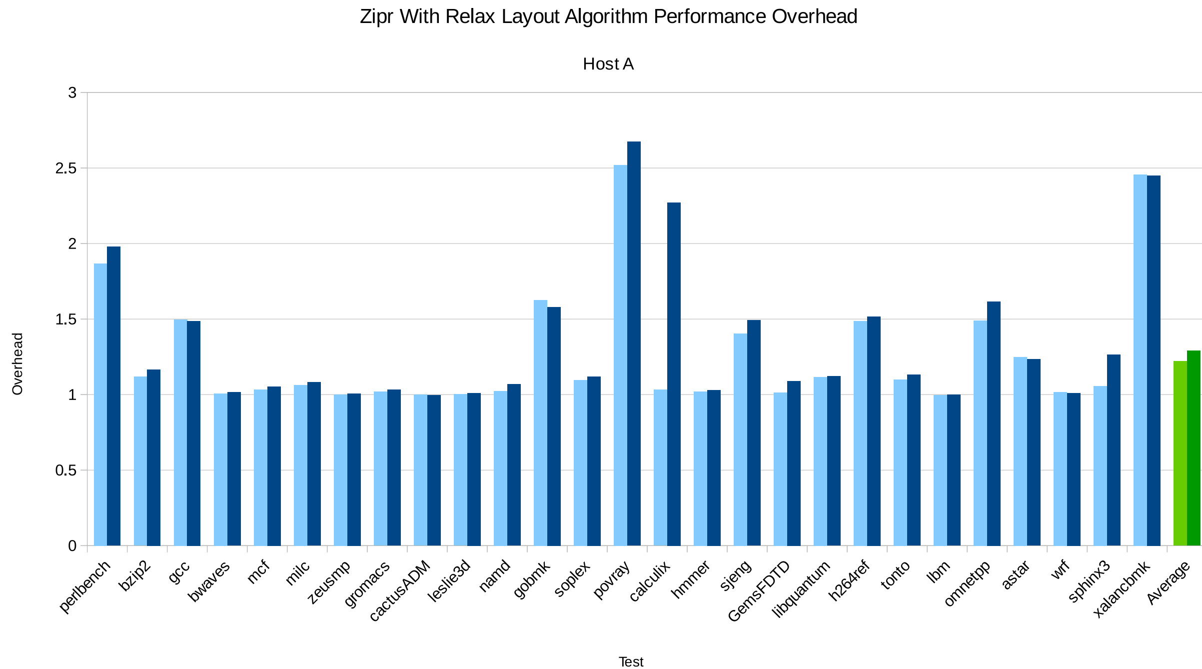

Figure 9.4: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

9.4.2 Memory Usage

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Relax Layout algorithm on Host A was 1.009. This is roughly equivalent to the average RSS overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm (1.008x). It is within less than a tenth of a percent.

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Relax Layout algorithm on Host B was 1.007x. This is roughly equivalent to the average RSS overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the Profile Layout algorithm (1.011x). It is within half of a percent.

The Relax Layout algorithm was the only optimized reassembly algorithm that significantly improved the page fault overhead for statically rewritten applications of the SPEC benchmark suite.

Figures 9.8, and 9.9 show the page fault overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. On Host A, the overhead ranged from 1.220% to 222.778%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.221x more page faults than

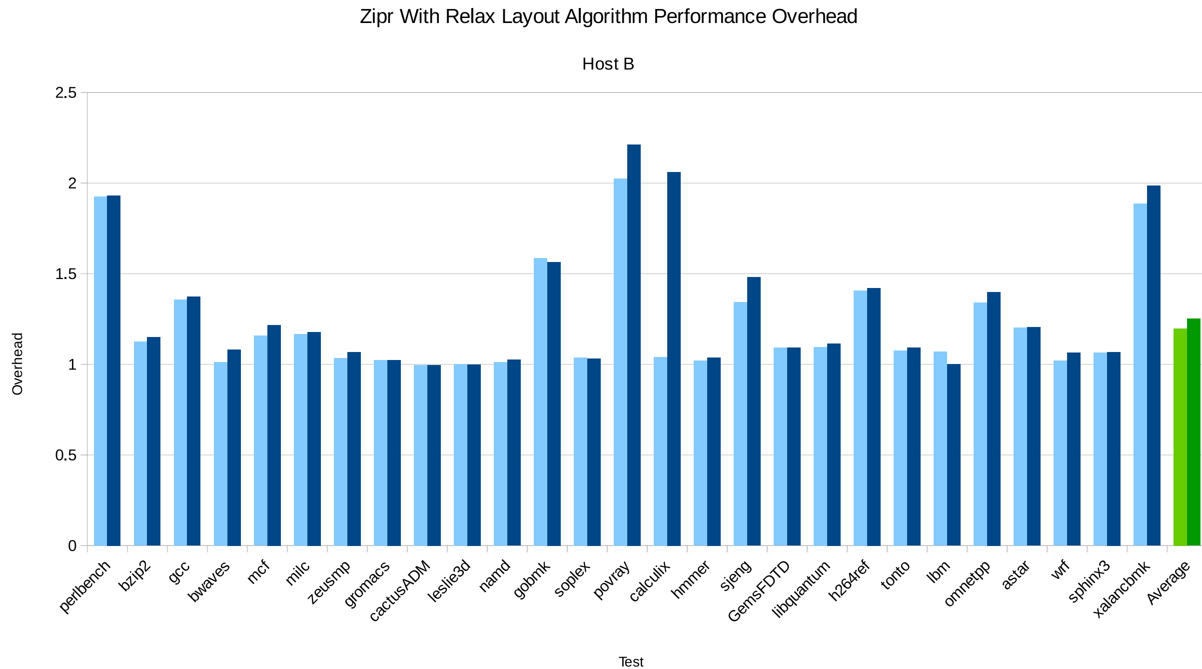

Figure 9.5: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

the native versions. That compares favorably (a 5.90% improvement) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using the Profile Layout algorithm (1.293x). On Host B, the overhead ranged from as little as 1.136% to 217.028%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.215x more page faults than the native versions on Host B. That compares favorably (a 5.68% improvement) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using Profile Layout algorithm (1.284x).

9.4.3 Instruction Cache Usage

As discussed in Section 5.5.3, the instruction cache is an important part of a host's hardware designed to improve performance. Using an algorithm like the Relax Layout algorithm that reassembles statically rewritten programs and libraries to take advantage of the combination of temporal locality and smaller outputs will improve the statically rewritten program's use of the instruction cache.

Figures 9.10 and 9.11 show results for evaluation of the instruction cache usage for Hosts A and B, respectively. For Hosts A and B, the versions of the programs of the SPEC benchmark suite statically

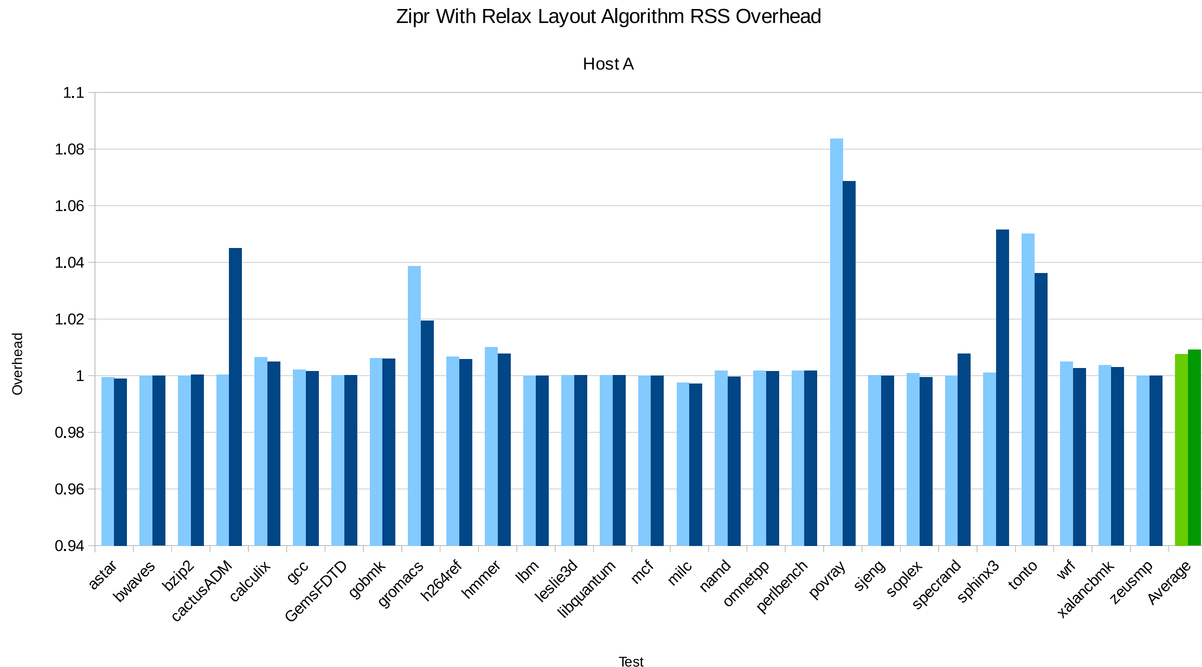

Figure 9.6: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

rewritten using the Relax algorithm incur 3.41x and 3.87x as many instruction cache misses as their native counterparts, respectively. Those overheads are improvements of more than 5% both Hosts A and B when compared to the instruction cache miss overhead of the programs of the SPEC benchmark suite statically rewritten using the Profile Layout algorithm which itself was already an improvement over the Locality Layout algorithm and the default algorithms of the Reassembly phase. The improvements to the instruction cache usage relative to the Profile Layout algorithm are not nearly as great as those of the Profile Layout algorithm relative to the Locality Layout algorithm. The relative lack of improvement in this metric seems to play a role in the overall decrease in performance associated with the statically rewritten versions of the applications of the SPEC benchmark suite with the Relax Layout algorithm. See Section 9.4.7 for additional discussion.

9.4.4 Filesize Overhead

The Relax Layout algorithm is the only optimized reassembly algorithm that produces a significant improvement in the filesize overhead with respect to the SPEC benchmark suite.

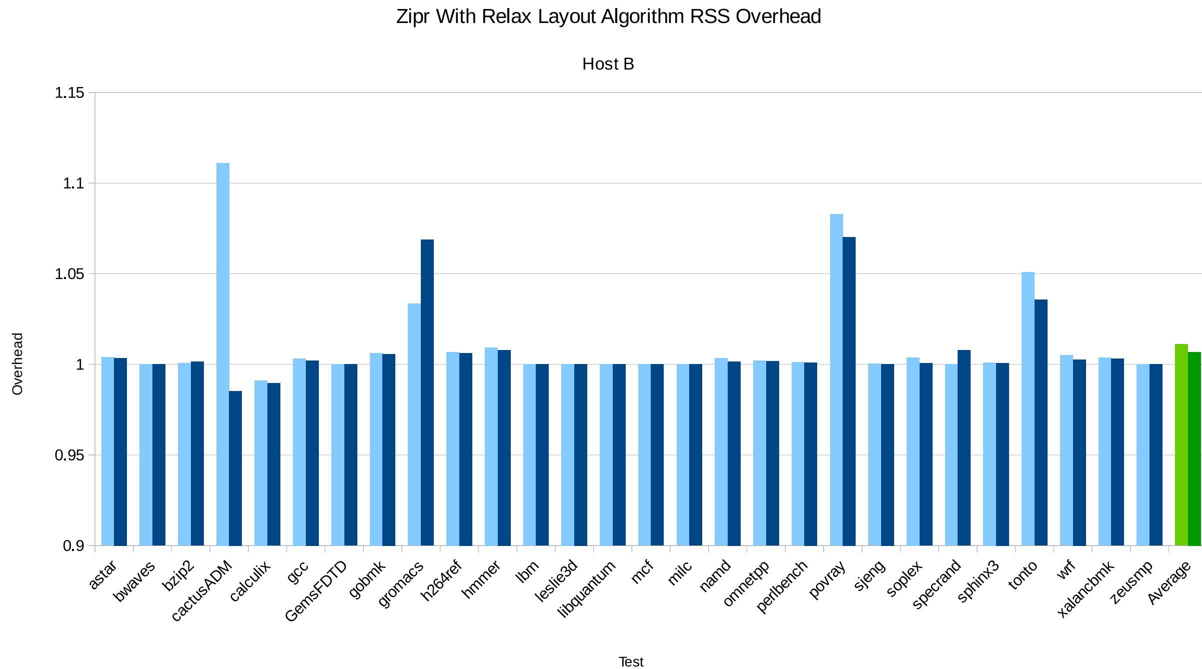

Figure 9.7: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

Figures 9.12, and 9.13 show the filesize overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. For Host A, the average filesize overhead of the applications of the SPEC benchmark suite was 6.436%. For Host B, the overhead was 6.448%. In both cases, the filesize overhead is minimal which again validates the ability of the architecture and algorithms of the static binary rewriter as implemented by Zipr to produce statically rewritten programs with low on-disk overhead [216, 11]. The filesize overhead using the Relax layout algorithm on Host A is a 2.785%, 2.058% and 2.890% improvement over the default, Locality and Profile layout algorithms, respectively. For Host B, the filesize overhead using the Relax layout algorithm on Host A is a 2.447%, 2.584% and 2.946% improvement over the default, Locality and Profile layout algorithms, respectively.

9.4.5 Relax Ratios

Through the introduction of the Relax Layout algorithm, all links in a statically rewritten program/library were unconstrained.[3] As described in Section 9.3, the goal of the Relax Layout algorithm is to use constrained

[3]Technically, chained links use constrained links (see Section 4.4.3). Since those are only used as a way to work around a lack-of-space required to place an unconstrained link, the statement above is satisfactory.

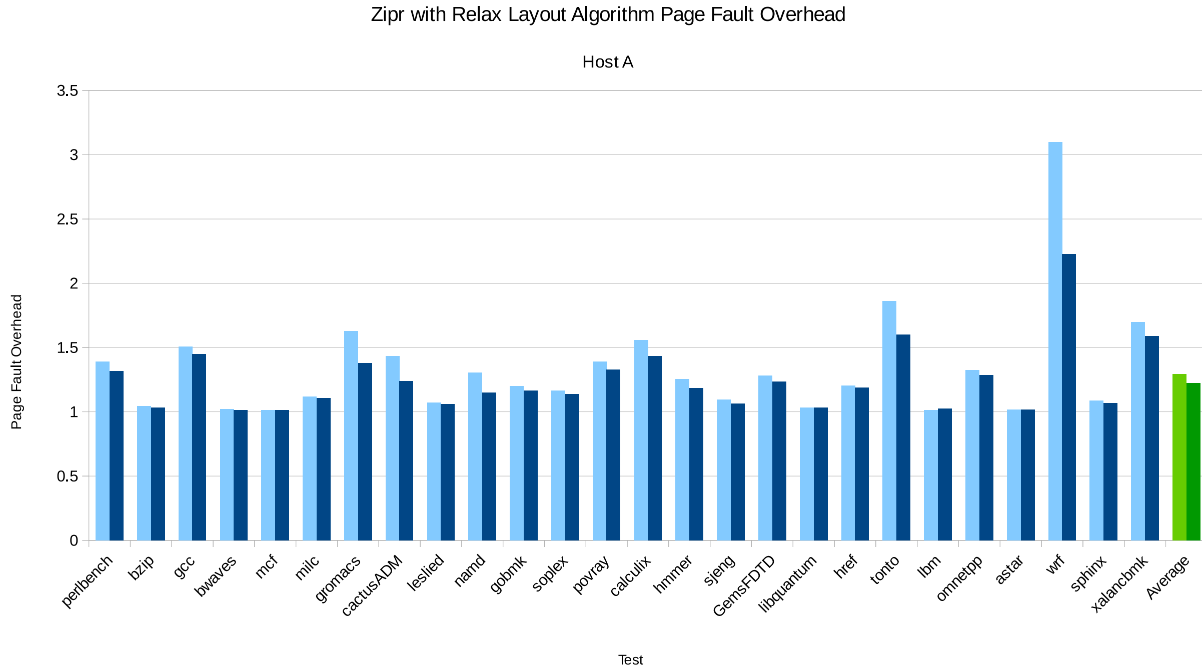

Figure 9.8: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

links as often as possible. Not every link can be constrained, however. No matter that the auction process attempts to place the target dollops as close to the links as possible, there will be cases where the dollops are still too remote for a constrained link. The results in Figure 9.14 show the ratio of constrained and unconstrained links used when the applications of the SPEC2006 benchmark suite are rewritten using the Relax Layout algorithm. On Hosts A and B, the relaxation ratios were 60.89% and 60.43%, respectively. As with the placement ratios described in Sections A.5 and 8.5.5, the relaxation ratios depend entirely on the binary that is being rewritten. This confirms the hypothesis that because the relaxation ratios depend on the compiler and libraries used to build input program/library, which are identical on Hosts A and B, the relaxation ratios will be similar on Hosts A and B.

9.4.6 CGC

The more than 140 CBs in the CGC dataset provide another way to assess whether the Profile Layout algorithm achieves its stated goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs. See Section 5.6 for a complete discussion of

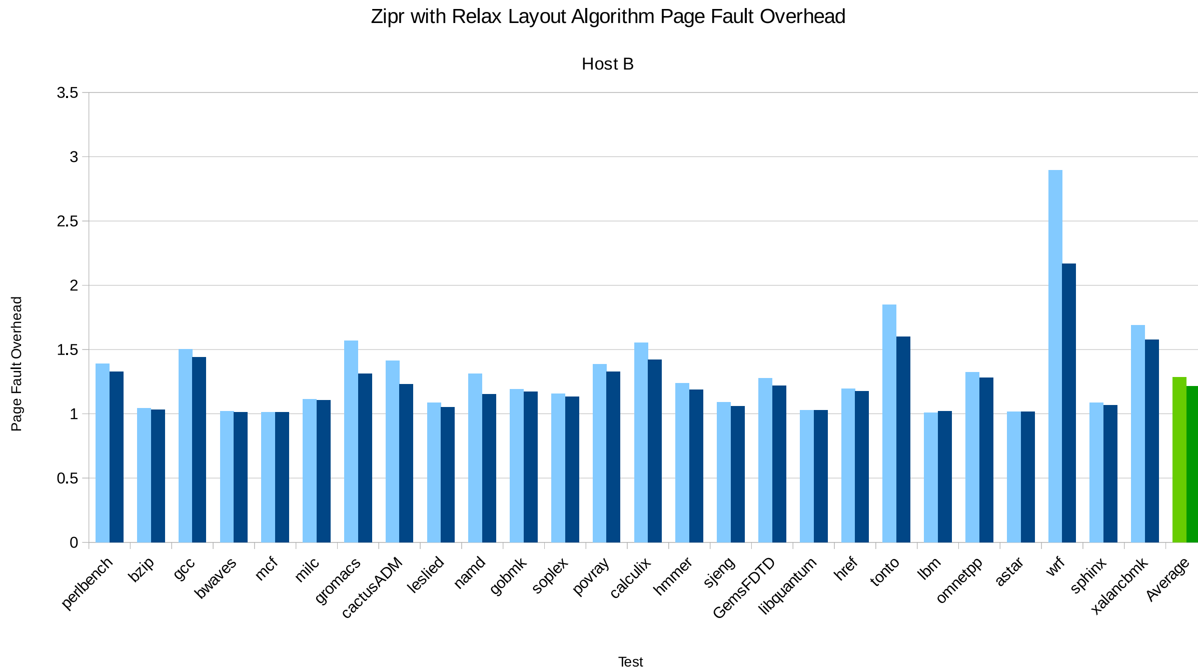

Figure 9.9: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

Category	Default	Locality	Profile	Relax
None	42	54	100	103
Performance	16	12	12	16
Filesize	1	2	0	0
Memory	83	74	29	23
Functionality	1	1	2	1

Table 9.1: Distribution of dominators for the RCBs in the CGC dataset generated using the Relax Layout algorithm.

the CGC dataset and the terms used in the description of the evaluation.

Figure 9.15 shows the availability scores for each of the RCBs generated by Zipr using the Relax Layout algorithm instead of the Profile Layout algorithm, the Locality Layout algorithm or the default algorithms of the Reassembly phase.[4] The average availability score was 95.41%. This compares well with the average availability score for the RCBs generated by Zipr using the default algorithms of the Reassembly phase (83.42%), the average availability score for the RCBs generated by Zipr using the Locality Layout algorithm (87.09%) and the average availability score for the RCBs generated by Zipr using the Profile Layout algorithm (94.19%) – a 13.41%, 9.12% and 1.29% increase, respectively.

[4]The Null Transformation was the only transformation applied to the CBs.

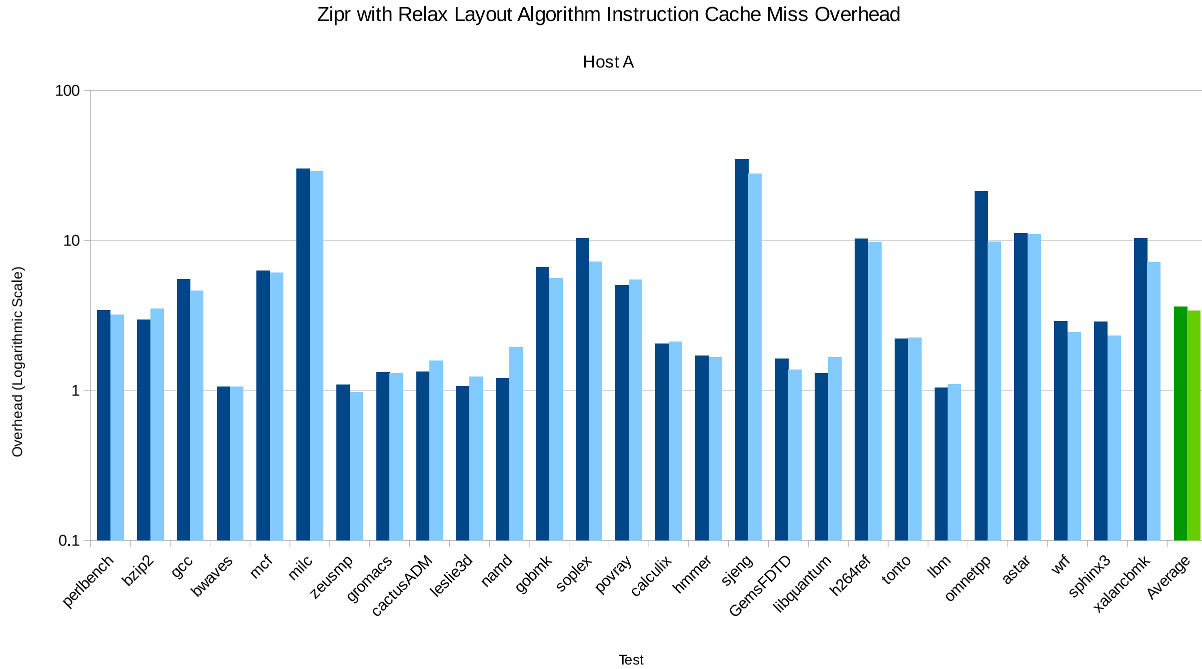

Figure 9.10: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host A. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm (without profiles) are shown on the left, in light blue and light green.

Table 9.1 shows the distribution of dominators for the RCBs in the CGC dataset generated using the Relax Layout algorithm. Overall there are 103 RCBs with a perfect availability score compared with only 42 RCBs with a perfect availability score when statically rewritten using the default algorithms of the Reassembly phase, 54 RCBs with a perfect availability score when statically rewritten using the Locality Layout algorithm and 100 RCBs with a perfect availability score when statically rewritten using the Profile Layout algorithm.

The distribution shows a notable decrease in the number of RCBs that have a lower availability score because of memory overhead. Of the RCBs that have less-than-perfect availability, more than 57% are most negatively impacted by their memory overhead. In absolute terms, there is an 113.21% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the default algorithms of the Reassembly phase, a 105.15% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the Locality Layout algorithm and a 23.08% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the Profile Layout algorithm.

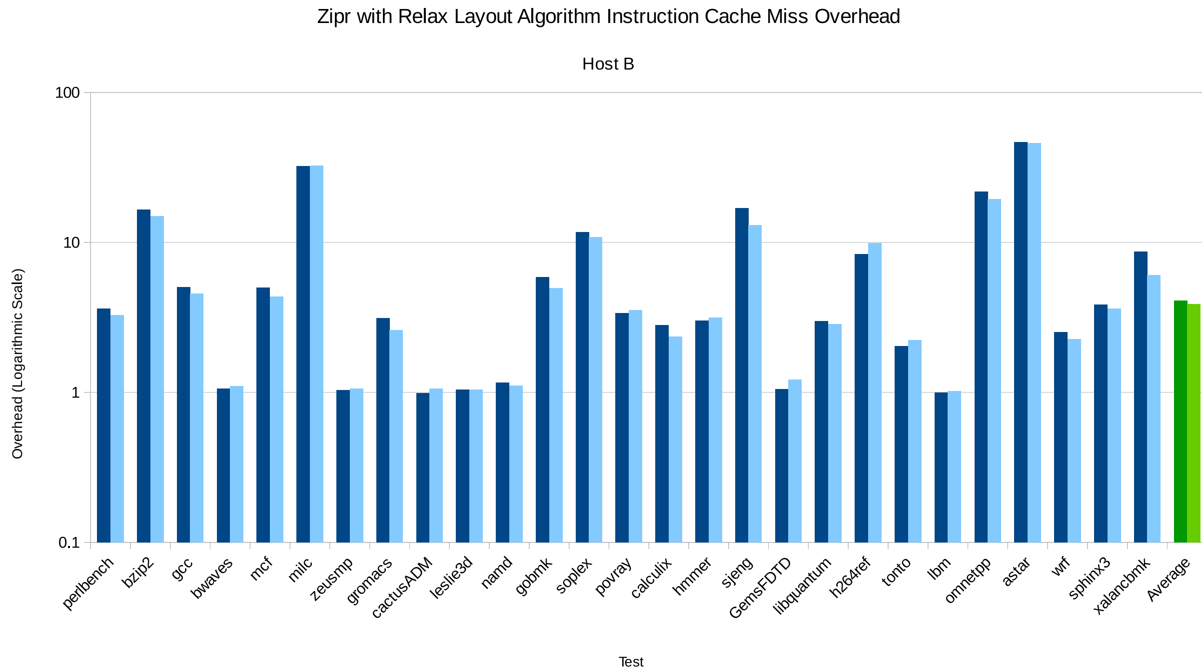

Figure 9.11: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Relax Layout algorithm. These results are for Host B. The results for the applications reassembled using the Profile Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the Profile Layout algorithm (without profiles) are shown on the left, in light blue and light green.

9.4.7 Discussion

The Relax Layout algorithm was developed as a compromised version of the Profile Layout algorithm intended to generate statically rewritten programs/libraries that had less on-disk and runtime memory overhead without sacrificing performance. The results show that with respect to the on-disk and runtime memory overhead, the Relax Layout algorithm performed as intended. Unfortunately, while the performance of the CBs of the DARPA CGC dataset statically rewritten using the Relax Layout algorithm improved slightly, the applications of the SPEC benchmark suite statically rewritten with the Relax Layout algorithm showed a decrease in overall performance.

The decrease in overall performance for the applications of the SPEC benchmark suite statically rewritten using the Relax layout algorithm is the result of the optimized reassembly algorithm's more lenient approach to dollop placement. In the Profile Layout algorithm upon which the algorithm for code placement in the Relax Layout algorithm is based, dollops are assigned to particular pages whose length matches the length of the pages of memory of the target platform. In the Relax Layout algorithm, dollops are assigned to hypothetical pages that are infinitely long. Infinitely sized pages allow for the ideal placement of dollops

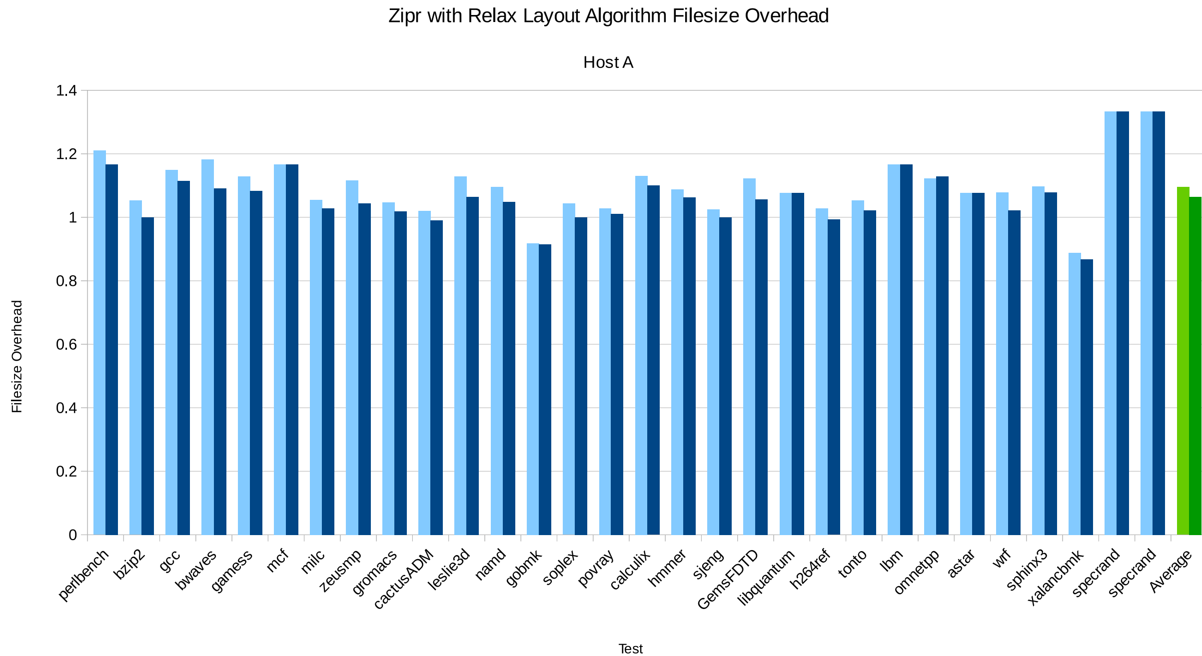

Figure 9.12: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

but problems arise when the list of pages and their dollops are merged into a single list of dollops. As the Relax Layout algorithm goes through the layout phase and the jump instructions implementing the links are relaxed, dollops may *slide* off the page where they should be placed to achieve the goal of creating the most efficient statically rewritten binary program/library.

Figure 9.16 shows an example of this problem. In the example program, there are four dollops – Green, Gray, Blue and Tan. The layout component of the Relax Layout algorithm determines that it is ideal for the Green and Gray dollops to be placed on Page A and the Blue and Tan dollops to be placed on Page B. Upon merging the infinitely sized pages and their dollops into a single list, the dollops *are* placed on the pages in a way that an efficient rewritten program/library will result. Unfortunately, those placements are possible only as long as the links between dollops (implemented here as `jmp` instructions) are the smallest size for the platform. Assume, however, that the link in the Green dollop cannot reach `X` when implemented with the smallest of the `jmp` instructions for the platform and the `jmp` must be relaxed. After relaxation, the layout of the dollops has shifted. In particular, the Tan dollop is now split between Page B and Page C. Were this the way that the program/library was memorialized, the execution of the Tan dollop would incur a performance penalty because it crosses a page boundary and likely crosses an instruction cache boundary,

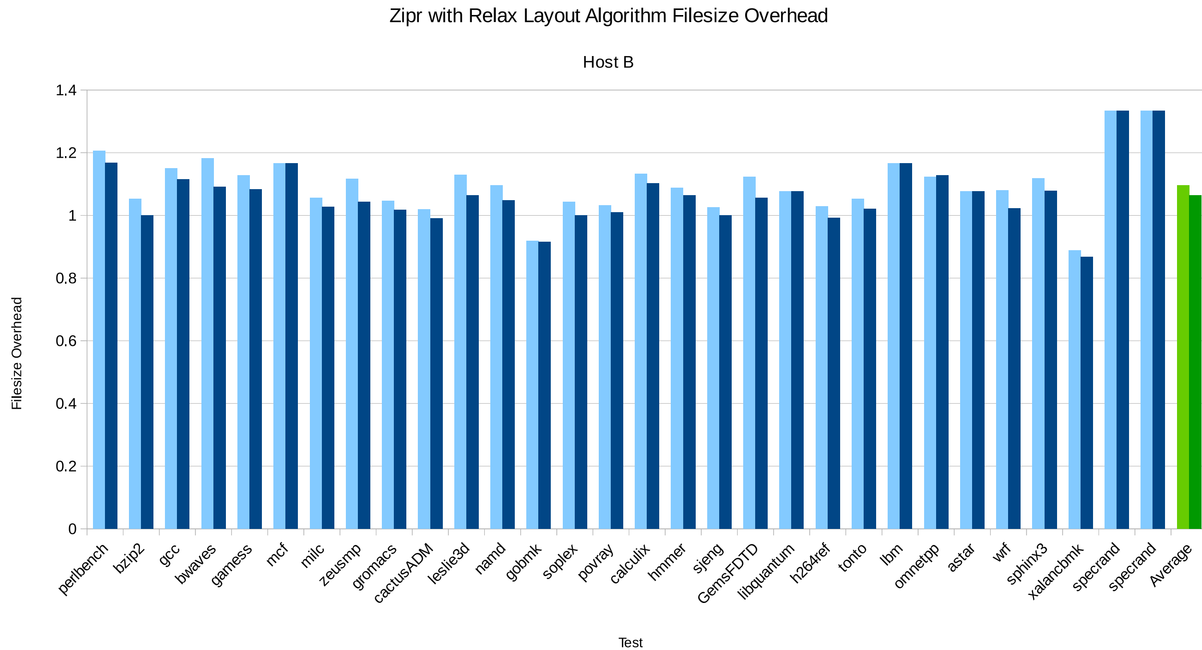

Figure 9.13: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Relax Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the Profile Layout algorithm are shown on the left, in light blue and light green.

too. This example shows how even a single relaxation can have a negative impact on performance of the statically rewritten program/library.

In the future it may be possible to better balance the tradeoff in the Relax Layout algorithm to combine the benefit of the strict code layout policy of the Profile Layout algorithm with the benefit of reduced on-disk and runtime memory overheads.

9.5 Related Work

The intuition that the static rewriter would produce more efficient and smaller rewritten binaries if it used the Relax Layout algorithm came from the LLVM Compiler Infrastructure [36]. LLVM, formerly known as the low-level virtual machine [125], is a cross platform, modular toolkit for building compilers [219]. Each of its modules can be composed to form a compiler for different source languages and different target architectures.

Besides modularity, LLVM is known for its code optimizations. Because one of its modules is an assembler and because it targets ISAs that have variable-sized `jmp` instructions, the designers of LLVM implemented an

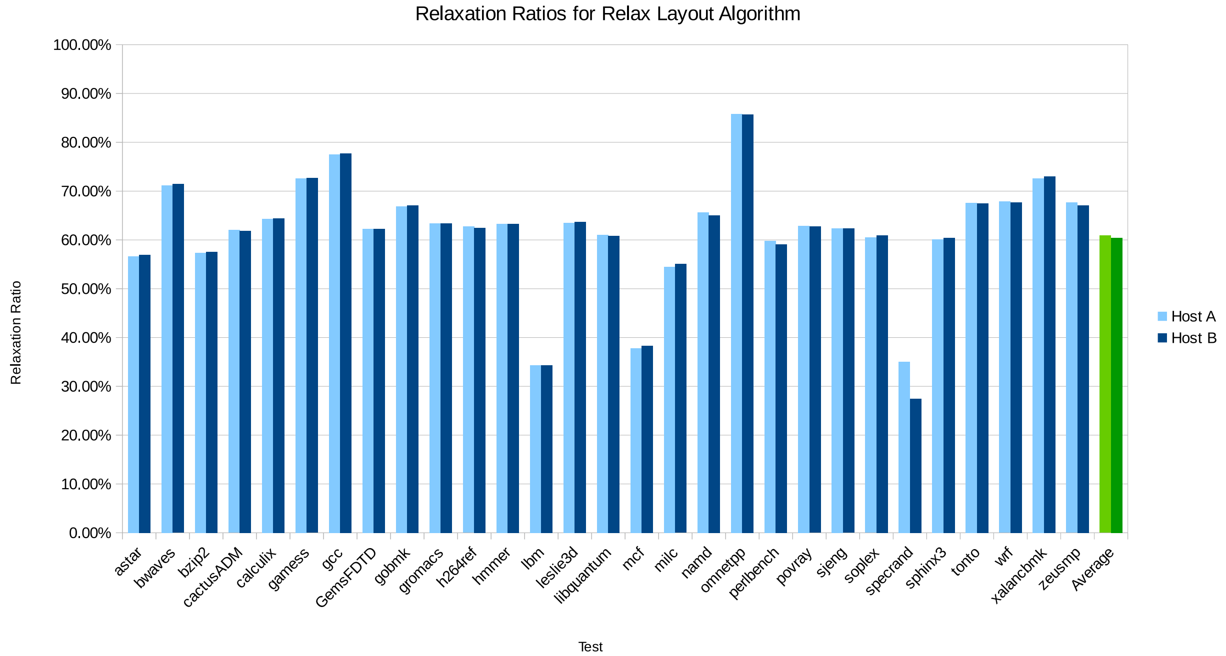

Figure 9.14: The percentage of the time that an unconstrained link can be replaced with a constrained link in the process of statically rewriting the applications of the SPEC benchmark suite. Host A's relaxation ratios are shown in light blue. Host B's relaxation ratios are shown in dark blue.

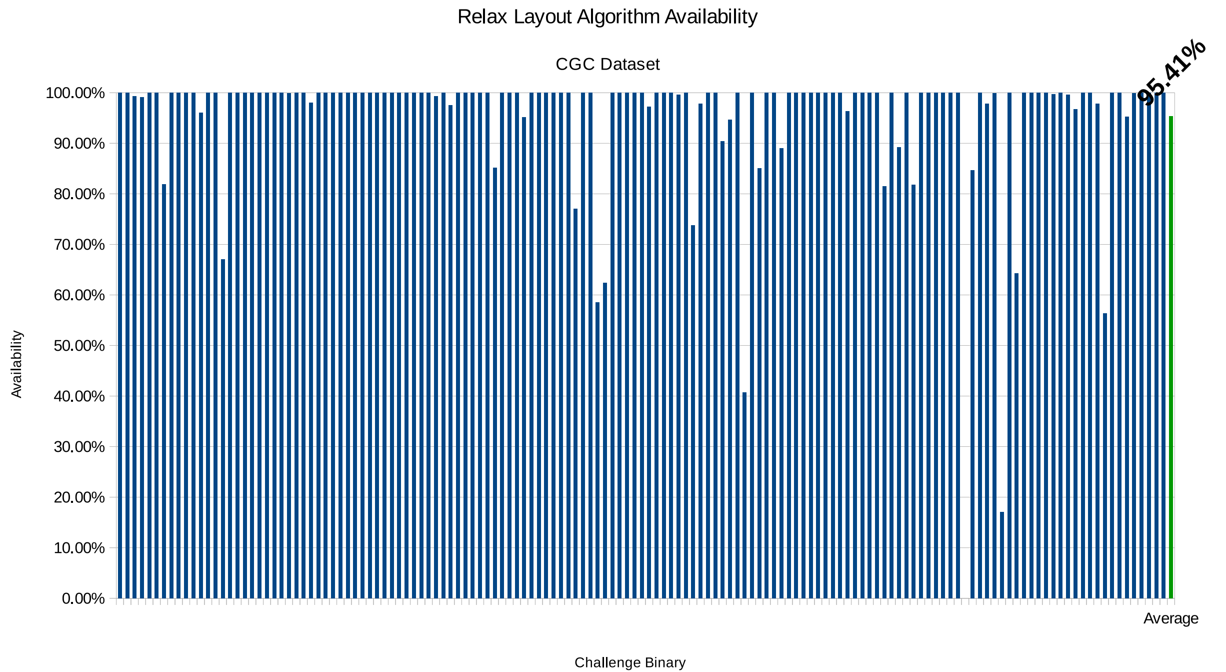

Figure 9.15: Availability scores for the RCBs in the CGC dataset when rewritten using the Relax Layout algorithm.

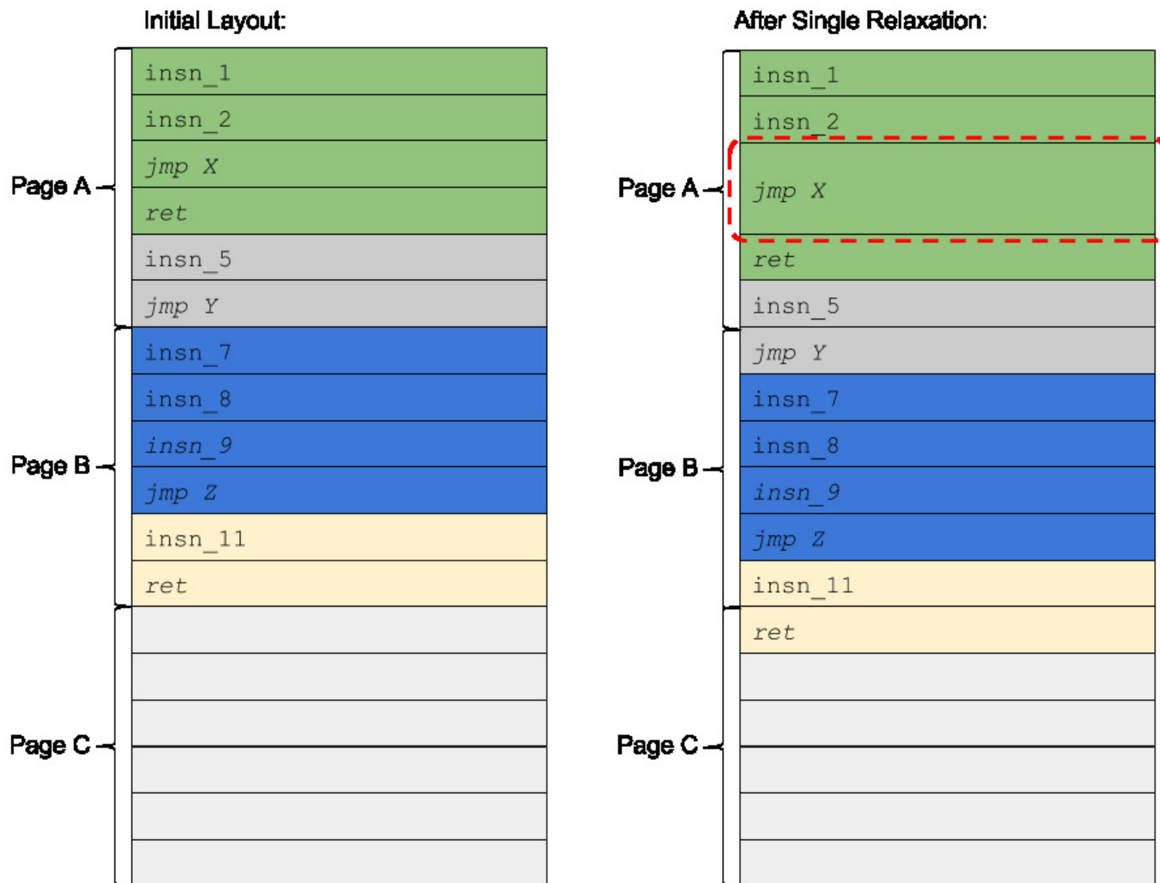

Figure 9.16: An example of the way that a single relaxation of the implementation of a link on a platform with variable-length instructions can cause a dollop to slide to a different page.

optimization to convert all `jmp` instructions to the version of those instructions that require the least amount of space given the distance to their targets. This optimization is known as Relaxation.

LLVM's assembler starts by parsing instructions into a series of fragments. There are two types of fragments – fixed-sized fragments and relaxable fragments.[5] The former are also known as data fragments and their size cannot be changed. Relaxable fragments encapsulate a single type of instruction – the `jmp` instruction – and their size may change throughout assembly. By default, the assembler encodes all `jmp` instructions using the shortest form given the target ISA. This is the same default stance that the Relax Layout algorithm takes when reassembling the statically rewritten program.

After LLVM's assembler builds the fragments, they are linked together in a list. Fragments are like dollops but, because there is only one type of dollop, all dollops are relaxable. LLVM's assembler performs its relaxation optimization algorithm using the list of fragments as its input.

[5] There are other types of fragments but those types are not relevant to this discussion.

The assembler uses a list of fixed-size and relaxable fragments instead of an array of bytes to represent the program as it is being assembled and optimized in order to minimize memory reallocations. If the program were represented as a single byte array, a change to the size of any of the program's instruction would prompt a reallocation of the entire array. In other words, fragments are not fundamental to the optimization's operation – they are simply an optimization to its implementation.

The assembler's relaxation optimization iteratively lays out the program. As it is laying out the program, if it encounters a `jmp` instruction that cannot reach its target within the size as it is currently encoded (i.e., the `jmp` needs to be encoded using a longer version defined by the ISA), the size of the instruction is changed. After every layout that changed the size of a `jmp` instruction, the assembler lays out the program again. The iterative process terminates when it is able to lay out a program without changing the size of any of its `jmp` instructions.

9.6 Conclusion

The default behavior of the algorithms of the Reassembly phase create unconstrained links in all cases. This eases the implementation of the algorithms of the Reassembly phase because it is possible to actualize two connected dollops independently of one another. However, on platforms where the link instruction is implemented with a variable size instruction (e.g., `jmp` on x86), this results in potentially a significant amount of wasted space. Inefficient link instructions are most common when the Profile Layout algorithm is performing reassembly and connected dollops are actualized near one another. The Relax Layout algorithm, originally an extension to the Profile Layout algorithm, builds on this intuition to perform reassembly in a way that optimizes the in-memory and on-disk size of statically rewritten binary programs/libraries.

Experimental results show that with respect to the on-disk and runtime memory overhead, the Relax Layout algorithm did generate statically rewritten programs/libraries that had less on-disk and runtime memory overhead than those generated with the Profile Layout algorithm. In fact, the Relax Layout algorithm was the only optimized reassembly algorithm to show significant improvements in on-disk and page fault overhead for the applications of the SPEC benchmark suite.

Unfortunately, while the performance of the CBs of the DARPA CGC dataset statically rewritten using the Relax Layout algorithm improved slightly, the applications of the SPEC benchmark suite statically rewritten with the Relax Layout algorithm showed a decrease in overall performance. The cause of this performance decrease is *sliding* of dollops during the relaxation process. Future work will investigate solutions to this problem.

Part IV

Applications

Parts II and III described the fundamental design and architecture of the static binary rewriter and extensions to those algorithms that produce more efficient statically rewritten programs/libraries. However, the goal of this research described in Part I was not limited to the creation of an architecture and algorithms of a static binary rewriter, *per se*. One of the fundamental goals of this research was to show that this type of architecture could be employed to transform SOUP to add security and reliability. In this part, three applications of the architecture and algorithms described in Parts II and III are described that demonstrate their utility in adding security and reliability to SOUP.

Chapter 10

Mixr

10.1 Introduction

Mixr is a system that improves on the standard ASLR security technique by adding "runtime ASLR" to existing software without requiring access to the software's source code or debugging information. ASLR was first described and implemented by the Pax Security Team as a way to add "randomness into addresses used by a given task ... [which] will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task" [168]. ASLR modifies the location of sections of a program's code every time the program starts. The promise of ASLR is protection against attacks that rely on information leaked from the vulnerable program about the location of certain code blocks in memory at runtime (e.g., return-to-libc and ROP). Common implementations of ASLR do not live up to this promise, however [195].

A "runtime ASLR" defense, however, would. As a runtime-ASLR-protected program executes, its program instructions are periodically and randomly rearranged in memory in a way that invalidates the leaked information about a program that the adversary would otherwise be able to use to mount his/her attack. Runtime ASLR is sometimes also referred to as binary stirring and various researchers have demonstrated that it is effective at preventing many common attacks. See Section 10.6. While the prior research on runtime randomization has demonstrated the effectiveness of the technique, each realization of the technology has limitations that make it unsuitable for the context described in Part I.

Mixr is a system that adds runtime ASLR to existing software without requiring access to the software's source code or debugging information. Mixr is implemented using the SDK for Zipr, the prototype implementation of the static binary rewriter, and adds security to binary programs and software libraries that

are vulnerable to control hijack attacks and information leaks. A Mixr-transformed program rerandomizes its code layout throughout execution. The Mixr user specifies when the program will rerandomize and the granularity of those randomizations. For example, the user can specify that the program rerandomizes itself on 60-byte boundaries every time the `write()` system call is invoked. The hypothesis is that these runtime rerandomizations will protect Mixr-transformed programs from common information leakage attacks.

The remainder of this chapter describes the technique, implementation and evaluation of Mixr. The chapter concludes with a discussion of related work and an analysis of how Mixr differs from these existing research efforts.

10.2 Motivation

Years ago, the most devastating software attacks were so-called shellcode injection attacks (injection attacks for short). In an injection attack, the attacker exploits a program that contains a vulnerable function that he/she can use to modify the stack. Through his/her attack, the attacker replaces the contents of the stack (which are typically local function parameters, return addresses, frame pointers, etc.) with executable code. When the vulnerable function completes execution and returns to its caller, the attacker has modified the return address to point to the newly installed malicious code. In a typical case, the attacker's code then launches a separate process that gives him/her unfettered access to the victim's computer.

There have been plenty of advances in software security since those attacks were pioneered which have largely mitigated the effects of those attacks. Non-executable stacks (NX), in particular, rendered most, if not all, of the code injection attacks impotent.[1]

In place of the code injection attacks, return-to-libc and ROP attacks prospered. In these attacks, the attacker replaces the contents of the return address on the program's stack with a list of addresses that direct the vulnerable program to use its own instructions to attack itself. The ROP attack was a derivative successor to the return-to-libc attack.

In a return-to-libc attack, the attacker exploits a program that contains a vulnerable function that he/she can use to modify the stack. Through his/her attack, the attacker replaces the contents of the vulnerable function's return address on the stack with the address of a libc function. In a typical attack, the `system()` libc function is usually the target. When the vulnerable function completes execution and returns to its caller, the program invokes the libc function at the target address instead of returning to the calling function.

[1] There are similar attacks where the attacker places malicious code into the heap and transfers program execution there. Although NX cannot prevent this version of the attack, other mechanisms have been developed and deployed to defeat this variant.

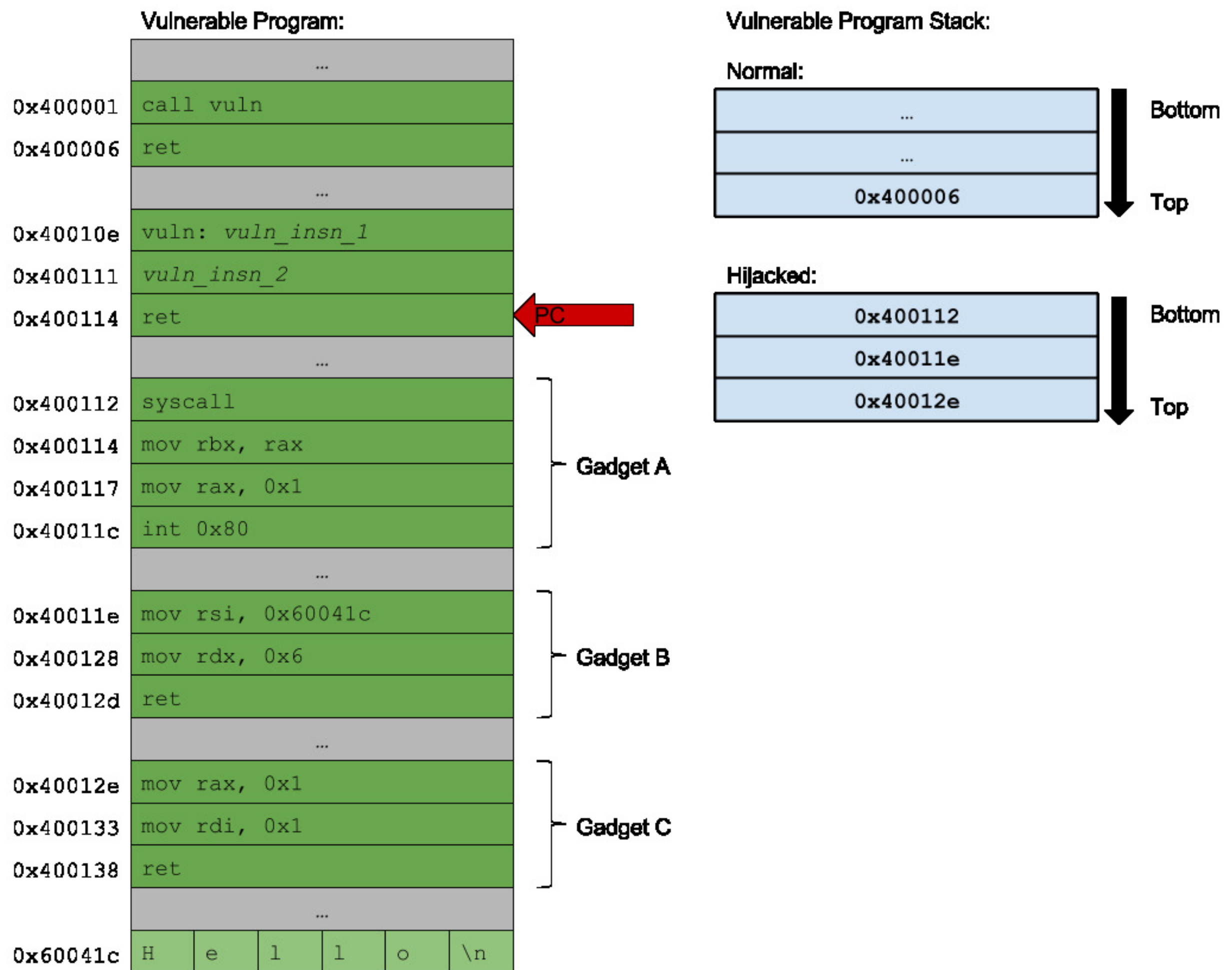

Figure 10.1: A program with a vulnerable function operating normally and after being hijacked. This program has three ROP *gadgets* that, when used together in a ROP *program*, can be used to print "Hello" to the screen.

Continuing the typical case, this results in the execution of an external program on the victim's computer of the attacker's choosing.

In a ROP attack, the attacker exploits a program that contains a vulnerable function that he/she can use to modify the stack. Through his/her attack, the attacker replaces the contents of the stack with a sequence of addresses, each of which points to small snippets of code in the vulnerable program. These small snippets of code are known as *gadgets* [194] and are targeted because of their particular properties. Gadgets typically perform two or three very minor operations (like loading a value into a register or swapping the values of two registers) and then execute a `ret` instruction. The combination of these properties makes it possible to *program* an attack as a sequence of gadgets written according to the gadget's addresses in memory.

The vulnerable program in Figure 10.1 has three gadgets, A, B and C. When used together in a program,

these gadgets can be used to print "Hello" to the screen using the `write` system call. `write()` takes three parameters. The first parameter is the file descriptor that is the target of the write. The second parameter is a pointer to the address of the string to write. The third parameter is the length of that string. The `write` system call is 1. When invoking a system call in a x86-32 Linux Kernel, the parameters are placed in *rdi*, *rsi*, and *rdx*. The system call number is placed in the *rax* register. The return value of the system call is placed in *rax*.

Gadget *A* actually invokes a system call depending on the value of the *rax* register at the time the instruction is executed. Gadget *B* fills *rsi* and *rdi* which effectively sets the value of the second and third parameters to a `write` system call. Finally, Gadget *A* specifies that it is the `write` system call to be invoked (by filling *rax*) and sets *rdi* which effectively sets the value of the first parameter to a `write` system call.

The contents of the program's stack when the PC is at `0x400114` and the program is operating normally are shown on the right. The `ret` that is about to execute will return program control to the instruction immediately after the `call` issued to invoke the `vuln` function.

The contents of the program's stack when the PC is at `400114` and the program is hijacked are shown on the right. The contents of the stack represent a ROP program that will, effectively, invoke `write(1, ''Hello\n'', 6);`. When the `ret` at the PC executes, the address on the top of the stack will be popped and program control will be transferred to that address and Gadget C will execute. This sets the system call to invoke and sets the first parameter to that system call. When the `ret` at the end of Gadget C executes, the address on the top of the stack will be popped and program control will be transferred to that address and Gadget B will execute. This sets the second and third parameter to the system call that will eventually be invoked. Finally, when the `ret` at the end of Gadget B executes, the address on the top of the stack will be popped and program control will be transferred to that address and Gadget A will execute. This actually invokes the system call and "Hello" will be printed to the screen.

What the return-to-libc and ROP attacks have in common is an assumption that an attacker can know the location of parts of the vulnerable program in memory at runtime. In return-to-libc attacks, it is assumed that the attacker can learn the address of the target libc function in memory. In the ROP attacks, it is assumed that the attacker can learn the address of the ROP gadgets necessary to build their attack program. The ASLR techniques introduced by the Pax Security Team provided an effective mechanism for invalidating this assumption and the implementation of ASLR in OSes has rendered many return-to-libc and ROP attacks invalid.

In their place, however, have been other attacks that are able to circumvent the protection offered by ASLR. The blind ROP (BROP) attack is just one such example. See Section 11.4.3 for a complete description of the attack.

Besides an advanced attack like BROP (and its successors JROP, etc.), there are fundamental reasons why ASLR itself is not completely effective. First, ASLR can only randomize the addresses of certain programs. If the user executes a program that the developer did not compile with PIC, the OS's ASLR-enhanced loader cannot place the program into memory at a location different than the one specified when it was being linked. Second, many ASLR implementations are limited by the constraints of the OS's virtual memory implementation. The constraints of the virtual memory system mean that ASLR cannot take advantage of all the entropy the system can offer for generating random addresses to place the code in memory at runtime. Third, common ASLR implementations only randomize the starting location for programs' sections. In other words, no matter how ASLR randomizes the location of the program's text section, the instruction locations are always the same relative to one another. Fourth, ASLR implementations are not effective against programs that do just-in-time compilation or can otherwise modify executable code at runtime (e.g., [43]).

Despite these shortcomings, ASLR is very *nearly* a completely effective defense against modern attack techniques. "Runtime ASLR", a type of ASLR that changes program layout throughout execution with fine granularity and high entropy, addresses the shortcomings identified with traditional ASLR and provides additional software protection.

10.2.1 Threat Model and Defenses

Mixr is designed to limit an attacker's ability to maliciously use the information he/she can learn about the runtime status of a vulnerable program. Mixr does not attempt to mitigate attacks that do not require the attacker to have knowledge about the target program.

It is assumed that the adversary can learn "information" about the vulnerable program. The information available about the program to the attacker includes the contents of memory and the location of instructions in memory. This information is available to the attacker from an *oracle* ($\mathcal{O}$) in the vulnerable program that gives responses to an attacker's *query* (q). Although the attacker can submit queries and get answers, the security architect attempting to secure the target program is aware of the parts of the vulnerable application that may be oracles.

In a typical attack scenario, oracles will exist at places in the vulnerable program where the attacker can trick the program into executing code that leaks information of their choosing. For example, for a vulnerable program on an x86 platform, an oracle may exist where the attacker can control `%rdi`, `%rsi`, and `%rdx` and then execute a `write()` system call. See Figure 10.1.

It is also assumed that the program contains a vulnerability through which an adversary has the power to use query responses to hijack program control. While it is assumed that the vulnerability allows the attacker

to control program execution and write data anywhere in memory, the OS upon which these vulnerable programs execute has protections against executing code injected onto the stack and into the heap.

The oracle's responses to the attacker's queries give him/her information required for attacking the vulnerable program. Learning enough information to mount a successful attack may require just one query. Alternatively, it may be the case that several queries are required before the adversary has enough information to attack.

$$\begin{aligned} response &= \mathcal{O}(q) \\ t_{response} &= TimeOf(response) \\ t_{invalidate} &= TimeOf(Invalidate(response)) \\ t_{attack} &= TimeOf(Attack(response)) \end{aligned} \tag{10.1}$$

Invalidate is implemented with an *appropriate* runtime rerandomization. When a program appropriately rerandomizes itself, responses to queries prior to that rerandomization are no longer valid and are not useful to the adversary when mounting his/her attack. In this context, an appropriate rerandomization is one done with fine enough granularity so that:

1. responses that contained the address a of instructions $i_1, i_2, \ldots, i_n$ required to launch an attack no longer points to those instructions;
2. responses that contained the address a of bytes $b_1, b_2, \ldots, b_n$ required to launch an attack no longer points to that data.

The period of time between when an attacker receives the oracle's response to a particular query and the time when that response is invalidated is known as the *attack window* (W). The attack window opens as soon as the adversary receives the response(s) required to mount an attack. The attack window closes as soon as those responses are invalidated.

$$W = [t_{response}, t_{response} + t_{invalidate}] \tag{10.2}$$

If the adversary cannot deploy an attack before the attack window closes at time $t_{response} + t_{invalidate}$, he/she cannot base their attack on that information. Invalidating oracle responses as quickly as possible through frequent appropriate runtime rerandomizations will decrease W and increase software security.

10.2.2 Design

Mixr is a system that allows the security architect to add appropriate invalidation points to potentially vulnerable software in order to minimize W and, consequently, improve its security. At a high level, to meet the demands and definitions in Section 10.2.1 and operate in the context outlined in Section 10.1, the Mixr system must meet the following requirements.

First, the system must be able to protect SOUP. Security architects are in a position where they must defend all applications no matter whether the program's source code is available or the software has debugging or metadata information embedded.

Second, the protected program must be able to run on the same platform as the original without modifications to the system's compiler, linker, loader or kernel.

Third, Mixr's design must not restrict the security architect's choice of which security policy to implement. As a consequence, the system must be optimized in a way that makes invalidations as lightweight as possible otherwise the security architect may not be able to implement his/her desired policy without excessive performance penalties. In Section 10.4, an alternate design is considered that demonstrates the tradeoffs of the design decision to optimize for lightweight rerandomizations.

Fourth, the system must be able to protect a program with granularity finer than basic blocks, functions, segments, or modules. The granularity is related to whether an invalidation is appropriate or not. As mentioned previously, one of the shortcomings of ASLR is that it randomizes the program segments as a unit. This means that an attacker can leverage information about the *relative* location of two instructions to craft his/her attack. As the granularity of the rerandomization increases, the less likely it is that the attacker can gain usable information about the relative location of instructions that may be useful in an attack. The granularity will affect runtime performance and must be an option for the user. A user more interested in security will specify a finer granularity while a user with more tolerance for risk may specify a coarser granularity in order to minimize performance overhead.

Mixr works in three phases. In the first phase, the preparation phase, the original program is rewritten to support runtime rerandomization. In the second phase, the linking phase, the original program is augmented with the components that implement randomization at runtime. Third, the rewritten program rerandomizes itself at runtime at the user-specified invalidation points.

The preparation phase relies on the user to specify the invalidation point(s) and the granularity. The native program is first organized into bundles whose size matches the user's choice of granularity. Each bundle is self-contained insofar as it contains all the information necessary to function correctly no matter where it exists in memory. Then the input binary is annotated to mark the user-chosen invalidation points.

See Section 10.3 for the details of the bundles and the metadata contained in each bundle required to make them self-contained.

The linking phase also takes user input and adds the code necessary to implement the runtime randomization at invalidation points as the program executes. The user chooses one of the several different runtime randomization algorithms that perform the invalidations. See 10.3.2 for the details of these algorithms.

No matter what algorithm the user chooses for invalidation, it is the code added in the linking phase that is executed at runtime to perform runtime rerandomization. Because bundles are self-contained, the implementation of the invalidation algorithms is straightforward. See Section 10.3.2 for details.

10.3 Operation

The Mixr preparation step converts a binary program into a set of *dollop bundles*, a combination of a dollop and associated metadata. The dollop bundles are stored in the free spaces of the rewritten program. Figure 10.2 shows an idealized representation of the layout of a Mixr-prepared program at startup. This representation is idealized because it does not show how the dollop bundles must be placed so they do not conflict with pinned addresses. Dollop bundles, while containing metadata themselves, are just one of the metadata components of a Mixr-prepared program.

10.3.1 Metadata

A Mixr-prepared program consists of several different types of metadata that comprise a dollop bundle, link dollop bundles to one another and serve as handles that are used to rerandomize a program at runtime. Each of the metadata components of a Mixr-prepared program are described in the remainder of this section.

Dollop Table

The dollop table is a contiguous area of memory that contains information about the current, randomized place of each dollop bundle (Section 10.3.1) in memory. For every dollop bundle, the dollop table contains three pieces of information.

Address A pointer to the dollop bundle's place in memory.

RW Table Offset A pointer to the start of the dollop bundle's RW Table (Section 10.3.1) entry. This offset is a relative value from the start of the dollop.

In-Use Flag A flag to indicate whether the dollop bundle is referenced from the program's runtime stack.

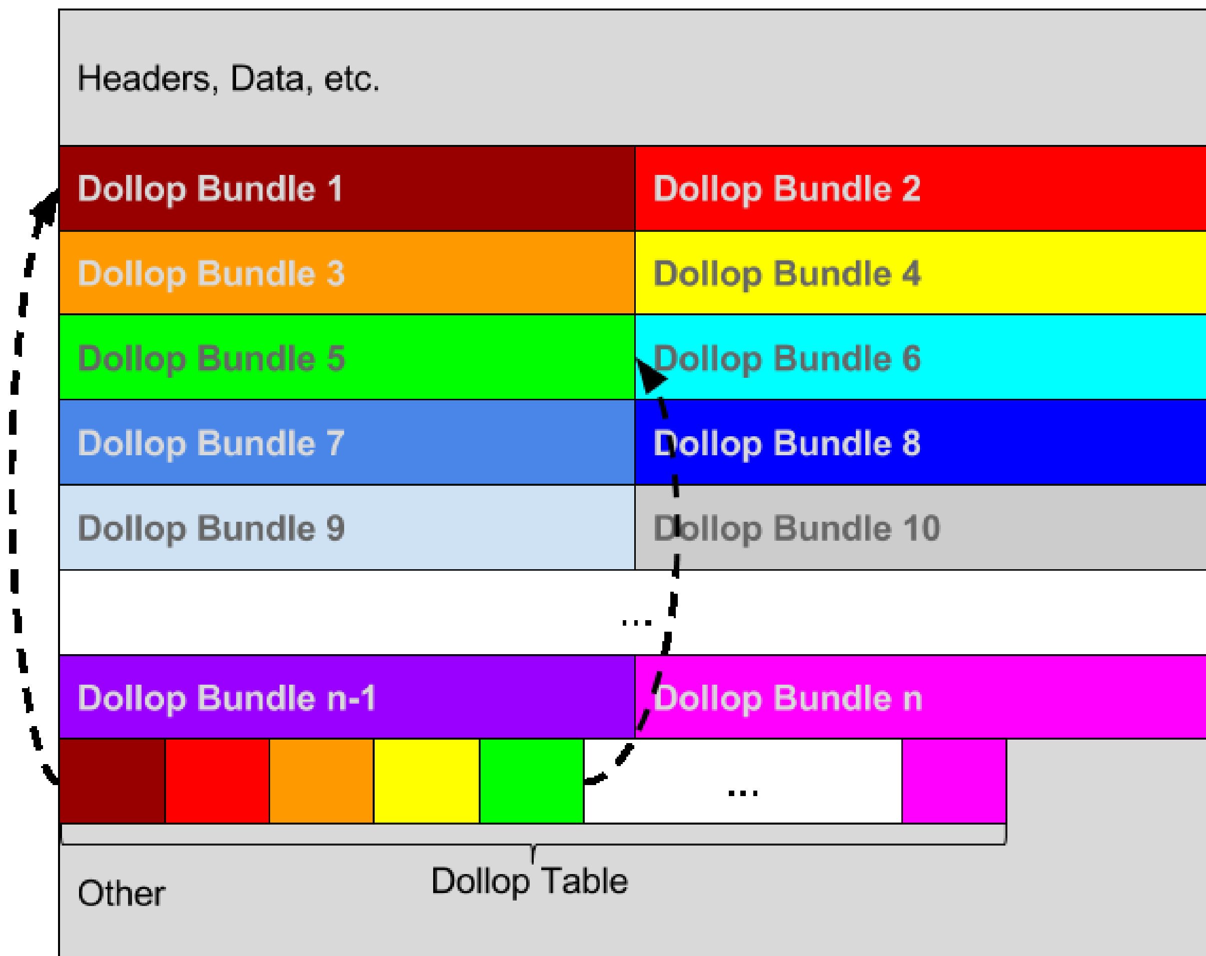

Figure 10.2: A Mixr-prepared program at startup.

Dollop Bundles

For every dollop, there is a dollop bundle. The dollop bundle contains the dollop itself and the associated metadata needed to make the dollop self-contained. For any dollop that cannot be entirely contained within a dollop bundle, the dollop contains an added instruction (the *fallthrough* instruction) that links it to the subsequent dollop bundle. In Figure 10.3, the instructions colored in red are fallthrough instructions. All the instructions in `foo` would constitute a single dollop since there is only one unconditional program control transfer operation (see Section 4.4.4). However, because of the requirement that a dollop bundle be a fixed-size, the dollop must be split among multiple dollop bundles and those are connected with fallthrough instructions. In order to completely contain a sequence of assembly instructions that can execute no matter where they are in memory at runtime, there needs to be additional information for any jump or fallthrough instruction and any instruction that makes a position-dependent memory reference. The dollop bundle contains a *dollop*

Figure 10.3: Creation of dollop bundles from straight line code in the original program.

entry (DE) table and a *read/write (RW) table* to handle these situations, respectively.

Read/Write Table

The RW table contains information needed to fixup the instructions in a dollop bundle that reference memory based on the location of the instruction in the program's address space, called position-dependent memory accesses.

Absolute Address The absolute address of the memory access by the position-dependent memory access instruction. The absolute address is calculated offline when the original binary is being prepared.

Instruction End The location of the end of the instruction to which this RW table entry applies.

Address Offset The location of the offset within the instruction itself. This value is an offset from the end of the instruction.

Address Size The size of the address field in the position-dependent memory access instruction.

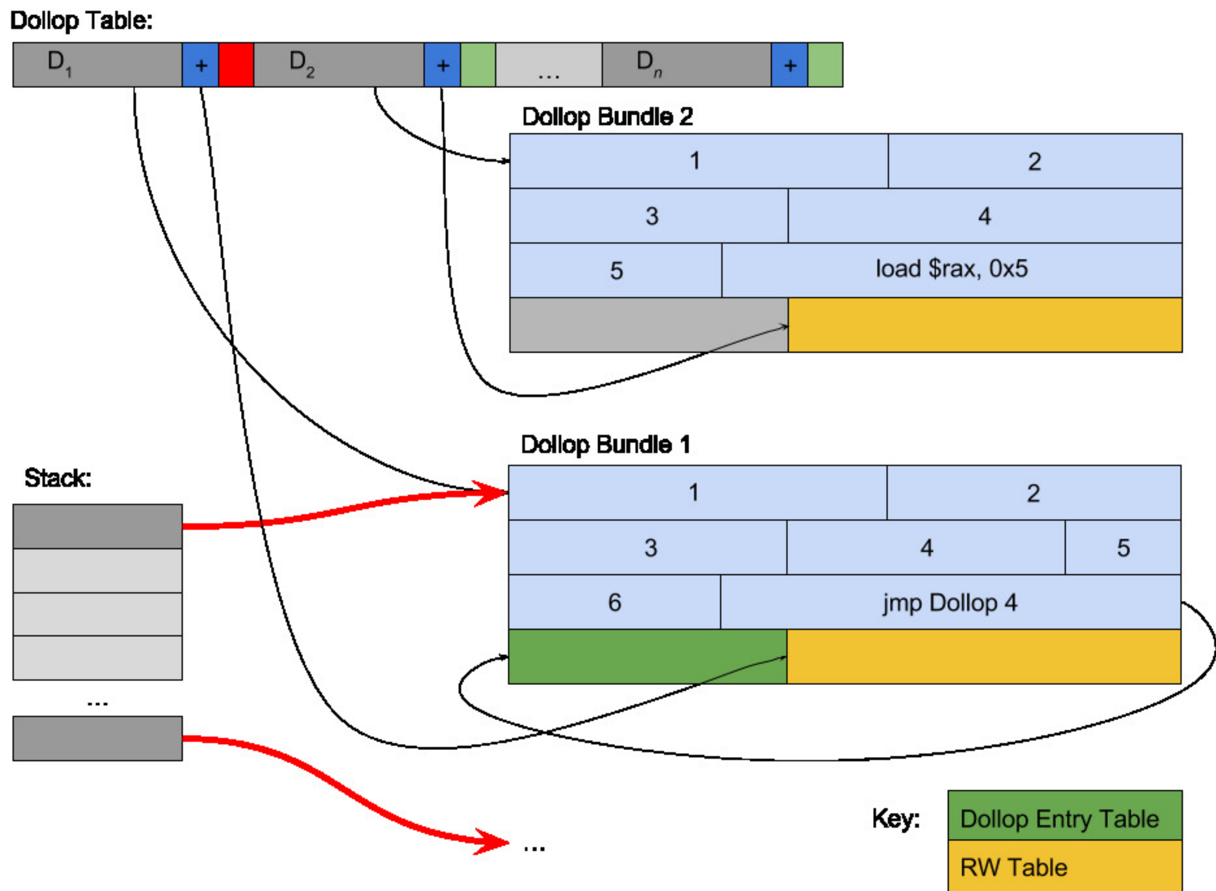

Figure 10.4: The links between the metadata components of a Mixr-prepared program.

Dollop Entry Table

The DE table contains information that augments the jump instructions in a dollop bundle. Each entry in the DE table is itself a jump instruction that transfers program control indirectly to the jump instruction's target dollop through the address of the dollop bundle in memory as stored in the dollop table.

Figure 10.4 shows the links between the metadata components described in the preceding subsections of a Mixr-prepared program. Dollop Bundle 1 is Active. The first entry in the stack points to Dollop Bundle 1 and the in-use flag is set for its entry in the dollop table. Dollop Bundle 2, on the other hand, is not active. There are no links to it from the stack and, therefore, the in-use flag is not set in its entry in the dollop table. In Dollop Bundle 1 there is a `jmp` that points to an entry in Dollop Bundle 1's DE table. In Dollop Bundle 2, however, there are no `jmp` instructions and, therefore, the DE table is empty. In reality, such a situation would result in a dollop bundle that does not contain any space for the DE table. Figure 10.4 shows a space for educational purposes only. Dollop Bundle 2 has a position-dependent memory access instruction

and, therefore, the RW table contains an entry. Dollop Bundle 1, on the other hand, does not have any position-dependent memory access instructions, and, therefore, the RW table is empty. Again, in reality a lack of position-dependent memory access instructions in the dollop bundle means that the dollop bundle does not contain any space for a RW table. The space is shown here for educational purposes only.

10.3.2 Rerandomization

The rerandomization algorithm is triggered every time the program reaches the user-defined invalidation point. Before rerandomization occurs, the algorithm searches for the dollops referenced by the program's runtime stack. The in-use flag of every referenced dollop is set. Once this search is made, rerandomization itself can happen.

The user has a choice of one of three rerandomization algorithms to deploy. Each rerandomization algorithm works fundamentally by choosing two different dollop bundles to swap. The variety in these algorithms is with respect to how those pairs are selected:

Random The random rerandomization algorithm chooses pairs randomly. The user controls how many pairs of dollops are swapped at each rerandomization point. The default is 10.

Sequential The sequential randomization algorithm walks the length of the dollop table and swaps the dollop with its adjacent dollop.

Sequential Random In the sequential random randomization algorithm, every bundle in the program a, is swapped with a randomly chosen dollop b.

Once a pair of dollop bundles is chosen, the *swap* function swaps them. The definition of SWAP is shown in Algorithm 9.

In Lines 2 through 4, SWAP checks to see if either of the dollop bundles to be swapped is in use by calling the InUse function. InUse, defined between Lines 19 and 21, simply returns the value of the dollop bundle's in-use flag. If either dollop bundle is in use, this pair cannot be swapped and SWAP immediately terminates. Dollop bundles that are referenced by address from the program's stack cannot be moved because those addresses may be used as the stack is unwound when functions return. Of course, the stack could be rewritten during rerandomization to remove this constraint but rewriting addresses on the stack is complicated because identifying addresses on the stack at runtime is not exact. Rewriting mathematical values stored on the stack that are incorrectly identified as addresses affect the program's correctness.

On Line 5, SWAP calls MEMORYSWAP. MEMORYSWAP simply swaps the contents of the two dollop bundles. Because every dollop bundle is a fixed size (the size matches the user-specified granularity), two dollop bundles can always be swapped without having to check whether there is adequate space.

On Lines 6 and 7, the dollop table entries for the two dollop bundles are updated with their new addresses. The addresses are calculated by calling an ADDRESSOF function. This function, not shown here, simply returns the address of the start of the dollop bundle.

Finally, the RW tables are updated for the two swapped dollop bundles. Lines 8 and 9 in SWAP calls UPDATERW to accomplish this. UPDATERW is a single loop over every entry in a dollop bundle's RW table (Line 12). For every entry rw in the dollop bundle's RW table, the following steps occur:

1. A $TargetAddress$ is calculated. The $TargetAddress$ is the PC-relative address where the location-dependent instruction should access based on its current position within the program. (Line 13)
2. $TargetAddress$ is truncated to fit within the space allotted to it in the location-dependent memory access instruction. (Line 14) The size of the address operand for the position-dependent memory access instruction is stored in the RW table entry's address size field.
3. $TargetAddressAddress$ is calculated. (Line 15) The $TargetAddressAddress$ is the absolute memory address where the size-appropriate $TargetAddress$ is written. The $TargetAddressAddress$ is the address of the beginning of the dollop plus the RW table entry's instruction end adjusted by the instruction offset.
4. The $TargetAddress$ is written into the proper place in the position-dependent memory access instruction. (Line 16) This is a simply call of the `memcpy` standard library call.

Algorithm 9 Swapping a pair of Dollop Bundles

1: **function** SWAP(Dollop Bundle A, Dollop Bundle B)
2: **if** INUSE(A) OR INUSE(B) **then**
3: **return**
4: **end if**
5: MEMORYSWAP(A,B)
6: DT.A.$Address$ ← ADDRESSOF(A)
7: DT.B.$Address$ ← ADDRESSOF(B)
8: UPDATERW(A)
9: UPDATERW(B)
10: **end function**
11: **function** UPDATERW(Dollop Bundle db)
12: **for all** rw∈db.$RWEs$ **do**
13: TargetAddress← ADDRESSOF(db)+rw.$InstructionEnd$−rw.$AbsoluteAddress$
14: TargetAddress← TRUNCATE(TargetAddress,rw.$AddressSize$)
15: TargetAddressAddress← ADDRESSOFF(db)+rw.$InstructionEnd$−rw.$InstructionOffset$
16: MEMCPY($TargetAddressAddress$, $TargetAddress$)
17: **end for**
18: **end function**
19: **function** INUSE(Dollop Bundle db)

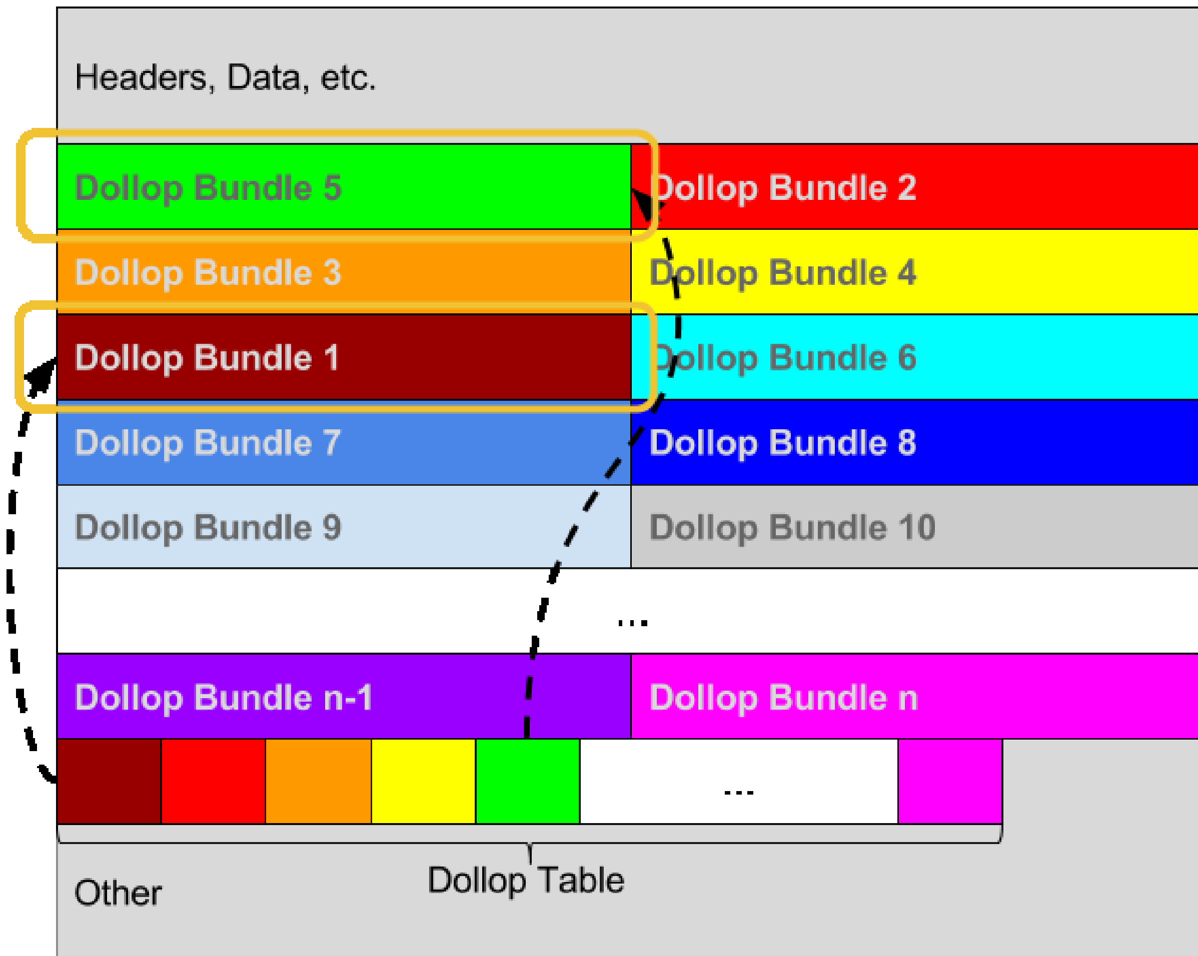

Figure 10.5: A Mixr-prepared program after a single rerandomization.

20: **return** *db*.InUseFlag
21: **end function**

Figure 10.5 shows the idealized representation of a Mixr-prepared program after a single shuffle. In this figure, Dollop Bundles 1 and 5 have been swapped. Note that the entries for Dollop Bundles 1 and 5 in the dollop table are updated to reflect the new location of those dollop bundles in memory.

10.4 Alternate Design and Implementation

As mentioned in Section 10.2.2, one of the design goals of Mixr was to make rerandomization efficient and simple to implement. This goal led to the decision to make every dollop bundle self-contained. For all but instructions that make position-dependent memory references, the resulting design made it possible to accomplish rerandomization with simple memory copies. To properly rerandomize instructions that make position-dependent memory references, however, some additional fixup is necessary.

Instructions that make position-dependent memory references are not the only instructions that require special handling if they are to be completely encapsulated in a bundle. In the original design, jump instructions are modified so that their target is always to an entry in the dollop bundle's metadata. Those targets then implement the position-independent transfer of program control that takes into account the current position of the target dollop in the program's address space (See Section 10.3.1).

This design decides for the user that performance of runtime rerandomization is more important than the runtime overhead incurred during normal program execution. During runtime rerandomization, jumps do not have to be rewritten but their execution incurs additional overhead because of the memory indirection.

Giving the user a runtime randomization implementation that has low overhead affords him/her the opportunity to specify frequently executed invalidations without worrying about performance impact and was explicitly recognized in the original design. However, it is not a choice that is proper for every use case. When Mixr is deployed on software where the Mixr user specifies that rerandomization occur very infrequently, the overhead of the additional indirection quickly outweighs the benefit of a high performance rerandomization implementation. Therefore, the Mixr user may benefit from an alternate technique for dollop encapsulation that optimizes for normal program execution.

An alternate encapsulation method treats jump instructions in the same way that the original encapsulation treats instructions that make position-dependent memory references. In the alternate design and implementation, there is still a DE table but the information stored in each entry is different. Instead of each entry being itself a jump instruction that redirects program control through the dollop table, the entry contains only the location of the target dollop's entry in the dollop table.

In practice, the alternate encapsulation method requires very little change to the specifics of the encapsulation but does result in space savings[2] – one byte per DE. The entry in the DE table in the alternate version is 5 bytes and contains two entries.

Instruction End The dollop-relative location of the end of the instruction to which this DE table entry applies. This location is stored in 1 byte.

Dollop Table Pointer The address of this DE table entry target dollop's entry in the dollop table. This address is stored in 4 bytes.

Besides changes to the dollop's encapsulation, the alternate design and implementation has implications for the runtime rerandomization process, too. Jump instructions are now treated like instructions that make

[2]The current Mixr implementation of the alternate encapsulation technique uses the same amount of space to ensure binary compatibility between encapsulation techniques.

position-dependent memory references. For every entry j in the dollop bundle's DE table, the following steps occur:

1. A $TargetAddress$ is calculated. The $TargetAddress$ is the PC-relative address of the target dollop of the jump instruction. (Line 15)
2. $TargetAddressAddress$ is calculated. (Line 16) The $TargetAddressAddress$ is the absolute memory address where the $TargetAddress$ is written. The $TargetAddressAddress$ is the address of the beginning of the dollop plus the DE Table entry's Instruction End adjusted by 4 – the fixed size for the target addresses of the jump instructions used in Mixr.
3. The $TargetAddress$ is written into the proper place in the position-dependent memory access instruction (Line 17) – a simple call of the `memcpy` standard library call.

Algorithm 10 Swapping a pair of dollop bundles using the alternate encapsulation technique.

```
 1: function Swap(Dollop Bundle A, Dollop Bundle B)
 2:     if InUse(A) OR InUse(B) then
 3:         return
 4:     end if
 5:     MemorySwap(A,B)
 6:     DT.A.Address ← AddressOf(A)
 7:     DT.B.Address ← AddressOf(B)
 8:     UpdateRW(A)
 9:     UpdateRW(B)
10:     UpdateJumps(A)
11:     UpdateJumps(B)
12: end function
13: function UpdateJumps(Dollop Bundle db)
14:     for all j∈db.DTEs do
15:         TargetAddress← AddressOf(db)+j.InstructionEnd−j.AbsoluteAddress
16:         TargetAddressAddress← AddressOff(db)+j.InstructionEnd − 4
17:         memcpy(TargetAddressAddress, TargetAddress)
18:     end for
19: end function
20: function UpdateRW(Dollop Bundle db)
21:     for all rw∈db.RWEs do
22:         TargetAddress← AddressOf(db)+rw.InstructionEnd−rw.AbsoluteAddress
23:         TargetAddress← Truncate(TargetAddress,rw.AddressSize)
24:         TargetAddressAddress← AddressOff(db)+rw.InstructionEnd−rw.InstructionOffset
25:         memcpy(TargetAddressAddress, TargetAddress)
26:     end for
27: end function
28: function InUse(Dollop Bundle db)
29:     return db.InUseFlag
30: end function
```

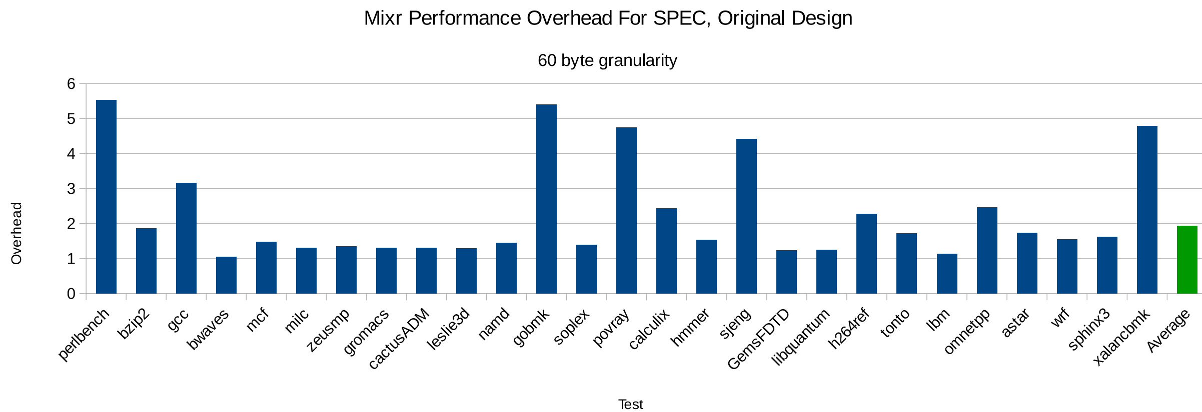

Figure 10.6: Mixr runtime overhead for the programs of the SPEC2006 benchmark suite.

10.5 Results

10.5.1 Original Design and Implementation

There are three sources of Mixr overhead. These overhead sources correspond roughly to the three different phases of Mixr.

Source 1 Preparation The first source of overhead results from the preparation phase of Mixr operation. The self-contained dollop bundles introduce an additional jump operation for every jump operation in the original program. See 10.3.1 for the details.

Source 2 Linking The second source of overhead results from the linking phase of Mixr operation. This overhead increases the size of the program on disk and may affect the memory usage of the Mixr-prepared program at runtime.

Source 3 Rerandomization The third and final source of overhead is from the rerandomization that occurs at invalidation points.

Unless noted otherwise, only Host B was used to collect data for this evaluation.

Figure 10.6 shows the results of an evaluation of the first source of overhead in a Mixr-prepared program. To isolate the performance overhead of the runtime rerandomization (overhead Source 2) from the overhead of the preparation process itself (overhead Source 1), only one invalidation point was chosen for each benchmark – the `start` function – and rerandomization was performed using the Sequential Random algorithm. As described above, the granularity of the preparation is a user-defined parameter. The granularity is set to 60 bytes in this experiment.

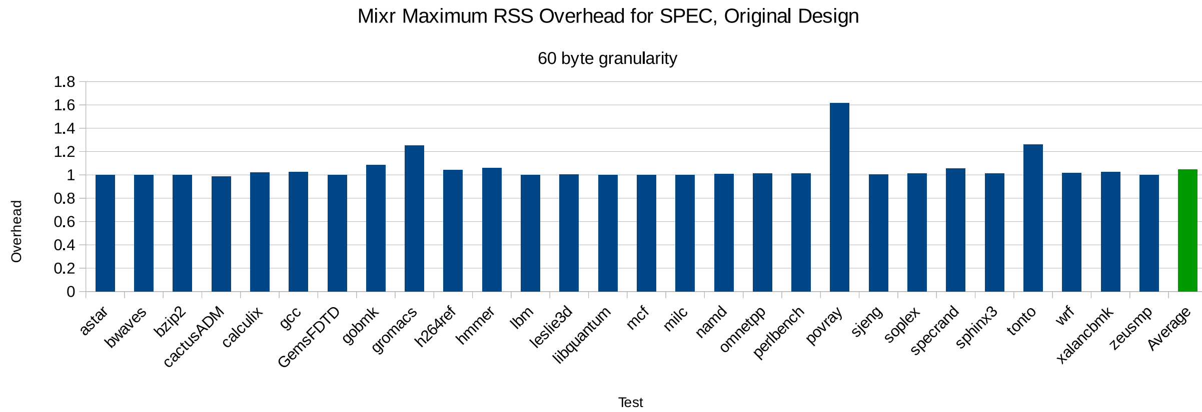

Figure 10.7: Mixr maximum RSS overhead for the programs of the SPEC2006 benchmark suite.

On average, a Mixr-prepared program exhibits 1.94x performance overhead at runtime. By virtue of the experiment's organization, it is possible to assign this overhead to two sources. The first source is the static rewriting process itself. See Chapter 5 for the overhead attributable to the rewriting process itself. The remaining overhead is from Source 1.

The variation in performance overhead among the programs in the benchmark suite is attributable to the division of the program's code into bundles. For programs in the benchmark suite that contain a significant number of jump operations in their computationally intensive kernels, the overhead is significantly higher because of the way that the bundles are designed to be self-contained. The programs in the benchmark suite whose computationally intensive kernels contain comparatively fewer jump operations exhibit much less overhead.

There are two measures of Source 2 overhead – on-disk file size of a Mixr-prepared program and the maximum RSS of a Mixr-prepared program at runtime. The difference in size between an original, input program and the Mixr-prepared version of the program is caused by the space required for metadata (Section 10.3.1) and the code added to the Mixr-prepared program to implement the rerandomization algorithms (Section 10.3.2).

The size of the code required for the implementation of the runtime rerandomization algorithms is constant. The Sequential Random runtime rerandomization algorithm, for example, is implemented in 16 kilobytes. No effort has been made to optimize this implementation for size.

The size of the metadata required by Mixr-preparation is determined by a) the number of bundles in the program, b) the number of jumps in the program and c) the number of position-dependent memory operations. For every bundle, there is a minimum of 11 bytes of space required for the dollop table entry. For every jump, there is a minimum of seven bytes of space required. For every position-dependent memory

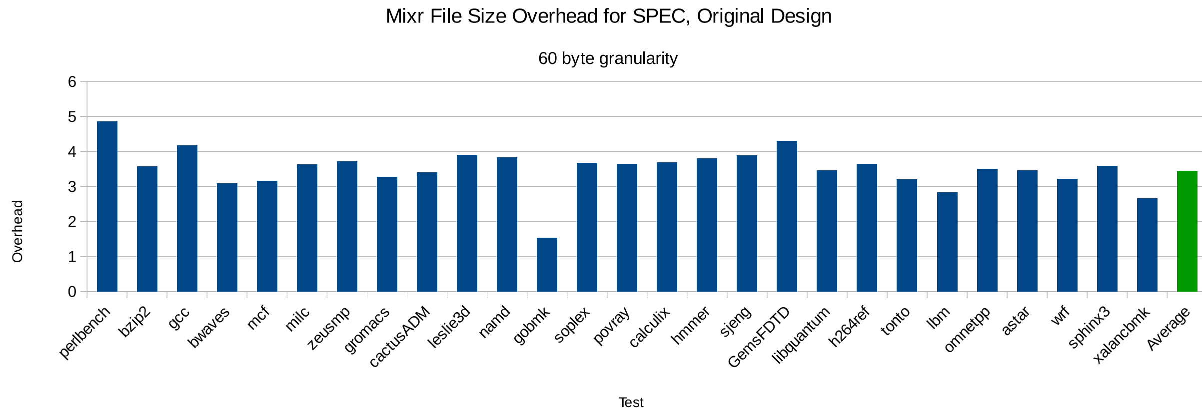

Figure 10.8: Mixr on-disk file size overhead for the programs of the SPEC2006 benchmark suite.

operation, there is a minimum of 12 bytes of space required. See Section 10.3.1 for details.

On average, there are approximately 38421 dollop bundles per application in the SPEC benchmark suite. Encoding those dollop bundles requires an average of approximately 422 kilobytes per application in the SPEC benchmark suite. On average, there are 48278 instructions that require entries in the DE tables. The DE tables, on average, require 337 kilobytes of space for each of the programs. On average, there are approximately 4769 position-dependent memory operations in each program in the SPEC benchmark suite when the programs are Mixr'd with 60 byte granularity. The RW tables, on average, require approximately 987 kilobytes for each of the programs. In total, the metadata for each SPEC program amounts to, on average, approximately 1.74 megabytes. For reference, the average size of each of the programs in the SPEC benchmark suite is approximately 1.27 megabytes. Along with the intrinsic on-disk overhead from the static rewriting process, the overall on-disk overhead for a Mixr-prepared program with 60-byte granularity is 3.45x.

It is difficult, if not impossible, to find a set of invalidation points in the programs in the SPEC2006 benchmark suite that would allow for a consistent, systematic analysis of the overhead from Source 3. Therefore, an alternate experiment is required. Each program in the SPEC benchmark suite was Mixr'd with the Sequential rerandomization algorithm and the special configuration to rerandomize 500 times at program startup. Once the benchmark application has performed its self-randomization 500 times, the program terminates. A comparison execution was performed where the programs rerandomized 1000 times. The difference in the performance of each of the benchmarks for 500 rerandomization at startup and for 1000 rerandomizations can be used to deduce the overhead of randomizations. The results are shown in Figure 10.9.

There is a very strong linear correlation ($r^2 = 0.98$) between the number of dollop bundles in a program and the time it takes to perform a single rerandomization. The linear correlation is expected considering that

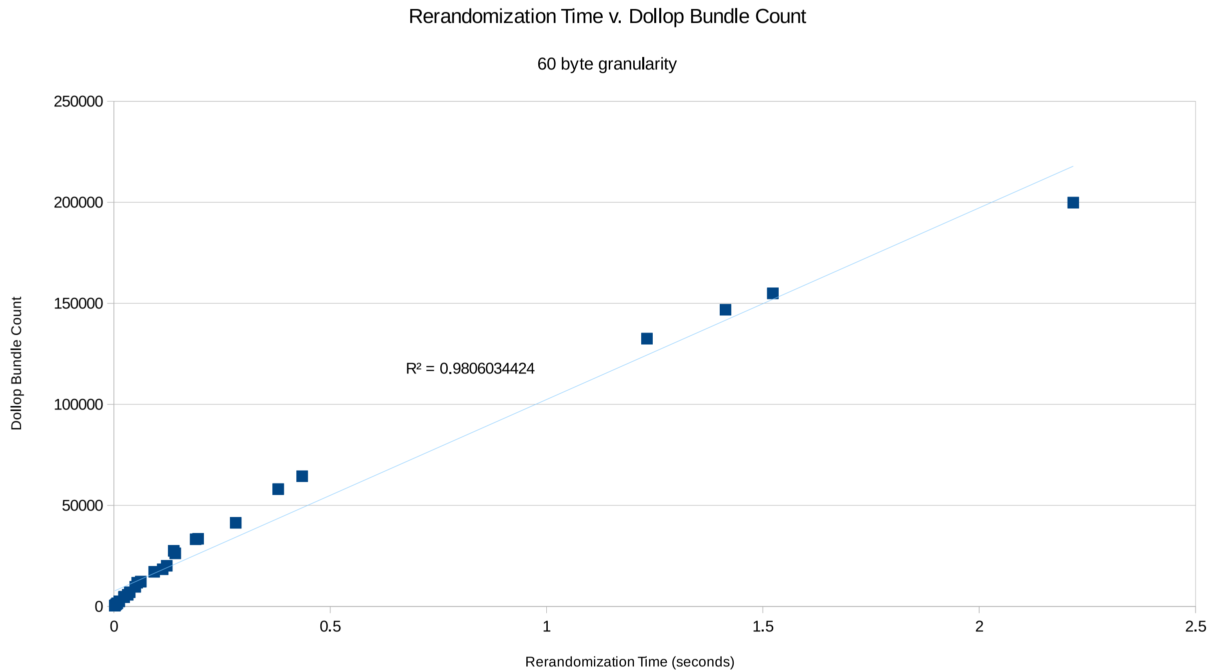

Figure 10.9: Rerandomization time plotted against the number of dollops in the Mixr'd program when prepared using the original design and implementation.

SWAP is $O(n)$ where n is the number of dollops in the Mixr'd program.

10.5.2 Alternate Design and Implementation

As explored in Section 10.2.2, there is a certain amount of extra overhead in a design focused on optimizing for rerandomizations. Section 10.4 described an alternate to the original design that is optimized to minimize the overhead of a Mixr'd program in steady state.

The following experiments evaluate this alternate design. Figure 10.10 shows the results of an experiment that are comparable to the results shown in Figure 10.6.

On average, a Mixr-prepared program that follows the alternate design exhibits 1.48x performance overhead at runtime. That is an 31.08% improvement over the original design.

Figure 10.11 shows the difference in performance between the original and alternate preparation mechanism for each of the tests in the SPEC2006 benchmark suite.

Although the alternate design optimizes the Mixr-prepared program's execution of jump instructions, there is no correlation between the number of those instructions in the program and the relative performance improvement. In other words, the programs of the SPEC benchmark suite that improve the most when Mixr-prepared with the alternate design are *not* necessarily those with the most jump instructions. This

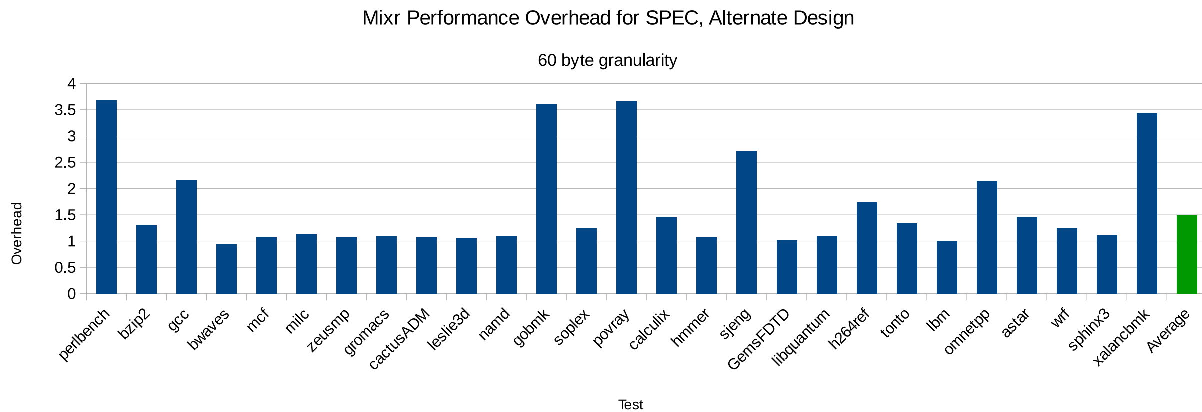

Figure 10.10: Runtime overhead for the programs of the SPEC2006 benchmark suite when Mixr-prepared with the alternate design.

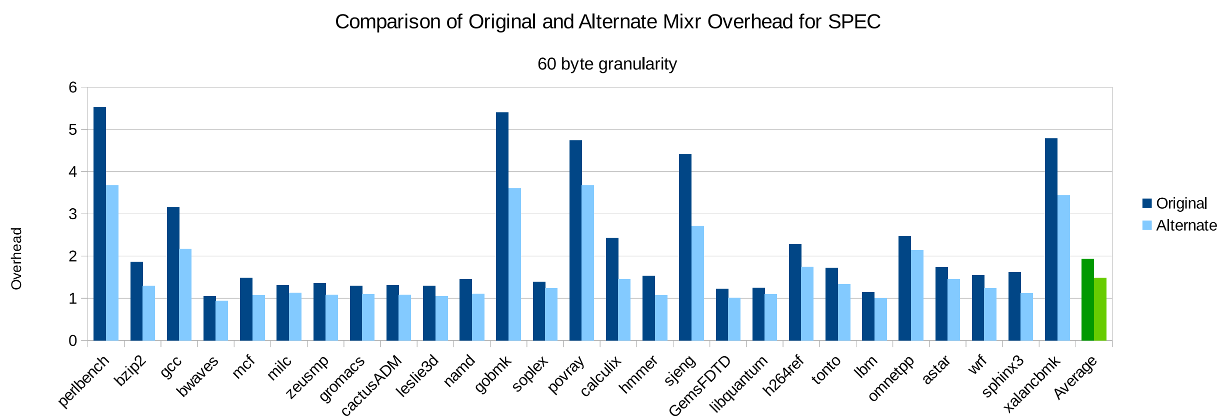

Figure 10.11: Comparison of the overhead for the programs of the SPEC2006 benchmark suite when Mixr-prepared with the original and alternate design.

result is not unexpected. The performance improvement has to do with the quantity of the jump instructions executed at runtime and not the quantity of jump instructions in the program's code.

In both the original and the alternate Mixr-preparation techniques, the dollop size should not change the steady-state performance overhead. In the original preparation technique, all jumps have an intermediate target that redirects to the actual destination. In the alternate version, all jumps are direct to their final targets. Neither one treats one jump instruction differently than any other jump instruction.

Figure 10.12 shows the results of an experiment designed to test this assumption for the alternate design. The results indicate that there is a performance improvement when the bundle sizes are increased. The mismatch between the actual results and the expected results are easily explained. Recall that at the beginning of each of the benchmarks, the Mixr-prepared program rerandomizes a single time using the Sequential Random rerandomization algorithm. As the bundles sizes increase, there are fewer bundles which means that

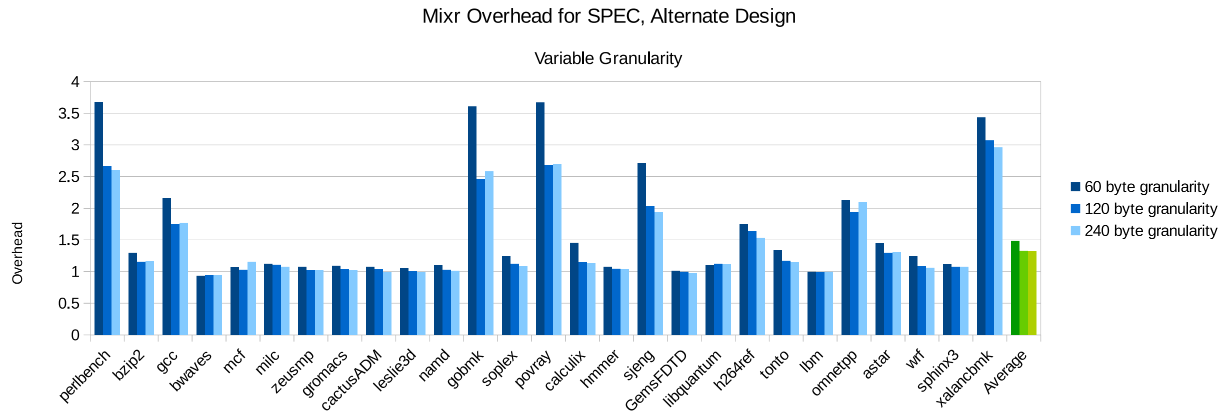

Figure 10.12: Affect of the choice of granularity on the performance overhead of the programs of the SPEC2006 benchmark suite when Mixr-prepared with the alternate design.

there are fewer operations to complete at the start of every benchmark to rerandomize and implies that the performance of the benchmark will improve, albeit slightly. The decrease in the number of bundles is the cause of the improvements show in Figure 10.12.

10.6 Related Work

There have been other efforts at run-time randomization of code. These research efforts are discussed in the following subsections. In addition to a description of the techniques employed by these parallel research efforts, their differences from Mixr are highlighted.

10.6.1 Remix

Remix is a runtime ASLR technique developed by researchers at Florida State University and Stony Brook University [50]. The goal of Remix, like most runtime rerandomization techniques, is to improve the protections offered by traditional ASLR and improve an application's resilience against ROP attacks. Their randomization technique targets user space applications for Linux and kernel code for FreeBSD.

To accomplish this goal, the Remix researchers identify a significant technical hurdle that a runtime rerandomization system must overcome to be successful. When portions of a program's code are moved at runtime, there may be pointers to those locations stored in memory, in registers or on the stack. This scenario commonly occurs when a program stores pointers to signal handlers or a program uses callbacks. To overcome this limitation, the Remix system maintains the starting address of functions and randomizes code within functions at the granularity of basic blocks. Assuming that the addresses of functions are the only

Tool	Access to Source Code	Compiler Modification	Kernel Modification	Linker Modification	Loader Modification	Granularity	Invalidation Points
Mixr						User-specified	User-specified
Remix [50]	✗	✗				Basic Block	User-specified
RuntimeASLR [132]						Modules	`fork()`
TASR [40]	✗	✗	✗	✗	✗	Segment	I/O system calls
CodeArmor [49]					✗	Segment	system calls
STABILIZER [62]	✗	✗				Functions	Every 500ms
Chronomorph [82]	✗					Basic Blocks	User-specified
Shuffler [243]	✗[1]				✗	Functions	Every 50ms
selfrando [53]	✗			✗		Functions	Startup
STIR [236]						Basic Block	Startup
Marlin [89]					✗[2]	Function	Startup

[1] Source code not required if the target binary program is compiled to save debugging information, symbol tables and relocations. Otherwise, recompilation is required.
[2] Marlin requires a modified shell to load programs.

Table 10.1: Comparison of requirements and features of each of the available runtime rerandomization technologies.

code pointers stored in memory, in registers or on the stack, maintaining the starting address of functions overcomes this obstacle.

The basic blocks of a function are emitted by the compiler in a particular order and the exit instructions of those basic blocks are sized accordingly. The ordering and size of exit instructions are particularly important for ISAs with variable-length instructions. For example, in x86, the basic block's exit instruction might be a `jmp` instruction. That `jmp` instruction can be short if the target basic block is nearby. When basic blocks are reordered at runtime, the target basic block might be too far away to be reached with a short `jmp`. Therefore, that short `jmp` must be expanded. Since Remix keeps the base function addresses fixed, the reordered basic blocks must fit within the space originally allocated by the compiler for that function. Expanding short `jmp`s into long `jmp`s might mean that the reordered basic blocks no longer fit within that space.

To handle this case, Remix requires users to build their software with a modified compiler. Remix does support adding live rerandomization to binary programs but with certain restrictions. In the case where a basic block ends with an exit instruction that could be expanded based on where basic blocks land during reordering, that basic block is merged with the another basic block and that bundle of basic blocks is randomized together. Each time a function is rerandomized, bundling choices are made anew in order to preserve as much randomness as possible in the layouts.

Remix users specify a rerandomization interval. At each interval, the program is rerandomized. A shorter randomization period increases performance overhead. Based on the SPEC benchmark suite, when Remix

Optimization Level	Functions Count	Functions Average Size	Basic Blocks Count	Basic Blocks Average Size
O0	91342	477.40052	974715	44.737917
O2	48991	609.765937	832713	35.874356

Table 10.2: Average basic block and function sizes, in bytes, for the programs in the SPEC benchmark suite when compiled with different optimization levels using gcc version 4.8.4.

Fixed Size	Dollops Count	Dollops Average Size
60	1367454	25.9827
120	900866	36.848951
240	792756	41.191778

Table 10.3: Average code sizes per bundle in the Mixr-prepared versions of the programs in the SPEC benchmark suite with different granularities.

rerandomizes a program once, the average performance overhead is 2.8% with a filesize overhead of 14.8%.

Remix is a significant accomplishment. Its support for user space applications and kernel modules differentiates it from other runtime rerandomizers. Its limited support for applying rerandomization to binary programs is a limiting factor and makes it inapplicable for the context described in Part I and 10.1.

The design decision in Remix to randomize at the level of the basic block does limit the amount of entropy added to the program and limits the user's ability to control the additional security provided by the technique. For SPEC2006 compiled with GCC version 4.8 and optimized at level 2, the average basic block is 35.87 bytes. When compiled at level 0, the average basic block is 44.74 bytes. The feature of Mixr to allow for a user-specified dollop bundle size means that the Mixr user controls the level of granularity of randomization. With Remix (and the other runtime rerandomization techniques discussed below that randomize at the function or basic block granularity) the user is subject to compiler and compiler options. Because the level of granularity for randomization is a key part of the security provided by a runtime rerandomization technique, leaving this choice to the compiler is unacceptable. See Tables 10.2 and 10.3 for a comparison of the granularity when randomizing at function and basic block levels with the granularity achieved using Mixr.

Besides simply quantifying the impact that the compiler has on the granularity of the rerandomizations in systems that fix those boundaries to either basic blocks or functions, it is possible to assess whether those groups of instructions contain code that enables attacks. With a ROP compiler [187] it is possible to perform this analysis. The results depend on the compiler used and the settings employed, but, overall, the more instructions there are the more gadgets. For instance, the version of `bzip2` compiled for the SPEC2006 benchmark according to the configuration described above has 25 functions with at least 10 ROP gadgets. On average, every function in `gcc` contains more than 7 gadgets and one function contains more than 90.

10.6.2 RuntimeASLR

RuntimeASLR is a runtime rerandomization technique developed by researchers at the Georgia Institute of Technology and Saarland University [132]. Their tool targets server software implemented using the traditional fork model. In this model, a server that can support multiple simultaneous connections is controlled by a parent process. As each client connects, the parent process spawns a separate child process to handle that request. The child, worker process is spawned from the parent proess using a `fork()` system call. Much of the popular server software that powers modern Internet services are implemented using this paradigm (e.g., nginx, mysqld).

RuntimeASLR improves a server's defense against ROP attacks by rerandomizing the server's code every time the server `fork()`s to spawn a worker process to handle a client request. The goal is to increase security while minimizing the runtime overhead of the child process whose responsibility it is to actually serve the client's request. These two goal drives their design and implementation.

The RuntimeASLR authors' intuition is that rerandomization can be done quickly, with no restriction, if the memory locations and registers that contain pointers to that code are always known. When the code and data locations are rerandomized, only those locations need to be updated.

Knowing the memory locations and register names that contain pointers is the hard part. To learn this information, the RuntimeASLR authors developed a pointer tracking mechanism that uses algorithms developed for taint tracking. The pointer tracking mechanism starts with taint policy generation. In this step, the target application is executed under the external control of Pin, a software dynamic translator, and repeatedly fed sample inputs. As instructions are executed that modify memory locations or registers, their behavior with respect to adding or removing taint is recorded. Instructions with ambiguous taint tracking semantics are clarified by running the target application multiple times on the same sample input with ASLR enabled. The policies generated by these runs can be intersected with one another to determine the precise semantics for each ambiguous instruction.

At runtime, the parent server process is executed under control of Pin and monitors taint propagation according to the policy developed in the offline step. The initial set of tainted pointers and registers is built from the set of pointers that each program initially contains as a result of pre-execution preparation by the OS. The only subsequent introduction of taint is from system calls. System calls are modeled according to their taint semantics and that model is incorporated into the taint tracking policy.

Under this system, when the parent process `fork()`s a child process to handle a client request, RuntimeASLR knows the location of every pointer in memory and in registers. When the child begins to execute, each of the mapped regions of memory are remapped as a unit using the `mremap()` function and the pointers

are modified accordingly. Once the rerandomization and pointer updating are complete, RuntimeASLR ceases to shepherd the child's execution by detaching Pin so the child can serve the client without instrumentation overhead.

For software that follows the traditional fork model for handling client connections, RuntimeASLR has very little runtime performance penalty because very little work is done in the controlling parent process. However, for software that does not follow this paradigm, the overhead of RuntimeASLR is prohibitive (217x to more than 110000x for certain SPEC applications).

RuntimeASLR is specifically targeted at preventing clone-probing attacks (e.g., BROP attacks). For this specific threat, the tool performs well. However, it is not applicable when servers handle clients in other ways (e.g, polling, multithreading, etc.). In these cases, the pointer tracking runtime implemented with Pin cannot be detached before servicing a client and the overhead continues to accumulate throughout execution.

10.6.3 Timely Address Space Randomization

Timely Address Space Randomization (TASR) is a runtime rerandomization technique developed by researchers at MIT [40]. Their runtime rerandomization technique improves the traditional ASLR protection by rerandomizing code at runtime whenever a program exposes, transmits or outputs any data. By rerandomizing at those points in program execution, any information about code locations leaked by those outputs will be stale by the time an adversary attempts to use that information for an attack.

TASR requires a modified Linux Kernel, GCC and dynamic loader to function correctly. It can add protections to most applications written in the C programming language. It will not work on programs that interpret code (e.g., scripting engines, etc.), will not work on programs that violate certain aspects of the C standard and will not protect against adversaries that leverage relative locations between instructions to mount their attack.

Like Mixr, TASR operates as a pipeline of components. In the first step, the program is recompiled from the original source code using the TASR-enabled version of GCC. The compiler adds metadata to the binary that will be used by later randomization steps and stores that information as pseudo-debugging symbols. TASR leverages the fact that all input and output of an application program is mediated by the Kernel through system calls and modularizes the rerandomization code so that it is not embedded within the application itself. Finally, there is a userspace component that actually performs the randomization when an input or output call is invoked.

At program runtime, the Kernel recognizes a TASR-enabled application by including special ELF information. Besides recognition of such information, loading and startup of a TASR-enabled application is

no different than the same process for a normal application.

When an application invokes a system call to perform input or output, the TASR Kernel support takes over. First, it scans the application for any code that has been added since the last randomization. This handles a situation where the userspace application loaded additional code through dynamic linking. Once the TASR Kernel component recognizes all the code that needs to be randomized, it generates random addresses that will be the new location for the code. Those addresses are fed to the application by Kernel writes into the application's userspace address space. The Kernel then inserts a randomization code segment into the user application address space and passes control to that segment. That segment runs in the context of the application and updates any pointers that refer to the code of the application that is about to be rerandomized. Once pointer manipulation is done, the Kernel regains control and cleans up the user space process by withdrawing the randomization code segment. The final step occurs as the Kernel moves the user space application code into their new locations. This movement is done efficiently by updating page table information instead of performing actual copies.

The authors evaluated the performance of TASR with the SPEC2006 benchmark suite. Average runtime overhead for TASR-enabled processes was 2.1% with a maximum of 10.1%. The RSS of a TASR-enabled program increased on average by 1MB with a maximum increase of 3.5MB. The authors admit that overhead is low on the SPEC benchmarks because of their limited use of input/output – the benchmarks are mostly CPU-bound. Because TASR only randomizes programs when they perform I/O, the number of rerandomizations performed during execution of the SPEC benchmark applications is low.

As proof that TASR actually adds additional security to an application program affected by a memory disclosure vulnerability, the authors modify nginx using TASR and subject it to a BROP attack. Their technique mitigates against this vulnerability. The authors claim further that their technique would prevent the exploitation of the counterfeit object-oriented programming (COOP) vulnerability [190].

TASR is a significant improvement over the state of the art in runtime rerandomization and does add security to applications. However, the significant limitations of the system (modified compiler, accessibility of source code, support for only applications written in C, modified Kernel and augmented dynamic loader) mean that it is ultimately inapplicable to the context described in I and 10.1.

10.6.4 Marlin

Marlin is a code randomization technique designed by researchers at Purdue University and James Madison University to add defenses against ROP attacks to ELF binary programs [89]. It is not accurate to say that

Marlin is a runtime code rerandomization technique, but it shares enough characteristics with the other systems described herein to warrant inclusion and discussion.

The Marlin authors' insight is that traditional ASLR does not provide enough entropy to successfully prevent a ROP attack. To address that shortcoming, Marlin randomizes the target application function-by-function whereas ASLR randomizes only the starting address of the target program as a single unit.

The Marlin code randomization technique randomizes the target application code once at startup at the function level. Marlin is integrated with BASH and works on any ELF binary launched by a user from the shell. The overhead of Marlin is limited because randomization happens exactly once (at program startup) and is amortized over program execution.

Randomization of the program at startup is done according to the following algorithm. First, Marlin identifies the locations of all the functions in an application. Marlin uses an ELF binary's function symbols to perform this identification. Where Marlin is applied to a binary stripped of debugging information, the target application is fed to a unstrip operation that restores the function information. The unstripping tool is provided by a third party – the Paradyn Project. Once the functions are identified, Marlin begins randomization *per se*. The first step is to generate new locations for each of the functions. Once the randomized addresses are generated, Marlin creates a patch table that contains the updated addresses of the functions and, presumably, their extents. Finally, Marlin updates each `jmp` and `call` according to this information by either patching the `jmp` instruction itself or the target addresses stored in the target program's data segment. The authors do not specify how calculated `jmp`s whose targets are not fetched directly from the data segment – i.e., jump tables – are patched. Further, the authors indicate that they do not patch short, relative `jmp`s because the targets of those `jmp`s are always within the same function, something that is not always absolutely true.

On average, Marlin added .87 seconds to program startup for a selection of 131 programs tested by the authors. At the extreme, the gimp application took more than 8 seconds to randomize. The median startup overhead was .53 seconds.

The researchers evaluated Marlin's runtime overhead using the byte-unixbench benchmark suite and executed each application in the suite twenty times and took the average results. Their results show that Marlin added no runtime overhead.

Marlin is an improvement over traditional ASLR defenses thanks to the additional entropy that it provides. However, the fact that it randomizes the program's code only one time at startup limits its ability to defend against a probing attack like BROP. Marlin's reliance on a modified shell to trigger randomization of target program means that applications spawned by a parent process other than the shell will not gain the Marlin's additional protection.

The design decision in Marlin to randomize at the level of the function does limit the amount of entropy added to the program and limits the user's ability to control the additional security provided by the technique. For SPEC2006 compiled with GCC version 4.8 and optimized at level 2, the average function is 609.765937 bytes. When compiled at level 0, the average function is 477.40052 bytes. See Section 10.6.1 for a detailed comparison between the granularity when randomizing at the level of functions and a comparison with the granularity achieved by Mixr. Because the level of granularity for randomization is a key part of the security provided by a runtime rerandomization technique, leaving this choice to the compiler is unacceptable.

10.6.5 Binary Stirring

Binary stirring is a technique developed by researchers at the University of Texas at Dallas [236]. The researchers implemented their technique through a process they call self-transforming instruction relocation (STIR). Like Marlin, STIR randomizes the program once at startup so it is not technically a runtime code rerandomization technique. Like Marlin, it does share enough characteristics of a runtime code rerandomization technique to warrant inclusion and discussion.

STIR randomizes the target application basic block-by-basic block at program startup. STIR works on any binary, with or without debugging or relocation information, for both the Linux and Windows platform.

STIR is designed to minimize as much as possible the runtime overhead of the target application. To achieve a minimum amount of runtime overhead for a target application, STIR has an offline and online phase. In the offline phase, the target application is disassembled, modified and then reassembled. In the online phase, the target application loads a runtime library that performs the rerandomization at program startup and then executes the rewritten binary without any additional interference.

The offline phase is a pipeline of steps. First, the target application is disassembled into its constituent functions and basic blocks. During this process, the superset of all possible program control targets is calculated. Second, the target application's original code segment is duplicated for inclusion in the rewritten version of the application. The copy is marked as read only and is included in the rewritten process for two reasons: to remove the need to patch data access instructions in the randomized version of the target application and for use as a place to store a patch table for computed jump targets at runtime. At the location of each address in the superset calculated in the first step, STIR includes a tag byte (`0xF4`) and enough space to store the address of the updated target address for that byte. The runtime randomization phase fills in the blanks. Third, another copy of the original program's code is made and modified at each place where a calculated `jmp` or `call` is issued. The modifications are required to detect the presence of a tag

byte and to update the calculated target address according to the values in the patch table. Fourth, and finally, the target application is reassembled and written.

The reassembled version of the binary includes two version of the original target application. This resembles a replica-based rewriting technique (see Section 6.8). The version of the original code that contains the patch table and exists for data access is marked read only so that an attacker cannot harvest ROP gadgets from that code. The version of the original code that is modified at each calculated `jmp` or indirect `call` is marked as executable and contains the code that will execute at runtime once randomization is complete.

The online phase is straightforward. At program startup, the randomizer actually stirs the basic blocks, places them in new random locations and records a map between their original locations and their modified locations. The randomizer then uses that map to walk through all the tagged targets in the read-only segment and rewrites them according to their updated addresses.

STIR's offline rewriting phase takes between 30 and 45 seconds per megabyte on Linux and Windows, respectively, on a computer with modern hardware. The rewritten binary is, on average, 73% bigger on disk and uses 37% more space at runtime. The on-disk overhead is less than 100% despite including two copies of the original code because the STIR disassembler can "safely exclude large sections of static data from the rewritten code sections . . . " [236].The authors believe that they can reduce the runtime process size overhead with a more efficient implementation of their randomization implementation. At runtime, a STIRd binary executes 1.6% slower, on average.

STIR is a significant improvement over traditional ASLR. It is also an improvement over Marlin because it works independently of a modified execution environment (i.e., a specialized shell). However, the choice to randomize once at program startup prevents it from protecting the STIRd application from attacks that use leaked information like BROP.

10.6.6 CodeArmor

CodeArmor is a technique developed by researchers at the University of Amsterdam that adds code diversification and runtime code rerandomization to binaries [49]. The combination of diversification and runtime rerandomization "completely decoupl[es] the code pointers stored in memory from the concrete location of their targets" and makes it unique among the other methods described herein. CodeArmor rerandomizes the location of the program code as a unit throughout program execution, hides the addresses of code from potential leaks and inserts traps throughout the code.

The target application is rewritten offline for diversification and instrumentation. The diversified and instrumented version of the original code is called the concrete code space. At runtime there is a second

version of the program's code called a virtual code space. The diversified version contained in the concrete code space includes randomly sized gaps between functions, between the first instruction of a function and its body and between function return points and fallthrough instructions. The diversified version of the original code in the concrete code space is also instrumented at each of the indirect program control points. Each of these indirect program control instructions is augmented with another instruction that adds a constant to the calculated value that will be used as the target of the indirect program control. This offset value is pulled from a global register and constantly updated at runtime to reflect the randomized location of the concrete code space in memory. Therefore, if the program executes a legitimate indirect program control operation, the PC will be properly redirected to the location where the concrete code space currently resides in memory. It is this mechnism that enables CodeArmor to rerandomize without having to track and update the values of code pointers stored in memory, in registers or on the stack. For an illegitimate indirect program control transfer (e.g., one from a return-to-libc or ROP attack), the PC will be redirected to a location in the virtual address space. The virtual address space is filled with nothing but code that will trigger an alert that an attack is underway. Only the concrete address space exists in the program when it is stored on disk. The virtual address space is built by CodeArmor's custom loader at program startup.

At runtime, the virtual address space does not move and resides in memory where the original program code would have resided if not for the offline rewriting. At program startup, the concrete code space is placed randomly and moved as a unit to randomly assigned base addresses throughout execution. Every time the concrete code space is relocated in memory, the offset value is updated. This makes it possible for the program's execution to remain synchronized with the movement of the concrete code space. Randomization of the concrete code space occurs automatically at every system call. When randomization is done, a copy of the concrete code space is moved to a new random base address and the current version is left in place. This prevents the system from failing for processes that execute more than one thread at a time. One by one as a program's threads reach a quiet state, the threads are transferred to use the updated offset value. Once all threads have moved to the updated offset value, the old concrete code space is removed and the new concrete code space becomes the current version. At this point, the cycle repeats.

Completely separating code pointers stored in memory from the true location of the target prevents an attacker from using leaked addresses to mount a targeted attack. That does not prevent the attacker from hijacking control to random addresses and observing behavior. This attack is prevented through the diversification that CodeArmor adds to the program code stored in the concrete code space. The diversification makes it statistically unlikely that transferring control to a random address will expose a gadget that is useful to an attacker.

CodeArmor imposes very limited overhead at runtime. Based on the SPEC2006 benchmark and a set

of tests against nine common server applications, a CodeArmor'd program operates 3.2% and 8.2% slower, respectively. Based on the SPEC2006 benchmark and a set of tests against nine common server applications, the runtime memory usage (RSS) of a CodeArmor'd program increases 4.4% and 13.4%, respectively.

CodeArmor is a significant technical accomplishment. It provides security to applications without access to their source code and without a runtime. However, there are two shortcomings. First, it requires a modified dynamic loader to operate. Second, the program is randomized as a unit. For these reasons CodeArmor is ultimately inapplicable to the context described in I and 10.1.

10.6.7 STABILIZER

STABILIZER is a runtime rerandomization technique developed at the University of Massachusetts, Amherst [62]. Although not designed specifically for the purpose of improving the security of software, it does add runtime rerandomization to a target application.

STABILIZER was designed to help researchers perform rigorous testing of the performance of software. The researchers hypothesized that performance is often dominated by code, stack and heap layout at runtime. In order to control for the effects of different layouts and isolate the performance of the optimized algorithms and code themselves, STABILIZER randomizes these components at 500ms intervals throughout program execution.

STABILIZER has an offline and an online component. STABILIZER's offline component requires access to a modified version of the LLVM compiler and the target application's source code. The online component requires an additional randomization library be linked with the target application.

STABILIZER randomizes the layout of the stack at frame boundaries. In the offline phase, STABILIZER calculates a random amount of padding to insert after the stack frame for every function. The offline component adds code to the function's preamble and epilogue to account for that padding. During program execution, the online component refills those pad spaces with random data.

STABILIZER randomizes the heap by allocation sizes. STABILIZER creates buckets of fixed-size heap space to satisfy dynamic allocations. For an allocation from a particular bucket, a prior allocation is selected and swapped with the results of a new allocation. After swapping the contents of the memory at the prior allocation point with the new allocation's point, the prior allocation point is returned to the caller. The process is reversed for deallocation. The location of a previous allocation is selected and the contents of that allocation replace the contents of the allocation to be freed. The result is that the previous allocation is the memory actually freed and internal bookkeeping is updated to reflect this.

STABILIZER randomizes the layout of code at function boundaries. In the offline phase, STABILIZER transforms functions and bundles each with a relocation table. At runtime, these bundles can be moved without affecting their correctness. Every 500ms, the runtime randomization system replaces the first instruction of every function with a trap. The trap initiates the randomization process for that function. Once a function is randomized its position is fixed for the remainder of the 500ms interval. At the next interval, the first instructions of every function is again replaced with a trap instruction and the process restarts.

At runtime, STABILIZER adds 6.7% overhead.

Although STABILIZER was not designed specifically to add security to an application, the technique does do just that. However, the requirement that the user recompile the target application from its source code using a modified version of a compiler limits its applicability.

10.6.8 Chronomorph

Chronomorph is a runtime code rerandomization technique developed by researchers at Smart Flow Information Technologies [82]. The technique combines selective runtime rerandomization with code diversification to add security but keep runtime overhead low. The authors present their technique independently of any particular architecture, but evaluate its performance for Linux-based x86 software applications.

Chronomorph's code diversification component and selective runtime rerandomization make it unique among the other techniques described herein. At runtime, only the basic blocks that contain attack gadgets are rerandomized. Chronomorph diversifies the code around function prologue and epilogue and replaces certain instructions with their semantic equivalents. In function prologue and epilogue, Chronomorph permutes `push`/`pop` operations to change the stack arrangement. Chronomorph replaces certain instructions with their semantic equivalents by 1) replacing a single instruction with multiple instructions that accomplish the same effect or 2) swapping the order of individual instructions in groups that achieve the same result no matter the other they are executed.

Chronomorph consists of an offline and online phase. Offline, Chronomorph uses a ROP compiler to analyze the binary and calculate the location of attack gadgets. The location and extents of the basic blocks surrounding those gadgets are marked in the rewritten binary and used by Chronomorph's runtime rerandomization engine. Chronomorph adds relocation space to the rewritten binary. This relocation space is where the marked basic blocks containing attack gadgets will be randomly placed during runtime. Finally, Chronomorph adds so-called morph triggers to the rewritten binary. These are places in the execution of the target application that will invoke a rerandomization.

Online, the instructions at the head of basic blocks containing the identified attack gadgets are replaced with `jmp` instructions. The basic blocks are then placed randomly in the relocation space and the `jmp` instructions are updated according to those placements. When the program reaches a morph point, the Chronomorph runtime randomizes all the basic blocks in the relocation space and updates the `jmp` instructions accordingly. Maintaining the link between the original addresses of basic blocks and their locations after rerandomization allows Chronomorph to rerandomize without having to track and update code pointers stored in memory, in registers or on the stack.

The authors rightly conclude that the more times the target application reaches a morph point, the greater the runtime overhead. However, they do not offer any systematic performance evaluation. They test their tool on a single desktop application and cite a small performance overhead of less than 2%.

As mentioned previously, the combination of selective runtime randomization and offline code diversification makes Chronomorph different from the other tools described herein. However, there are several restrictions that limits its applicability. The basic blocks that contain attack gadgets can only be relocated if they are 1) bigger than a `jmp` instruction, 2) do not contain a `call` instruction and 3) end with an indirect control flow operation (e.g., `ret`). The size of the relocation space is fixed which could limit the number of basic blocks that Chronomorph can relocate at runtime if the number and size of the basic blocks containing attack gadgets is larger than that space. Finally, reordering `push`/`pop` operations in function prologue and epilogue can only be performed for functions that are not reentrant.

10.6.9 Shuffler

Shuffler is a runtime ASLR technique developed by researchers at Columbia University and the University of British Columbia [243]. Shuffler rerandomizes the program during execution at function boundaries once every 50ms. Shuffler is designed to prevent code the crippling ROP, BROP and JIT-ROP code reuse attacks. At the same time that Shuffler protects a target program, the Shuffler runtime support system protects itself. Shuffler protects programs without requiring access to their source code and does not require changes to the hosts compiler or kernel. To operate, it does, however, require that the target application have preserved debugging and metadata information.

The Shuffler rerandomization technique is similar to a high performance graphics system that relies on double buffering. Once every 50ms, an alternate, randomized version (the *in-shuffle* version) of the program is prepared asynchronously. After the new version of the program is created, execution switches to it and releases the old version. During the next rerandomization cycle, the old version of the code is rerandomized and becomes the running version of the program once the shuffle process is complete.

Rerandomization is done in parallel with the execution of the secured program in a separate process. The running and in-shuffle versions of the code are stored in a sandbox. The sandbox is divided equally in two so that the running version and the in-shuffle versions are always separate. The running version of the code sandbox is hardware protected to be execute/read only while the in-shuffle version is hardware protected to allow writes.

Shuffler is unique among the runtime ASLR techniques surveyed herein because it protects itself. The separate process that performs the asynchronous rerandomizations is itself being rerandomized. Shuffler relies on a bootstrap process to achieve this.

The secured program is only paused when transitioning between the in-shuffle and running versions of the code. At the time of transition, the secured program is paused and the stack is unwound. Addresses on the stack of the secured program that point to code are rewritten with the updated addresses of that code after the most recent shuffle. Once this process is complete, the secured version of the program is restarted.

Shuffler works on programs in their binary form and requires no changes to the compiler or kernel. There are limitations to the system, however. In order to prepare a program offline for runtime rerandomization, Shuffler requires that the target program be compiled to "preserve the symbol table . . . and . . . relocations." Shuffler also requires that the binary program contain DWARF[3] debugging information.

Shuffler uses the symbol table and relocations during its offline preparation phase. In this phase, direct function calls and instructions that use the address of functions as immediates (e.g., `leas`) are converted to function calls that indirectly transfer control to their targets and instructions that indirectly load function addresses. The indirect calls and loads are done through the *code pointer table*. The code pointer table is a table of pointers to the location of functions in the randomized, secured version of the program. In other words, the code pointer table can always be used to find the locations of functions at runtime and allows Shuffler to rerandomize without updating the value of code pointers in memory, in registers or on the stack. This is similar to the dollop table and the DE tables in a Mixr-prepared program.

The DWARF information is used to properly unwind the stack of the secured program when Shuffler forces a transition between the running and in-process versions of the code. At runtime, the stack can contain pointers to code. This is commonly the case when the stack contains the return address of functions called previously. It may also occur when pointers to callback functions are taken as parameters to the functions. When Shuffler transitions the secured program to a newly randomized version, the pointers on the stack must be updated to reflect the new locations of the targets. To perform this update, Shuffler unwinds the stack using the DWARF information and rewrites the addresses where necessary.

[3]DWARF was originally a play on the name of an associated format, ELF, and a meaning was later attached, Debugging With Attributable Record Formats [67].

As a measure of further protection, during its offline preparation phase, Shuffler adds code to the prologue and epilogue of the target program in order to encrypt return addresses as they are stored on the stack. Each return address is encrypted with a key that is stored in a global memory location. Shuffler stores the key in the same global memory space as canary values are stored by systems that detect stack smashing attacks by checking stack canaries [57]. The authors do not address whether this means that Shuffler and stack canary implementation systems are incompatible but it seems unlikely that the two technologies can interoperate.

The authors evaluate the security and performance of the Shuffler system using a battery of tests, from SPEC to other application-specific benchmarks. For SPEC, Shuffler exhibits 14.9% performance overhead. In order for Shuffler to work on the programs in the SPEC benchmark suite, the authors have to compile the programs with the special flags required by Shuffler to maintain the system table, DWARF debugging information and relocations. The authors further evaluate Shuffler on nginx, MySQL and SpiderMonkey, a webserver, database server and Javascript runtime engine, respectively. They use application-specific benchmarks to demonstrate that Shuffler operates with low-overhead and prevents ROP, BROP and JIT-ROP attacks.

Overall, Shuffler is a remarkable system that protects programs with very little overhead. The preparation and runtime phases of Shuffler are very similar to those of Mixr. Shuffler, however, does not work on programs that are stripped of their debugging information and other metadata. The authors argue that most programs are distributed (or could be distributed) with this information without a burden. However, this does not fit with the assumptions described in Part I. More importantly, Shuffler does not offer its user the choice of granularity for rerandomization – it only rerandomizes at function boundaries. As mentioned previously, this exposes a significant number of ROP gadgets that the adversary can exploit using relative addresses.

10.6.10 librando

librando is a runtime diversification technique developed by researchers at the University of California, Irvine that is applicable to JIT compilers [101]. Their technique diversifies the native code generated by JIT compilers without requiring changes to the compilers source code. Thanks to the additional diversity, the code generated by the JIT compiler and modified by their technique is better protected against JIT spraying and ROP attacks. Although their technique does not actually rerandomize the location of the generated code at runtime, their framework makes adding such a feature straightforward.

Their system works in either a blackbox or greybox mode. The blackbox mode requires no modification to the JIT compiler. The randomization works by setting the code cache as non executable. When execution transitions to undiversified code (which is stored in those non-executable sections) then a segmentation

fault signal is generated. librando catches that signal, diversifies the code at the target and then transfers control to that version of the code. There are many subtleties with which librando has to deal in order for it work properly. For instance, librando has to make sure that process registers are maintained, that the stack contents are maintained (including pointers to undiversified code), and that signals are handled correctly. It also has to work around JIT compiler optimizations that make the correct implementation of librando more difficult. For instance, JIT compilers garbage collect generated code that is no longer needed. In order to detect the fact that the space used by previously generated native code is being reused, librando marks the pages that contain rewritten diversified code as read only. The OS raises a segmentation fault signal when the JIT compiler attempts to write those blocks with updated native code. Because librando intercepts all those signals, it gets notified and takes the proper action. There are details in this as well with which librando has to attend.

The second mode of operation is greybox mode. This requires some knowledge of the compiler and requires access to and changes in the source code. This is not exactly the same as whitebox integration. Although the authors actually call it whitebox.

No matter the mode of operation, librando adds diversity to the native code generated by the JIT compiler. The diversity is the result of NOP insertion and constant blinding. For NOP insertion, the generated code is peppered with `nop`s between instructions. For constant blinding, the constants that are generated for generated code are translated and blinded through a cookie value. Normal implementations use an XOR operation to accomplish this translation but XOR changes the flags – this is not acceptable for librando. librando instead use an `lea` to perform the translation.

To test and measure librando, the authors benchmarked V8 and HotSpot. For librando running on v8 in blackbox mode with all diversification enabled (and all implementation optimizations), the average overhead was 3.5x. For tests that show the least impact from librando, a small number of diversified blocks are generated once and execution is concentrated within those blocks. The blocks consist mostly of numeric calculations.

For librando running on HotSpot in blackbox mode, the authors tested librando against a subset of the benchmark programs for Java in the Computer Language Benchmarks Game. Based on their tests using fannkuchredux, mandelbrot, nbody, and spectralnorm, they report that librando introduces 15% overhead. However, those four tests are not the only four in the benchmark suite. They separately include results for another one of the tests (binarytreesredux) and report that librando introduces a 70x performance overhead.

librando is very significant research and is the first tool that allows the runtime diversification of code generated by a JIT compiler. The fact that it works without requiring modifications to the JIT compiler and performs with reasonable overhead is impressive. Although the implementation presented in the research

does not include runtime ASLR *per se*, the underlying framework makes the addition of such technology relatively straightforward. Because it does not address the issue of rerandomizing compiled software, however, it is ultimately inapplicable to the context described in I.

10.6.11 selfrando

selfrando is a runtime ASLR technique developed by researchers at Immunant, The Tor Project and Università degli Studi di Padova, among others, that rerandomizes a program at function boundaries at startup [53]. selfrando adds the runtime ASLR defense to programs that can be recompiled from source using an augmented linker. Although there is a requirement that the program be recompiled, no source code changes are necessary. Nor are changes required to the compiler, loader or kernel. The goal of selfrando is to improve the traditional ASLR by adding entropy to the program by rerandomizing the vulnerable program at startup along function boundaries.

selfrando is implemented in offline and online components. The offline component is implemented as an augmented linker. The augmented linker is implemented as a drop-in replacement for the system loader. The build process for the vulnerable software does not need to be modified to accommodate this linker.

The linker adds metadata to the compiled binary that is used at runtime to perform the actual rerandomization. This information, know as the Translation and Protection (TRaP) information, includes the boundaries of all the functions in the program and the location of references (calls or pointers) to those functions. TRaP information is loaded as a new segment to the end of the protected binary and referenced by an ELF segment header at the front of the recompiled program. Before the linking process is complete, the runtime component of selfrando is linked to the target program and the programs header is rewritten to point to that library as the first function to execute when the program is launched.

The library and this function constitute the online phase of selfrando. When the program begins execution, the selfrando library uses the TRaP information to locate all the functions. Each of those functions is moved to a new address. The pointers and calls to that function (also contained in the TRaP information) are updated to reflect this new information. Further, the DWARF information (used by exception handlers) is updated as well so that unwinding in the presence of exceptions is performed correctly. As a final detail, in order to handle the address sanitization (ASan), selfrando generates a map file to augment ASans symbolization procedures. selfrando replaces the native symbolization function with a custom version that consults the map file when doing symbolization.

By virtue of rerandomizing at function boundaries rather than segment/module boundaries, the authors reason that selfrando makes it so that an adversary would have to guess "...at least 39 bits [for the smallest

programs], while for the biggest, the attacker needs [to guess] at least 78 bits."

For testing, the authors used the Tor browser, the SPEC2006 benchmark suite and common Javascript benchmark tools. They also tested integration of selfrando with some common applications like BASH, Chromium, etc. to make sure that it works on real-world applications.

To test the overhead of selfrando on program startup, the authors tested load time performance for the Tor browser and found that there was about 17% increase.

For SPEC2006, the authors ran all the benchmark suite applications after compiling with clang and GCC. Running the applications of SPEC2006 with all self-randomization enabled, "[t]he geometric mean of the positive overheads is 0.71% for GCC and 0.37% for Clang." To measure the load time overhead of selfrando on the SPEC benchmark applications, they ran the applications according to an identity transformation – the functions are left in the original order but all the code is run. They found that the load time overhead is negligible. Finally, they used SPEC2006 to test the memory overhead of selfrando and found it to be less than 1% on average, although there were outliers (the small applications of the benchmark suite).

In order to test the performance of selfrando on Javascript interpreters, the authors ran the Tor browser without JIT and against three standard benchmarks. The results indicate that selfrando introduced an average performance overhead of 2.5%.

Finally, to determine whether selfrando works on actual applications, the authors tested against BASH, less, socat, nginx, thttpd. Each of the applications continued to perform normally after being secured with librando.

selfrando is a significant technical achievement. That it is incorporated into a production application used by many people (i.e., the Tor browser) makes it unique among the other runtime rerandomization techniques. However, the fact that it requires access to the target programs source code makes it inapplicable to the context described in I.

10.7 Conclusion

Mixr is a system that can add the MTD of runtime ASLR in a way that existing runtime ASLR systems cannot. Mixr is implemented using Zipr, the prototype implementation of the architecture and algorithms of the static binary rewriter described in this dissertation, and works on SOUP. A program secured with Mixr can run on the same platform as the vulnerable version without changes to the compiler, linker, loader or kernel. Because the Mixr user is able to customize the rerandomization intervals and the boundaries upon which to rearrange program instructions, it provides a flexibility that the other systems do not. This flexibility represents a tradeoff. The possibility exists that the potentially vulnerable target application cannot be

accurately disassembled making Mixr's protections inapplicable. Furthermore, the runtime overhead of a Mixr'd program is non-trivial.

Although the focus of this chapter is on Mixr's ability to add an MTD of runtime ASLR to a vulnerable program/library, Mixr is not limited to this application. In the future, extensions to Mixr can be used to implement selective runtime randomization and artificial noise injection.

Mixr is an example of the way that the static binary rewriter whose architecture and algorithms are described in this dissertation can be applied to SOUP to add security. In the following chapters, two additional applications will be described that also add security to SOUP. Together, the three applications demonstrate the utility of the static binary rewriter in adding security to software and validates the hypothesis outlined in Part I.

Chapter 11

Dynamic Canary Randomization

11.1 Introduction

Stack canaries are a well-known, effective technique for detecting stack overflow attacks and GCC has supported their use for more than ten years [57]. At function invocation, a secret value, the *canary*, is placed on the stack between the caller's saved return address and the space allocated for the callee's local variables. Before a function returns, the canary value on the stack is compared to the program's reference canary value. A mismatch indicates that the program wrote beyond its allocated boundary on the stack. Upon detection of such a condition, the program halts. If the canary value was modified by an attacker attempting to hijack control of the target program by overwriting the return address, his/her efforts are thwarted. If the value of the reference canary is leaked to the attacker, its protection becomes ineffective. The attacker can use the leaked canary value to craft their attack payload so that the stack always contains a matching canary value.

For programs compiled using GCC, the reference canary value is randomly generated at program invocation and fixed throughout execution. Moreover, for software running on the Linux OS, canary values are inherited from the parent process and only changed if/when the child process `exec()`s a different program. Researchers and others have exploited these behaviors to craft real-world attacks that bypass the protections of stack canaries. For instance, researchers have discovered that it is possible to infer the canary values for every process on a device running the Android OS [115]. Researchers have also developed the BROP technique that bypasses the protection offered by canaries in server software that follows the accept-fork paradigm to process multiple clients simultaneously (e.g., *nginx*, *mysql*) [42]. As of July 2017, *nginx* alone operates 29% of the top 1 million most popular websites on the Internet [158] thus attacks of this class are a significant threat.

Dynamic Canary Randomization (DCR) is a technique for rerandomizing stack canaries at user-defined

rerandomization points during program execution and can be applied directly to binary programs using the static binary rewriter designed according to the architecture described in this dissertation. DCR is an improvement over existing canary rerandomizers because it allows rerandomization to be applied at any point during execution and at any frequency. The evaluation shows that DCR prevents the leaked value of a canary from being used to launch an attack against target software and shows that it defends against BROP attacks (Section 11.4.3).

11.2 Design

The DCR *design* is applicable to binary programs and libraries without requiring access to their source code or debug information. The DCR *implementation* described in this chapter is applicable to binary programs that have been compiled with GCC with stack smashing protection (SSP) (`-fstack-protector`). Comparing GCC SSP with DCR helps explain the latter's design while highlighting its innovation with respect to the former and other existing security techniques.

SSP is GCC's term for instrumenting a program with stack canaries. In some modern Linux distributions, GCC defaults to this option. GCC SSP inserts code into the function prologue and epilogue. In a function's prologue, code is inserted that copies the reference canary value from thread local storage (TLS) to the program stack. In a function's epilogue, code is inserted that compares the stack's canary to the reference canary value. If these values do not match, the program invokes a handler that terminates itself.

DCR uses a similar mechanism: It inserts code in the function prologue that places canary values on the stack. In the function epilogue, it inserts code that verifies that the value has not changed. DCR and GCC SSP are so similar that, in fact, DCR is implemented by using Zipr, the prototype implementation of the architecture and algorithms designed in the course of this research, to replace the canary handling instructions emitted by GCC with its own handling instructions and reusing the TLS space GCC allocates for the reference canary value. Like GCC SSP, DCR keeps the reference canary value in TLS. In DCR, like in GCC SSP, an executing program has only a single reference canary value at a time during program execution.

The differences between DCR and GCC SSP are the reference canary value(s) and the canary values stored on the program stack. Because GCC SSP's holds one unchanged, reference canary throughout program execution, every copy of the canary value on the stack always matches the reference canary, as long as the program has not been attacked. At user-defined randomization points, DCR changes the reference canary value *and* the canary values stored on the program stack. If DCR only updated the reference canary value, the canary values on the program stack at the time of rerandomization would no longer match the reference

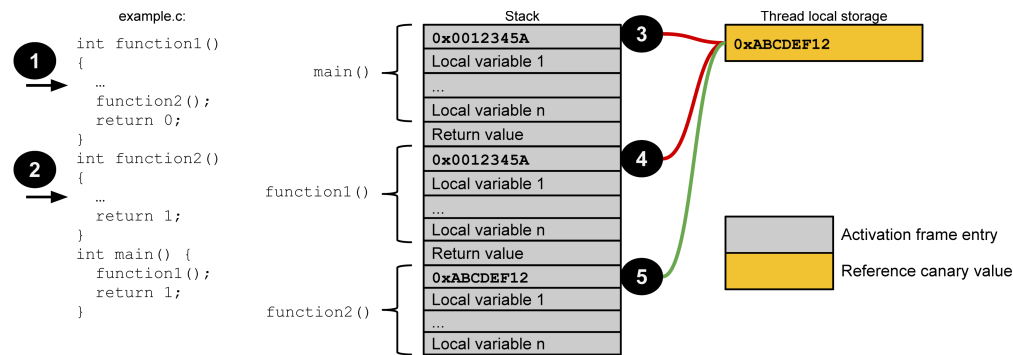

Figure 11.1: All canary values on the stack must be rewritten when the reference canary value is rewritten.

canary value. As a result, the protected program would incorrectly detect a canary mismatch when comparing the reference and actual canary values in the epilogue of functions invoked prior to rerandomization.

Consider the example in Figure 11.1. The figure shows the contents of the stack when it reaches the statement labeled (2). The DCR user directed canary randomization to occur at the statement labeled (1). The figure shows the problem when all canary values on the stack are not rewritten at the rerandomization point. Although the rerandomized reference canary value is placed on the stack at the invocation of `function2()` (5), the canary values already on the stack (3, 4) no longer match the reference canary value. In this case, DCR would detect a stack overflow error in the epilogue to `function1()` where one does not exist and incorrectly terminate the program.

To facilitate rewriting every canary value on the stack at runtime, DCR embeds offsets and addresses in the values of canaries on the stack and stores an additional offset alongside the reference canary value in the TLS space. These values are used by DCR to build a linked list of canaries on the stack at a given point during program execution. The offsets stored in a canary value on the stack point to the previous stack canary. A pointer to the head of the list is embedded in the reference canary value. See Figure 11.2.

In Figure 11.2, the stack growth is visualized from the top of the figure to the bottom. Time a is before time b; time b is before time c; etc. The reference canary value is `0x123456`. The numbers in green in the reference canary value are the offsets to the head of the stack canary linked list.

When a function is invoked at time a, the reference canary value is copied to the stack at address `0x7777f0f00`. The reference value is updated to "point" to that stack location, `0000` in this case. When a function is invoked at time b, the reference canary value is copied from TLS. Its offset is updated to the difference between the current stack address and the offset embedded in the reference canary value. The result is pushed on the stack at address `0x7777f0ee8`. The reference canary value's offset is updated to point to the top of the stack, `00018` in this case. On completion of a function, this process is reversed and the canary value is verified. If DCR detects that a canary value is modified, the program terminates.

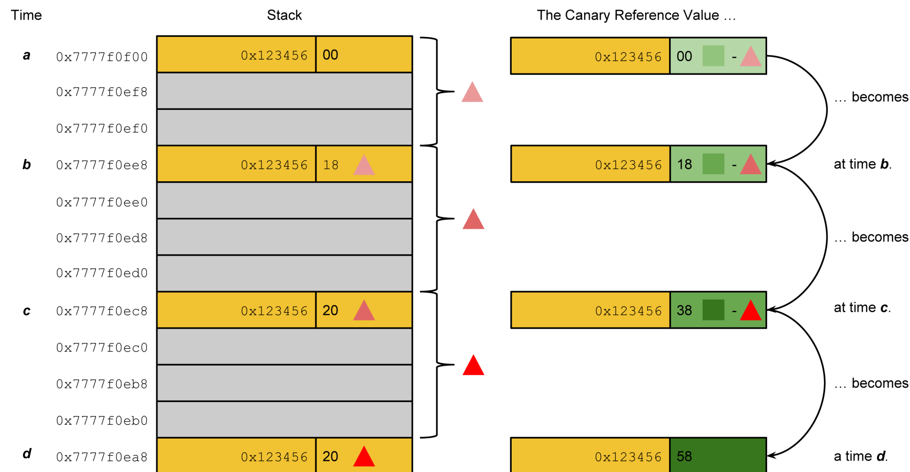

Figure 11.2: Building the linked list of canary addresses on the stack at runtime.

Using these calculations, the program can retrieve the head of the stack canary linked list at any point during program execution by accessing the reference canary value and using the embedded offsets in the canaries on the stack to iterate through every canary value. At the point where a user-directed rerandomization occurs, DCR iterates through the linked list of canary values and rewrites the entries. Because unsafe values may be latent on the stack (from buffer overflows that occurred in previously invoked functions), DCR verifies the canary values as each one is rewritten. If there is a mismatch, the program immediately terminates. This check is also necessary to verify the integrity of the offsets in the stack canary linked list.

11.3 Implementation

Again, the design of DCR is applicable to binary programs and libraries without access to the source code or debug information but the implementation of DCR described in this chapter is applicable to binary programs compiled with GCC SSP. Because the DCR implementation relies simply on the pattern of instructions that GCC SSP emits to prepare the canary values in functions' prologue and epilogue, other compilers could easily be accommodated. DCR is implemented as a User-specified transformation for Zipr, the prototype implementation of the architecture and algorithms for static binary rewriting described in this dissertation, that modifies the IR of the input binary program. Zipr alters the input program accordingly and generates an updated binary program. The resulting binary program executes without any additional runtime support on exactly the same platforms as the original program.

When DCR is applied to a program, its user specifies a particular function whose invocation will trigger the randomization. For instance, the DCR user may specify that rerandomization occur every time that

`read()` is called. In the future, it is possible to add support for the DCR user who wants to randomize canary values after every n instructions or every s seconds.

The implementation of DCR has been tested against SPEC2006 and found to be robust enough to support the programs in that benchmark. The implementation has also been tested against *nginx*, a well known, open-source webserver. See Section 11.4.

11.4 Evaluation

This section describes the process used to evaluate the performance impact of DCR and its ability to add additional security to vulnerable software. Performance impact is measured using the industry-standard SPEC2006 benchmark suite. Each experiment is a configuration and execution of the entire suite of SPEC2006 benchmark applications.[1] The security aspect of DCR is demonstration by its ability to thwart a real-world attack.

All experiments were performed on a server-class host containing two 2.80GHz Xeon processors, each with 10 cores, and a total of 50GB of RAM. The host runs Ubuntu 14.04 LTS with Kernel version 3.13.0-30. With the exceptions noted below to configure SSP, all executions of SPEC were performed using the default compilation parameters and optimization at level `-O2`. Each application in the benchmark suite was executed once on reference input.

11.4.1 Steady-State Performance Overhead

A comparison of the *steady-state overhead* incurred by stack canaries implemented by the GCC compiler and DCR isolates the performance impact of DCR. Steady-state overhead is the performance penalty for simply maintaining canary values at runtime. For GCC's SSP this is the cost of pushing canary values on to the stack in the prologue and checking their values in the epilogue. For DCR, this is the cost of pushing canary values on to the stack in the prologue, checking their values in the epilogue *and* maintaining the linked list. The overhead of rerandomizing the canary values at different intervals is studied in Section 11.4.2.

As mentioned previously, DCR is implemented using Zipr. The experimental methodology is designed to isolate the overhead of the static rewriter from the overhead of DCR and contrast that with the overhead introduced by GCC's SSP.

It is possible to enable SSP in GCC in several ways. On most modern OS platforms, GCC defaults to protecting any function that calls `alloca()` or uses "buffers larger than 8 bytes" [2]. The user can force GCC

[1] Two of the 29 benchmark applications are not included because they did not run on the host platform when compiled and linked with the standard toolchain.

to add SSP to every function no matter the size of the local buffers or the means of memory allocation. In this evaluation, former is referred to as *selective SSP* and the latter as *complete SSP*.

Six experiments were performed to conduct the assessment.

Experiment 1 Execution without SSP: An execution of SPEC2006 with SSP disabled (`-fno-stack-protector`). The baseline for Experiments (2) and (3).

Experiment 2 Execution with selective SSP: An execution of SPEC2006 where GCC determines the functions to protect (`-fstack-protector`) [2].

Experiment 3 Execution with complete SSP: An execution of SPEC2006 where GCC is configured to protect all functions (`-fstack-protector-all`).

Experiment 4 Execution with static rewriter, no transformations with selective SSP: An execution of SPEC2006 where the application binaries are compiled with selective SSP and rewritten with the static rewriter. Comparing these results to the results of Experiment (2) isolate the overhead of the static rewriter.

Experiment 5 Execution with DCR and selective SSP: An execution of SPEC2006 where the application binaries are rewritten to add DCR. The application binaries themselves are compiled with selective SSP. Comparing these results with Experiments (2) and (4) isolate the steady-state overhead of DCR when SSP is selectively enabled.

Experiment 6 Execution with DCR and complete SSP: An execution of SPEC2006 where the application binaries are rewritten to add DCR. The application binaries themselves are compiled with selective SSP. Comparing these results with Experiments (3) and (4) will isolate the steady state overhead of DCR when SSP is fully enabled.

Experiments (1), (2) and (3) show the overhead of GCC's SSP. Experiment (4) isolates the overhead of the static rewriter. Experiments (5) and (6) show the steady-state overhead of DCR. Results are shown in Figures 11.3 and 11.4.

On average, GCC's selective SSP adds negligible (.38%) overhead while GCC's full SSP adds minimal (2.89%) overhead. On average, the static rewriter incurs 24% overhead while DCR's selective and full SSP adds .045% and 13.38%, respectively. While a 24% overhead may seem high, stack protection applied directly to the binary offers several benefits: 1) no source code is required for DCR, 2) DCR is usable on programs written in different programming language binaries and 3) DCR requires no changes to fragile and complicated build processes.

Finally, the overhead of the static rewriter is independent of the DCR technique which means that improvement to the rewriter itself will translate directly to improvements to the runtime performance of

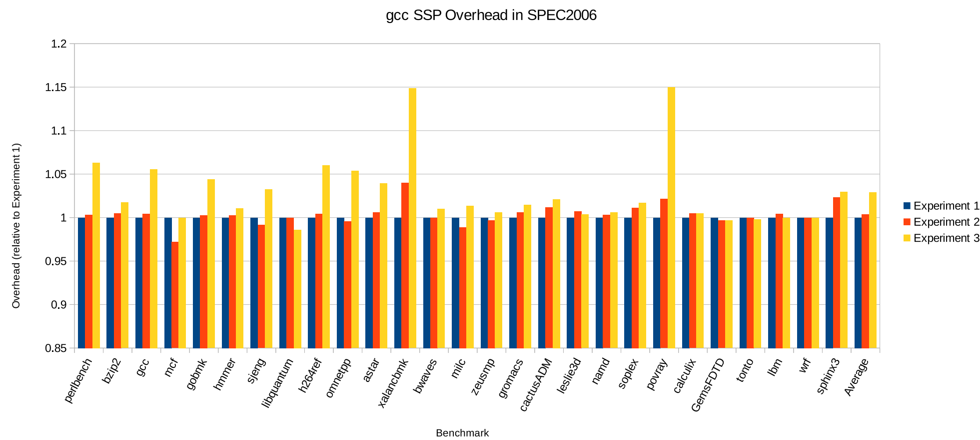

Figure 11.3: Overhead of GCC's SSP in SPEC2006 benchmark. Experiment (1) is the baseline (no stack protection). Experiment (2) is selective SSP, and Experiment (3) is full SSP.

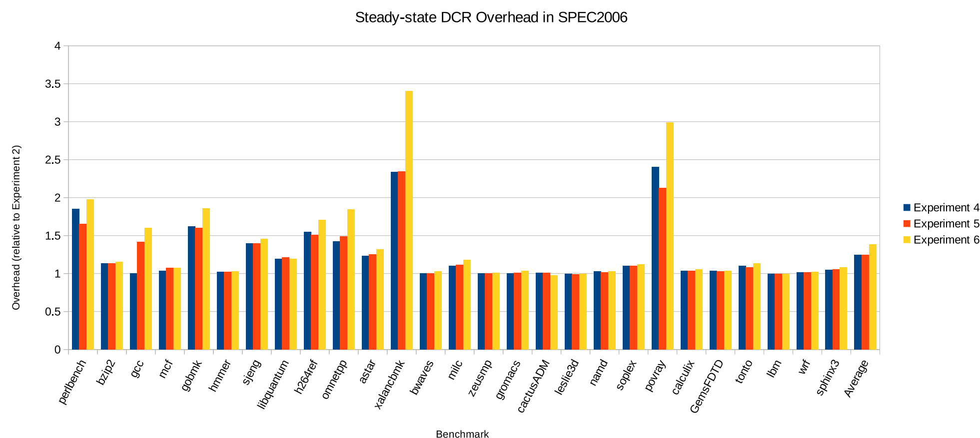

Figure 11.4: Overhead of DCR's steady-state performance in SPEC2006 benchmark. Experiment (4) is static rewriting only. Experiment 5 is selective DCR applied with static rewriting. Experiment (6) is full DCR applied with static rewriting.

DCR. These results were gathered without the benefit of the optimizations introduced in Part III of this dissertation which showed that the runtime overhead of statically rewritten programs can be significantly improved. Research subsequent to that presented in this dissertation show that even further improvements to the runtime performance of statically rewritten programs/libraries is possible [96].

11.4.2 Case Study: Rerandomizing Canaries in bzip2

Besides steady-state performance overhead, DCR introduces runtime overhead every time that the canary value is rerandomized. The more often canaries are randomized, the more overhead imposed by DCR. On the other hand, the more often canaries are rerandomized, the less time is available for an attacker to take advantage of any data leaks from the application. See Section 10.2.1 for an in-depth discussion of applicable

threat model and the terminology of *oracles*, *queries* and *attack window* and the importance of minimizing the attack window for added security. This security-versus-performance tradeoff is fundamental to the DCR method.

This section describes a study of DCR's overhead when protecting a particular Internet-facing web application. A study of the most effective way to balance the performance overhead of DCR with the additional security it provides will be studied in the future. See Section 11.5.

Consider a cloud infrastructure provider whose web-based configuration interface allows the user to upload compressed configuration files. Those files are uncompressed and then parsed and used to configure a virtual host. Assume that the configuration files are compressed and decompressed with a daemonized version *bzip2* that is compiled with SSP. This version of *bzip2* accepts input and commands (i.e., whether to compress or decompress) over a socket and writes output back to the user over the same connection. If there is a stack overflow vulnerability in *bzip2* that leaks the stack canary values (the oracle), then an intruder could build a specially crafted compressed file to a) retrieve the stack canary value (queries) and b) launch an attack on the cloud provider.

For such an attack to succeed, the attacker relies on the fact that, once learned, stack canary values remain consistent throughout program execution. Applying DCR to the *bzip2* program that decompresses the configuration file would prevent such an attack.

Recall that the rate at which *bzip2* rerandomizes its stack canary values determines the period of time that an intruder can use a leaked canary. As the rerandomization rate increases, the period of validity decreases, and vice versa. Again, managing this tradeoff is important and something that will be studied in detail in the future (see Section 11.5).

To secure this hypothetical web application from this particular threat vector, DCR is configured to rerandomize canary values every time that *bzip2* invokes `memcpy()`. The overhead associated with this choice is quantified using the version of *bzip2* from SPEC2006.

For the reference test input set to *bzip2* in SPEC2006, there are 1704322 invocations of `memcpy()`. The number of canary values updated at each randomization point depends on a) the depth of the stack of the running program at that time and b) the number of stack frames that contain a canary. By default for the version of the compiler used to generate *bzip2* in this experiment, only certain functions are protected with stack canaries[2]. In other words, DCR may not rewrite canary values at every rerandomization point. In this experiment, there were 1704286 canary values updated, for an average of 0.9999 canary updates per randomization.

`memcpy()` is obviously not the only potential oracle in the vulnerable *bzip2* application. Since the adversary could exfiltrate data written to standard output or the filesystem, an additional randomization point was set

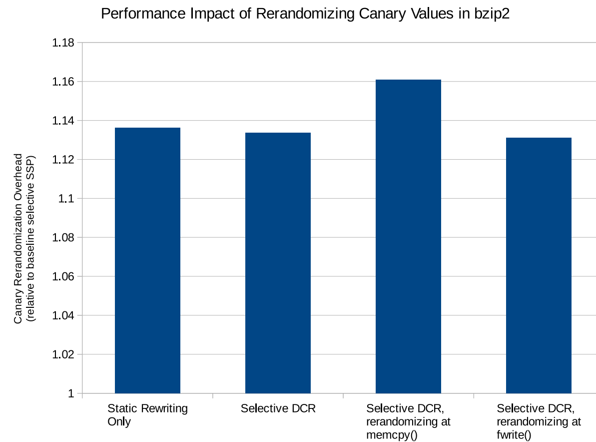

Figure 11.5: Performance impact of rerandomizing canary values at every `memcpy()` and `fwrite()` during an execution of *bzip2* under SPEC2006.

at every `fwrite()` call. For the reference test input set to *bzip2* in SPEC2006, there are 29826 invocations of `fwrite()` and there are 10070 canary values updated which yields 2.96 canary updates per randomization.

Four different executions of the SPEC2006 version of *bzip2* were used to isolate the performance overhead of canary rerandomization in DCR. The first execution measured the speed of the *bzip2* without any modifications. The second execution measured the speed of *bzip2* compiled with selective SSP after it had been statically rewritten (equivalent to Experiment 4 in 11.4.1). The third execution measured the speed of *bzip2* after it had been statically rewritten and modified to include a configuration of DCR that rewrites canary values at every invocation of `memcpy()`. The final execution measured the speed of *bzip2* after it had been statically rewritten and modified to include a configuration of DCR that rewrites canary values at every invocation of `fwrite()`.

Figure 11.5 shows the results of these four experiments. The overhead of DCR rerandomizing canary values at every `memcpy()` is 16.08%, with 13.33% of that overhead due to the static rewriter itself. By comparison, the overhead of DCR rerandomizing canary values at every `fwrite()` is negligible relative to the overhead of the static rewriter itself. Although there are more updates per randomization in the latter case, there are significantly fewer total randomizations performed when compared to the number of randomizations performed in the former. This accounts for the differences in overhead in the two scenarios. This is a concrete case where the choice of how often to rerandomize affects performance. Again, the plan is to study the tradeoffs of variable attack window sizes in the general case in future work.

11.4.3 DCR and BROP

ROP is a new and dangerous form of software attack. In ROP attacks, the attacker finds a list of "gadgets" that he/she can chain together to perform a nefarious task. The attacker places the chain of gadgets on the stack and then transfers program control to that chain. A ROP attack is particularly dangerous because it defeats the protection afforded by NX. A complete description of ROP is beyond the scope of this paper. See Shacham for a complete description of ROP [193].

To launch a ROP attack, the attacker must be able to find the addresses of gadgets in the target program. If the attacker cannot examine the object code of a process offline (if, for instance, the target is a server running on a remote host), then the attacker's task of locating gadgets becomes very difficult. The BROP attack, presented in 2014 by Bittau et al., is designed to find and exploit gadgets in just this scenario [42]. Their attack carefully probes a remote target using its oracles until it is able to find a set of gadgets that it can use to download a copy of the program's instructions. The attack proceeds by, first, finding a gadget that will invoke the `write()` system call, the so-called *write gadget*. The write gadget is the attack's oracle. Then, it finds gadgets that will control the parameters to that function. Finally, it invokes the write gadget repeatedly (queries the oracle) to download the entire contents of the program. After that, the attacker can analyze the program's instructions offline, find the locations of other gadgets and inflict further damage using a traditional ROP attack.

The BROP attack relies on several very important assumptions. The authors rely on the fact that the remote target software automatically restarts after a crash and does not re-randomize its address space between executions. This assumption holds for servers that spawn separate child processes to handle clients (the accept-fork paradigm for handling multiple clients simultaneously). In such a framework, the address space of the processes that handle client requests are not re-randomized, even if ASLR is enabled. In other words, the information gained with every attack probe is valid on subsequent attack attempts – the attack window never closes.

To demonstrate that these assumptions are not unreasonable, consider that *nginx* meets these criteria. *nginx* is a web server that, as of July 2017, powers more than 29% of the top 1 million busiest Internet sites [158].

To summarize, BROP attacks are widely applicable and defeat NX and ASLR. However, their deployment still requires a way of controlling the target. This constitutes what can be considered a separate attack with its own oracles, queries and attack window. This complete the preparatory attack, Bittau et al.'s BROP attack uses the traditional stack overflow. The function vulnerable to stack overflow is the oracle and rewriting the stack using that vulnerability are queries.

Program	GCC and SSP	GCC and DCR
nginx	V	P
ali	V	P
nslr	V	P

V: vulnerable, P: protected

Table 11.1: Results of BROP attacks against software protected with traditional canaries (SSP) and with Dynamic Canary Randomization (DCR)

Recall, however, that stack canaries explicitly detect this type of attack. Therefore, the value of the stack canary must be leaked for BROP attacks to succeed. The authors use byte-for-byte stack reading to leak the canary values from their remote target. First introduced by Zabrocki [248], byte-for-byte stack reading is an iterative process: it overwrites the stack byte-by-byte and observes program behavior. If the program crashes when byte b is written to the stack at position a, then something about the stack is wrong. However, if the program does not crash, it is reasonable to assume that the target program's stack does hold b at a. On the next query, the attack will maintain the previous byte in it's position and overwrite stack position $a+1$ until it determines the value b'. Because the values of the canary are static during the course of execution (the attack window), the attacker is eventually able to learn the canary values and defeat their protections.

The developers of the BROP attack call this generalized stack reading (GSR). It is fundamental to the attack but easy to prevent with DCR. Reproducing the author's results was the first step in demonstrating the effectiveness of DCR at thwarting BROP attacks. The authors developed an automated tool, braille, to perform BROP attacks. They used braille to attack *nginx*, *mysql* and *ali*.[2] Besides successful replication of their attacks on *nginx* and *ali* using braille[3] this experiment includes an additional data point: a program with a seeded buffer overflow vulnerability, *nslr*, compiled with GCC and SSP and attacked with braille. Their automated attack succeeded against *nslr*.

To test the effectiveness of the defense, each of these vulnerable binary programs was protected by DCR. Canary randomization was set to occur at every `fork()`. After applying DCR, the attack window was closed at every `fork()` system call and attacks using braille were unsuccessful in every case. See Table 11.1.

11.5 Future Work

In the future, it will be important to optimize the runtime efficiency of DCR and to study the tradeoffs between variable attack windows and runtime performance. Because DCR relies on a static rewriter for

[2] *ali* is a custom server written by an independent researcher at the author's university. In an attempt to simulate attacking a completely "closed" server over the network, Bittau et al. were not given offline access to the program source code *or* binary.

[3] Bittau et al. did not provide enough information in their paper to recreate their tests on *mysql*.

implementation, further study is warranted regarding performance improvements for the static rewriter itself in addition to the DCR technique *per se*.

There are also additional features that can be added to DCR. For instance, a useful feature would be support for the DCR user who wants to randomize canary values after every n instructions or every s seconds.

It is also important to understand the other classes of server software that could benefit from the security offered by DCR. DCR is applicable to multi-threaded programs but servers are also implemented in a single threaded, single process program that coordinates handling multiple tasks simultaneously through a workqueue. Servers like `ntpd` and `olsrd` use this technique and are widely deployed.

11.6 Conclusion

Stack canaries are a well-known and effective technique for detecting and defeating stack overflow attacks. However, they are not perfect. DCR is a technique for rerandomizing stack canaries that is applicable to binary programs (i.e., it does not require access to the program's source code). The results described in this chapter show that DCR operates with minimal overhead and gives its user the flexibility to specify the conditions under which to rerandomize the canary. In comparison to existing canary rerandomizers, the results for DCR show that it improves on the state of the art. DCR improves software security by demonstrating its ability to prevent real-world attacks on well-known software (e.g., *nginx*) "protected" by traditional stack canaries. Finally, DCR is another demonstration of the ability of the static binary rewriter architecture and design described in this dissertation to transform SOUP to improve its security and reliability.

Chapter 12

Control-Flow Integrity

12.1 Introduction

Generic security protections are a class of security tool that, although not 100% effective in every situation, greatly improve the baseline security posture of a majority of software. Because these protections are generally applicable, they can be deployed proactively to protect against the flaws inevitably latent in all software.

Control Flow Integrity (CFI) is one such generic security protection. CFI is a "security policy that dictates that software execution must follow a path of a [CFG] determined ahead of time" [22]. In other words, the CFI policy restricts runtime transfers of program control to locations determined offline, statically. These valid targets of runtime control flow can be determined at one or more of the following levels: "source code, binary analysis or execution profiling" [22].

This chapter describes the design and implementation of a User-specified transformation that applies CFI to SOUP. The transformation, named SelectiveCFI, is implemented using Zipr and has two parts.

Analysis In this phase, SelectiveCFI determines the valid runtime control flow targets by analyzing SOUP binaries. Analysis at any other level would either require a program's source code or a test suite with 100% code coverage, either of which would make the solution inapplicable to the conditions described in Part I. The analysis will generate a set P of pairs of instructions that perform indirect program control transfer and the set of that instruction's valid targets during *nominal* execution of the original program. By implication, indirect transfers that happen at runtime that do not match a pair in P mean the program/library has been hijacked.

Instrumentation In this phase, the input program/library is modified to add runtime enforcement of the SelectiveCFI policy. The statically rewritten binary will contain *nonces* that *decorate* each of the

targets of the indirect control flow transfer instructions in P at their actualized address in the statically rewritten program/library. Further, the transformation will rewrite each program control instruction in P to check that the target is properly decorated with a nonce before performing the control transfer. If the target is valid (i.e., decorated with a nonce), execution continues. If not, execution terminates according to a user-defined shutdown procedure.

12.2 Design

12.2.1 Analysis

The Analysis phase processes the program being protected to determine the set of valid targets for indirect program control transfers. The output of the analysis, P, a set of pairs of sources of indirect program control transfer and the set of their valid targets, is the input to the subsequent, instrumentation phase.

$$P = [[a, a_t = [a_{t_1}, a_{t_2}, a_{t_3}, \ldots]], [b, b_t = [b_{t_1}, b_{t_2}, b_{t_3}, \ldots]], \ldots] \tag{12.1}$$

Pairs in P indicate that for the source of an indirect program control transfer a, the valid targets for that indirect program control transfer are a_{t_1}, a_{t_2} and a_{t_3}. In other words, if the original program/library indirectly transfers program control from a to a target t such that $t \notin a_t$, its control flow has been hijacked.

For the purposes of understanding the *completeness* of the SelectiveCFI analysis, let there be a set $P_{\mathcal{O}}$. $P_{\mathcal{O}}$ has the same structure as P. Further, let there be an oracle function, $\mathcal{O}$ that, given an indirect program control flow instruction a, returns the actual, valid targets that the program may take at runtime.

$$P_{\mathcal{O}} = [[a, A_t = \mathcal{O}(a)], [b, B_t = \mathcal{O}(b)], \ldots] \tag{12.2}$$

The set P is built using data flow and control flow analysis of the disassembled code of the vulnerable program.

Data Flow Analysis

Data flow analysis is used to determine a_t for a in $[a, a_t]$ in P using algorithms like liveness analysis, constant propagation and reaching definitions. In SelectiveCFI, data flow analysis is done on the static single assignment (SSA) form of the program's instructions and is commonly used to determine the range of potential values of targets of indirect `call` instructions.

High Level Source Code	Compiler-generated Object Code

`void caller(int a_or_b) {`
` void (*f)(void) = a;`
 	
` if (a_or_b == 1)`
`  f = a;`
` else if (a_or_b == 2)`
`  f = b;`
 	
` f();`
`}`
	`40054c: sub   $0x8,%rsp`
	`400550: cmp   $0x2,%edi`
	`400553: mov   $0x400539,%edx`
	`400558: mov   $0x400526,%eax`
	`40055d: cmove %rdx,%rax`
	`400561: callq *%rax`
	`400563: add   $0x8,%rsp`
	`400567: retq`

Table 12.1: Invoking a function conditionally and the idiomatic object code emitted by the compiler.

Compilers often generate indirect `call` instructions to perform virtual dispatch in object-oriented languages that support such semantics or to invoke dynamically loaded library functions.

Compilers also, often, generate indirect `call` instructions in scenarios where the HLL programmer is using a function pointer as a callback. Listing 12.1 shows an example of a utility function (`caller`) that conditionally chooses a function to be called based on an input parameter. In this example, the HLL programmer is using `caller` to invoke either function `a` or `b` depending on whether the programmer passes 1 or 2 as the value of `a_or_b`.

As mentioned previously, data flow analysis in SelectiveCFI is performed on the SSA representation of the program's machine code. For the example in Listing 12.1, data flow analysis works this way: The analysis itself is triggered when the analyzer sees the indirect `call` at `0x400561`. The analyzer deciphers the `call` and records that it is using register `rax` to hold the address of the function to be invoked. From there, the analyzer works backward to determine whether the potential values for that register at runtime can be determined statically. In this case, they can. The analyzer determines that the `cmove`, `mov` and `mov` instructions at addresses `0x4055d`, `0x400558` and `0x400553`, respectively, affect the value of register `rax` at runtime. The `cmove` at `0x40055d` is represented as a PHI node in the SSA representation of the function and the `movs` are DEFs. Based on this analysis, SelectiveCFI can reason that there are only two possible values of `rax` at runtime: `0x400539` and `0x400526`. Therefore, $[a = 0x400563, a_t = [0x400539, 0x400526]]$ becomes an entry in P.

Control Flow Analysis

Control flow analysis is used to determine a_t for a in $[a, a_t]$ by reasoning about a program's control flow operations near a. Control flow analysis may be seen as a special case of data flow analysis – it relies more on an analysis of the program's control flow operations near a to determine a_t than data flow analysis which is based more on liveness, reaching and using analysis. Control flow analysis is triggered in several cases.

High Level Source Code	Compiler-generated Object Code
int main() {	4005b8: 29 05 40 00 00 00 00 00
int a = 5, b;	4005c0: f4 04 40 00 00 00 00 00
switch (a) {	4005c8: fd 04 40 00 00 00 00 00
case 1:	4005d0: 06 05 40 00 00 00 00 00
b = 1;	4005d8: 0f 05 40 00 00 00 00 00
break;	4005e0: 18 05 40 00 00 00 00 00
case 2:	4005e8: 21 05 40 00 00 00 00 00
b = 2;	
break;	4004da: movl $0x5,-0x4(%rbp)
case 3:	4004e1: cmpl $0x6,-0x4(%rbp)
b = 3;	4004e5: ja 400529
break;	4004e7: mov -0x4(%rbp),%eax
case 4:	4004ea: mov 0x4005b8(,%rax,8),%rax
b = 4;	4004f2: jmpq *%rax
break;	4004f4: movl $0x1,-0x8(%rbp)
case 5:	4004fb: jmp 400529
b = 5;	4004fd: movl $0x2,-0x8(%rbp)
break;	400504: jmp 400529
case 6:	400506: movl $0x3,-0x8(%rbp)
b = 6;	40050d: jmp 400529
break;	40050f: movl $0x4,-0x8(%rbp)
}	400516: jmp 400529
return b;	400518: movl $0x5,-0x8(%rbp)
}	40051f: jmp 400529
	400521: movl $0x6,-0x8(%rbp)
	400528: nop
	400529: mov -0x8(%rbp),%eax
	40052c: pop %rbp
	40052d: retq

Table 12.2: A switch statement implemented in C and the idiomatic object code emitted by the compiler.

It is especially useful for determining the pair $[j, j_t]$ for indirect `jmp` instructions. These instructions commonly appear in compiler-generated code that implements an HLL's `switch` statements. See Listing 12.2. Control flow analysis detects these idiomatic sections of code and computes the possible values for the register that hold the target of the indirect `jmp` instruction. In the example, the analysis attempts to reason about the potential values of `rax` at `0x4004f2`. The analysis inspects the instruction at `0x4004ea` and determines that the potential targets of the `jmp` instruction are loaded from memory beginning at `0x4005b8`. Further, the analysis uses the `cmp` instruction at `0x4004e1` to recognize that there are only 6 potential targets for the `jmp`. Using the instruction at `0x4044ee` to reason about how far apart each address is in memory, the analysis recognizes that the target addresses are located in memory between `0x4005b8` and `0x4005e8`. The code in Listing 12.2 is just one example of the type of idiomatic code that the analysis is able to recognize and dissect.

Analysis Completeness

In both of the examples above, at the conclusion of analysis, the SelectiveCFI is able to determine the *complete* set of targets for the indirect control instructions. In a complete analysis of an indirect control flow operation a, the set a_t contains a superset of all the possible runtime targets for that operation. In other words, a_t may contain targets that a cannot actually reach but there is no target that a can reach at runtime that is not in a_t. With respect to the formal definitions above, a complete analysis of the indirect program control instruction a means that $a_t \subseteq A_t$ where $A_t = \mathcal{O}(a)$.

On the other hand, there are other indirect program control flow operations that cannot be analyzed completely because of shortcomings in the data flow and control flow analysis. For example, analysis of a return indirect control flow operation from function f is incomplete when control flow analysis cannot determine precisely the addresses of calls to f. Or, analysis is incomplete when data flow analysis cannot precisely determine the value of a register or memory location that contains a pointer to a function invoked using a call indirect control flow operation. When incomplete, the analysis of the indirect program control flow instruction at a determines that the set a_t is missing at least one possible runtime target. In other words, $a_t \nsubseteq \mathcal{O}(a)$. Because a program protected by SelectiveCFI enforces $t \in a_t$ for each target $t \in \mathcal{O}(a)$ at each indirect program control transfer a at runtime, incomplete analysis can cause broken functionality.

Therefore, those sets a_t that are incomplete must be augmented with additional targets to preserve program functionality. The targets that augment incomplete sets of targets are known as *hell nodes* [46, 225].

For a program where $\exists [a, a_t] \in P \mid a_t \nsubseteq \mathcal{O}(a)$ there are three sets of hell nodes: the jump hell node (H_{jump}), the return hell node (H_{ret}) and the call hell node (H_{call}). The jump hell node contains the address of every jump target in the program; the return hell node contains the address of every instruction after a `call`; the call hell node contains the address of each function in the program.

Depending on the type of indirect program control flow instruction that cannot be analyzed completely, the set of targets is augmented with the addresses from the appropriate hell node. For instance, if the indirect program control flow instruction at a that cannot be completely analyzed is a `ret`, then a_t is expanded such that $a_t = a_t \cup H_{ret}$.

Hell nodes are built from the same underlying analysis as the rest of SelectiveCFI and require an analysis of the program's CFG to build. All targets in the hell nodes, no matter the type of the hell node, are IBTs. A *provenance* describes the source of the IBT and each of the IBTs discovered by the analysis of the program's CFG is assigned one or more provenances. The provenance(s) of the IBT(s) is/are used to classify and sort them into one or more of the three hell nodes.

Data Section The provenance of an IBT is labeled as Data Section if the IBT was discovered from the Data Section of the input binary.

Read Only Data Section The provenance of an IBT is labeled as Read Only Data Section if the IBT was discovered from the Read Only Data Section of the input binary. IBTs of this provenance are mostly used in jump tables.

Text Section The provenance of an IBT is labeled as Text Section if the IBT was discovered from the Text Section of the input binary.

Dynamic Symbol The provenance of an IBT is labeled as Dynamic Symbols if the IBT was discovered among the entries in the dynamic symbols table of the input binary.

Return Point The provenance of an IBT is labeled as Return Point if it is the fallthrough instruction of a `call`.

Init Array The provenance of an IBT is labeled as Init Array if it comes from the array of functions called when the program is initialized [161].

Fini Array The provenance of an IBT is labeled as Fini Array if it comes from the array of functions called when the program is terminated [161].

Unreachable The first instruction in basic blocks marked as Unreachable by CFG analysis is marked as an IBT with an Unreachable provenance. Although control flow analysis labels these instructions as unreachable, they are conservatively treated as IBTs. As testing proves the reliability of reachability analysis, IBTs of this provenance may no longer be required.

User-specified The SelectiveCFI user may manually specify IBTs and these IBTs are labeled User-specified. An end user may specify IBTs for debugging purposes or to correct for a flaw in the control flow analysis.

Unknown An IBT whose provenance cannot be determined is labeled as Unknown.

Again, because the use of hell nodes means that $A_t \subseteq a_t$ where $A_t = \mathcal{O}(a)$, a program secured with SelectiveCFI program will not produce any false positive security alerts. However, any target $t \in a_t$ and $\notin A_t$ represents a way to bypass the security of SelectiveCFI. Continuing to improve the performance of the Analysis so that a_t eventually equals A_t is the subject of future work.

Category	Data Section	Read Only Data Section	Text Section	Dynamic Symbols	GOT	Addressed	Return Points	Init Array	Fini Array	Unreachable	User-specified	Unknown
Return							✗			✗	✗	✗
Jump	✗	✗	✗	✗	✗	✗				✗	✗	✗
Call	✗	✗	✗	✗		✗		✗	✗	✗	✗	✗

Table 12.3: Sources of Hell Nodes.

Optimizations

Enforcement is costly for performance with additional overhead to check the safety of every indirect program control transfer.See Section 12.4. The most effective way to decrease the overhead of SelectiveCFI is to not enforce CFI in the first place. Maintaining the security properties gained through the enforcement of control flow integrity without having to check the targets at runtime requires effective static analysis of the program.

Safe Indirect Jumps There are only a few cases where compilers commonly emit indirect jumps – one of the type of instructions that a CFI implementation is designed to protect.

The most common use by compilers of indirect jump instructions are to compile `switch` statements. As part of the code emitted by a compiler for a `switch` statement, the compiler emits code that checks the bounds of the value in the register used for indirect program control transfer and/or emits code that selects potential values for the register used for indirect program control transfer from read-only memory.

As more and more developers implement programs in object-oriented programming languages, indirect jumps are becoming more and more common. Research as early as 1994 indicated that applications compiled from C++ contain 23x more indirect jumps than applications compiled from C [47]. In the SPEC benchmark suite, the programs written in the C programming language average approximately 0.08 indirect jumps per kilobyte of object code whereas the programs written in the C++ programming language average approximately 0.22 indirect jumps per kilobyte of object code.

A vast majority of the Challenge Binaries for the CGC are written in the C programming language. The ones that are written in C++ do not use any virtual functions. For the entire corpus of the CGC programs, there are only 750 switch statements that compile into 218 indirect jumps. Presumably the difference is the result of the compiler using constant propagation to compile `switch` statements into a series of conditionals or direct jumps. As a result, for the Challenge Binaries in CGC, the rate of indirect jumps per 1K of object code is vanishingly small.

Because of the way that indirect jumps are typically used, enforcement of the safety of the targets of these instructions can be omitted without significantly decreasing the protection offered by SelectiveCFI for programs written in non-object-oriented programming languages.

Safe Functions A safe function's return indirect control flow operation does not need to be enforced. A *safe* function is one that, under all circumstances at runtime, only ever returns to its caller. The safety of a function is transitive. A function f cannot be safe unless all the functions that it calls are also safe. Static analysis determines whether a function is safe based on several heuristic criteria.

Static analysis begins by assuming that a function is safe and checks a series of conditions that, if true, indicate a function is unsafe.

Invalid SSA or RTL Generation In pathological cases, the overall analysis fails because it could not generate a register transfer language (RTL) or an SSA IR of the function. Checking for the following conditions presumes the valid generation of an SSA and RTL representation of the function. When this is not the case, the analysis cannot perform any assessment of the function's safety and marks the function unsafe.

Unresolved Indirect Jump An unresolved indirect jump is a `jmp` through a register or memory location whose value cannot be determined statically. The target of an unresolved indirect jump may be anywhere in the program. For the purpose of analyzing a function's safety, the target of the indirect jump may very well be in the current function. Because of this possibility, the CFG constructed for this function upon which the analysis is based cannot be trusted. Therefore, the analysis marks the function as unsafe.

Unresolved Indirect Calls An unresolved indirect call is a `call` through a register or memory location whose value cannot be determined statically. The presence of such a `call` instruction in a function means that, because it is impossible to guarantee that the callee is safe, the analysis has to assume that this function is also unsafe.

Unsafe Write to the Local Stack Frame An unsafe write to the function's stack frame can potentially overwrite the return address. Because the return to the calling function is often accomplished by reference to that address as it is stored in the program's stack, operations that affect the values in the function's stack frame must be analyzed closely. Memory operations that address the stack frame using an offset from the stack pointer that can be computed offline makes it possible for the analysis to definitively determine whether the operation modifies the function's return address. The offset can

be computed offline if it is based on a constant value or, if the offset is computed with respect to a register value or a memory value, the range of values for that register or memory location can be computed statically. It is possible to compute ranges of values for the offset statically using an SSA or constant-propagation analysis. On the other hand, if the offset for the stack memory operation uses a register or memory location whose range of values cannot be determined statically, the analysis cannot guarantee that operation will never overwrite the function's return address. In the latter case, the function is marked unsafe.

Push of the Stack Pointer to the Stack When a function (outside of its epilogue) pushes the stack pointer address to the stack, its value can be used by any callee function to, potentially, arbitrarily overwrite parts of the program's stack. Arbitrary writes to the stack could overwrite the stored return address. If that return address is overwritten, the function is marked unsafe because the analysis cannot guarantee it will return only to its caller in all circumstances.

Unsafe Use of a Copy of Stack Pointer When an instruction makes a copy of the stack pointer (outside the prologue or epilogue) into a general purpose register, the analysis must determine whether that general purpose register is used elsewhere as part of an unsafe memory operation. An unsafe memory operation is one that uses a loop variant or a value passed by the function caller to calculate the memory operation's destination address. When the stack pointer is used in this way, the analysis marks the function as unsafe.

Unsafe Access of/Write to the Return Address An access of or write to the function's return address is a special case of an unsafe write to the stack and also results in the analysis marking the function as unsafe.

Multiple Entry Points The existence of entry points in a function beyond its prologue invalidates certain assumptions made in the analysis. Therefore, in the presence of multiple entry points, the analysis cannot guarantee the correctness of its assessments about the safety of the function and simply marks the function as unsafe.

Unanalyzed Stack Pointer One of the assumptions of the proper analysis of the function's safety is that it is possible to precisely model the value of the stack pointer throughout the function's execution. In particular, it is assumed that, at every instruction, the analysis knows the value of the content of the stack pointer register relative to its value when the function was entered.

The presence of one of the above conditions is the primary way a function is marked unsafe. There are, however, situations where an otherwise safe function may become unsafe because it invokes a child function with certain characteristics. In particular, a function is unsafe as a callee when

It is Itself Unsafe *or*

It Writes To the Stack Outside Its Own Frame A function that performs unsafe writes outside (above) its own stack frame could potentially affect the return address of one or more of the invoking functions. Therefore, the presence of this function in the call chain means that any function that invokes this function (directly or indirectly) becomes unsafe.

To completely analyze the safety of a function, two factors are considered. First, whether the function itself is unsafe. Second, if the function invokes any other functions, whether any of those functions are unsafe or are unsafe as callees. If either of these two factors are true, then the function is unsafe.

The security architect can afford to not instrument enforcement of the `ret` indirect control flow operation from a safe function and maintain CFI. Results show that not instrumenting enforcement of these operations improves the performance of SelectiveCFI significantly. See Section 12.4.

There are two options for handling the return indirect control transfer instruction from a safe function: retain the instruction itself or exchange that instruction with a version of the safety instrumentation that omits the check. Although both choices achieve the same benefits, there are performance implications. Section 12.2.2 describes the instrumentation options and 12.4 evaluates the performance of the two choices.

12.2.2 Instrumentation

After the Analysis phase is complete, the Instrumentation phase knows P, the set of pairs that represent at least all valid indirect control flow transfer sources and destinations in the native program. The goal of the Instrumentation phase is to create a statically rewritten program that constrains the runtime indirect control flow to the pairs in P. In other words, in the statically rewritten program/library, if there is source of indirect program control transfer a that transfers control to an address outside a_t, the program must halt. This is the essence of CFI enforcement.

The remainder of this section describes the implementation of the Instrumentation phase for the x86 platform. Although the details are specific to the platform, the overall concepts are applicable to other platforms.

The first step of instrumentation is to decorate each target with a nonce. The nonce is a value that will be used for comparison to determine if the target of an indirect program control transfer operation is valid.

```
4000f2: lea   0x400126,%r9
4000fa: cmp   $0x2,%r10
4000fe: jne   400108
400100: lea   0x40012c,%r9
400108: callq *%r9
40010b: retq

40010c: lea   0x40012c,%r9
400114: cmp   $0x3,%r10
400118: jne   400122
40011a: lea   0x400132,%r9
400122: callq *%r9
400125: retq

400126: mov   $0x1,%ebx
40012b: retq

40012c: mov   $0x2,%ebx
400131: retq

400132: mov   $0x3,%ebx
400137: retq
```

Table 12.4: An example of two indirect program control flow operations whose targets are not identical but share some targets in common.

The nonce may be placed anywhere that is a fixed distance from the target as long as that distance is known *a priori*. In practice, however, there are good choices and bad choices for where to place the nonce relative to the target. In the implementation described herein, the nonce is placed immediately before the target.

It seems possible to assign a single nonce to each target and decorate the targets accordingly. If that were the case, the position of the decoration relative to the target would depend solely on $|P|$. In other words, the nonce would have to be just big enough to uniquely identify set a_t for every indirect program control transfer operation a in P.

However, there is the possibility that more than one source can validly transfer program control to the same target. Consider two pairs $A = [a, a_t = [a_{t_1}, a_{t_2}, a_{t_3}, ...]]$ and $B = [b, b_t = [b_{t_1}, b_{t_2}, b_{t_3}, ...]]$. It is possible that $a_t \cap b_t \neq \emptyset$. In other words, the program control transfers at a and b might both validly transfer control to the same target. Decorating $[a_t \cup b_t]$ with the same nonce would allow, incorrectly, all transfers from b to any valid target of a, and vice versa.

Such a scenario is exemplified in Listing 12.4. The instructions between addresses `0x4000f2` and `0x40010b` comprise a function `caller1`; the instructions between addresses `0x40010c` and `400125` comprise a function `caller2`. The functions that start at `0x400126`, `0x40012c` and `0x400132` are called `a`, `b` and, `c`, respectively. Depending on the contents of register `r10`, `caller1` can call either `a` or `b`; depending on the contents of register `r10, caller2` can call either `b` or `c`. Therefore, $0x400108_t = [0x400126, 0x40012c]$ and

$0x400122_t = [0x40012c, 0x400132]$ and $0x400108_t \cap 0x400122_t \neq \emptyset$. To maintain the integrity of the control flow of the program, the `call` at `0x400108` should not be able to invoke `c` and the `call` at `0x400122` should not be able to invoke `a`.

In order to accommodate this case, instructions that are the valid target of more than one transfer must be decorated with multiple nonces. Naïvely, it would seem, then, that enforcement would require that instrumentation sequentially check multiple nonces to validate each program control transfer.

An alternate approach is to place the nonce for each program control source at the same fixed offset relative to all of its targets. The combination of the source, targets and offset is known as the *color*. The set of colors is

$$C = [[a, a_t, a_o], [b, b_t, b_o]] \tag{12.3}$$

where

$$\forall [a, a_t, a_o], [b, b_t, b_o] \in C, a_t \cap b_t \neq \emptyset \Rightarrow a_o \neq b_o \tag{12.4}$$

This approach makes the lookup of the nonce value relative to the target instruction deterministic at each control flow transfer source at the expense of potentially wasted space. Consider two target sets a_t and b_t where $[x_1, x_2] = a_t \cap b_t$. x_1 is a very common target – many different program control flow instructions besides a and b target that address. x_2 however, is only targeted by a and b. Because of the popularity of x_1, the first offset that is available is 15 bytes before the target and so the nonce must also be placed 15 bytes before x_2. There is no wasted space at x_1 because the preceding 15 bytes are filled with other nonces. However, there are 15 wasted bytes at x_2 because there are no other nonces that decorate the target.

As is obvious from the discussion of colors, using colors is not strictly necessary. Colors add granularity to the protection offered by the implementation of CFI at the expense of potentially wasted space and additional nonces. If the security architect is willing to sacrifice some security for additional performance, he or she may choose to accept that every indirect control flow transfer can safely target any of the statically identified IBTs rather than just a specific subset. In the experimentation below, the coarse version of CFI is referred to as Basic SelectiveCFI and the granular version of CFI that uses colors is referred to as SelectiveCFI with Coloring.

No matter whether the user prefers the coarse or the granular approach to CFI, the second step of instrumentation is to add the enforcement mechanism at the source of each indirect program control transfer. On the x86 platform there are three different instructions that effect indirect program control transfer: `jmp`, `call` and `ret`. Instrumenting these three different sources of indirect program control transfer to enforce CFI requires similar, but not identical, techniques.

jmp	ret
push target	
	pop r11 cmp (r11), NONCE jne SLOWPATH jmp *r11

Table 12.5: Enforcement instrumentation for CFI.

Overall, enforcement has a *fast path* and a *slow path*. On the fast path, the target of the program control transfer is checked for validity by comparing the nonce at the target (adjusted for the offset) with the expected value. If the values match, then program control transfer is valid and the enforcement mechanism allows program execution to proceed. The enforcement mechanism for `jmp`s and `ret`s share as much code as possible to simplify implementation. In both cases, the value compared with the nonce is in register `r11`. Register `r11` was chosen because the x86 ABI specifies that its value may be destroyed at calls [189]. Although an indirect program control transfer via `jmp` is not the same as a `call`, the semantics are similar and it is assumed that `r11` is not preserved at those transfers. Experiments and experience show that, however, this is not the case (see 12.4.2). The mechanism for loading the target address into `r11` differs depending on whether the transfer is done by `jmp` or `ret`. At a `ret`, the target address is on the top of the stack. Therefore, loading the target address can be accomplished with a `pop` into `r11`. Driven by the desire to share as much code as possible between the enforcement mechanism of `jmp`s and `ret`s, at a `jmp` the target address is pushed on the top of the stack and then the exact mechanism employed for return enforcement can be reused. A pseudo implementation is shown in Table 12.5.

As described in Section 12.2.1, for a function statically deemed safe, there is no need to instrument for safety the return indirect control flow operation. One way to do this is to simply retain the return indirect control flow operation in the statically rewritten program/library. This technique will be referred to as the *basic* method of instrumenting for safety the return indirect control flow operations from functions statically deemed safe. The other option is to replace the return indirect control flow operation with part of the instrumentation shown in the right column of Table 12.5. In the case of a return indirect control flow operation from a safe function, there is no reason to perform the `cmp` and the `jne`. This technique will be referred to as the *alternate* method for instrumenting for safety the return indirect control flow operation. See Table 12.6 for the two methods for safety a return indirect control flow operation. Although there are more instructions executed in this alternate method, the basic method introduces a mismatch with the hardware's return address predictor which has performance implications. See Section 12.2.2 for a detailed description of hardware return address predictors and Sections 12.4.2 and 12.4.3 for an analysis of the performance

Basic	Alternate
`ret`	`pop r11` `jmp *r11`

Table 12.6: Methods for instrumenting for safety the return indirection program control operation from a function statically deemed safe.

implications of this choice.

If the nonce value does not match the expected value, the transfer is not necessarily invalid. There may not be enough space preceding every indirect control flow transfer target to place the nonce values. This occurs when two pinned addresses are nearby and there is not enough space to place all the nonces. In other cases, the target of the indirect program control may be at the very first memory location of the program in memory. To determine whether the transfer is actually invalid when the fast path fails to validate a target, enforcement takes the slow path.

There is good reason to call this the slow path. On the slow path, the source address of the program control transfer and its target are compared iteratively against a list of valid transfers. Experimental evidence shows that slow path code is infrequently necessary. See 12.4.

Consider the example in Figure 12.1 that shows SelectiveCFI instrumentation of an indirect `jmp`. In this example, the original program contains a `switch` statement with three cases that has been compiled into a jump table in the original program. Corresponding to the three cases are three IBTs. As a result, the Analysis phase determines that there are three places that need to be decorated with nonces to indicate that they are valid indirect control flow targets. See the Original version of the program in Figure 12.1. In this program, $|P| = 1$, there are no overlapping targets, and only one color. As a result, this example omits those details. There is space enough before a_{t_1} and a_{t_2} to accommodate the nonces. There is not enough space before a_{t_3}, however. At a in the original program is the source of the indirect program control transfer. The version of the `jmp` instruction instrumented for enforcement is rewritten at a' in the rewritten program. In the case where program control would be transferred to a_{t_1} or a_{t_2}, the enforcement mechanism would proceed along the fast path. However, in the case where the program control would be transferred to a_{t_3}, the enforcement would proceed along the slow path. The slow path code is not shown in this example.

Consider the example in Figure 12.2 that shows SelectiveCFI instrumentation of a `ret`. In this example, the original program calls `func_1` as the second operation in its `start` function. SelectiveCFI instrumentation of the `ret` in `func_1` ensures that the function resumed after completion of `func_1` is the function that called `func_1` in the first place. In other words, if `func_1` were a vulnerable function and an attacker replaced the function's return address with the address of a ROP gadget, for instance, the protected program would terminate rather than allow the hijacked program to continue.

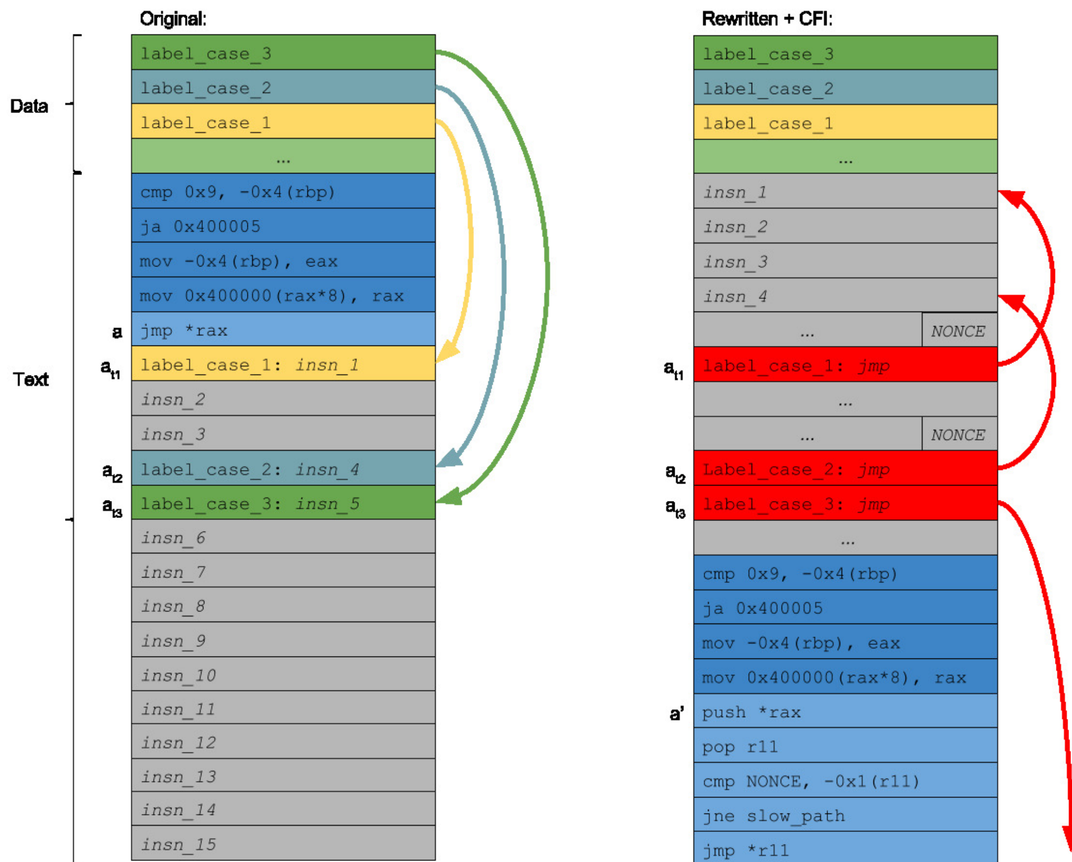

Figure 12.1: The SelectiveCFI instrumentation of a jump.

The instruction that calls `func_1` is rewritten at a' where the `call` instruction is rewritten as a `push`/`jump` (See 4.2.2). The instruction after the call is pinned. Because the target function, `func_1` in this case, uses an indirect program control transfer instruction to return control to this instruction after its completion, the instruction at b is necessarily an IBT and is, therefore, pinned. (See 4.4.1 for more information on this.)

In this program, $|P| = 1$, there are no overlapping targets, and only one color. As a result, this example omits those details. There is space enough before d and b to accommodate the nonces so there is no need to include slow path code in the rewritten program.

The enforcement code is placed at c' in the rewritten program where the `ret` would have been placed. The target of the program control instruction, `func_1`'s return address, is on the top of the stack at the time the enforcement code is executed. In order to do a comparison with that value, the first step of the enforcement is to load that value from the stack into register `r11`. Again, the ABI does not mandate that `r11` be preserved across function calls so it is safe to use that register as a temporary value during enforcement. The second step of enforcement is to load the nonce value from the potential target. The `cmp` instruction is built the way it is in the example because $|P| = 1$, there is a single color and the nonce value is one byte

Figure 12.2: The SelectiveCFI instrumentation of a function call.

wide and placed exactly 1 byte before the potential target address.

As mentioned earlier, because all the nonce values in this example fit, there is no need for the rewritten program to include slow path enforcement code. Therefore, the only code at the `slow_path` target is the code required to terminate the program. Because there are no false positives to disambiguate along the slow path, the only reason that program control would be transferred to the slow path is if the enforcement mechanism detects a hijacking attempt.

Executable Nonces

There is a significant amount of literature on the performance benefits of the CPU's call/return branch predictor (e.g., [201]). The predictor uses a stack data structure to match calls with returns to predict the

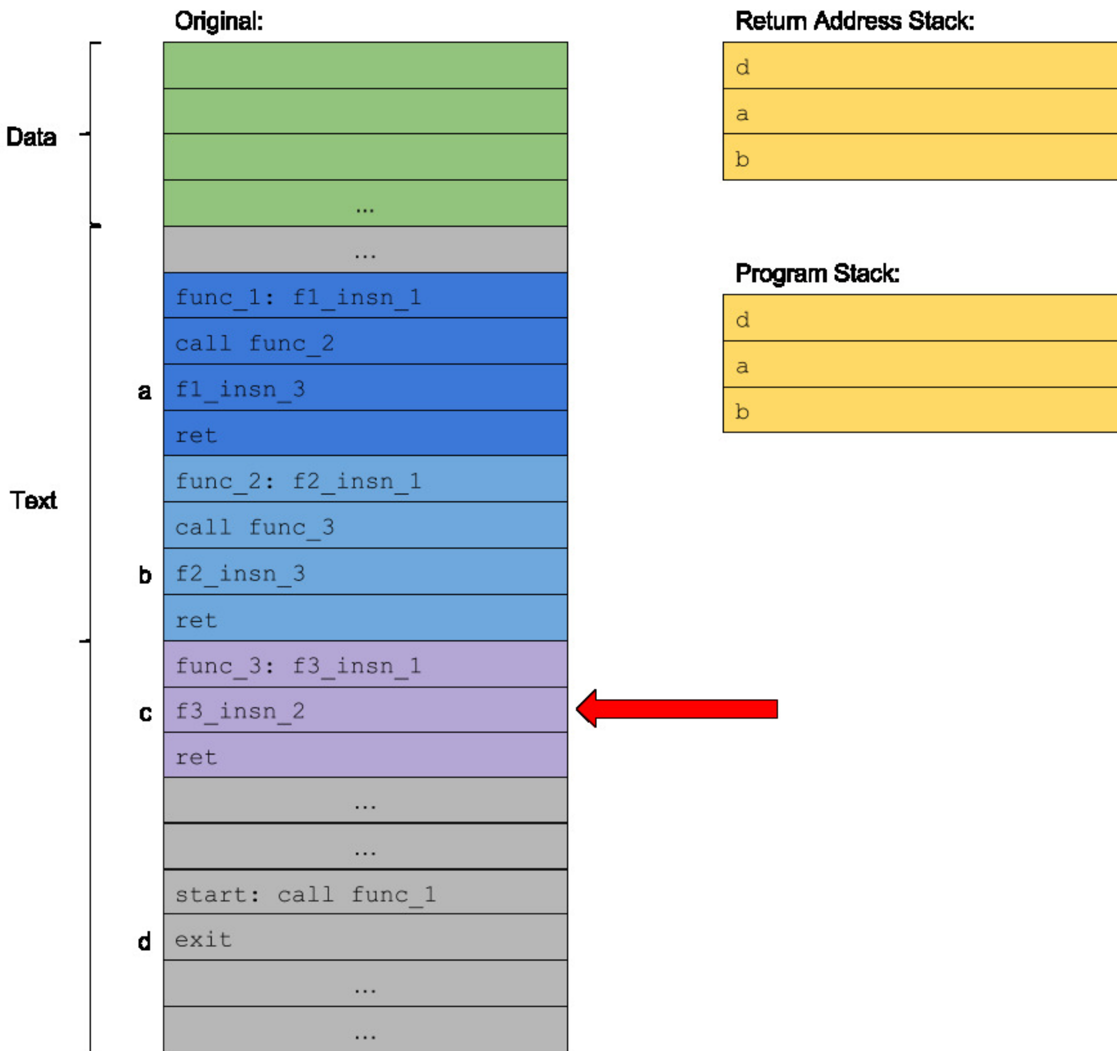

Figure 12.3: The contents of the program and return address stacks at runtime.

destination of a return at runtime and speculatively begin executing instructions at the return site even before the return is reached [226, 79, 107].

The SelectiveCFI instrumentation removes the benefits of the return address predictor because it replaces all `calls` with `push/jmp` combinations in order to separate the call from the fallthrough instructions in order to make space for the nonce decorations before the target of the return indirect program control flow operation.

A way to enforce CFI that can take advantage of return address prediction would significantly decrease the performance overhead of SelectiveCFI. Executable nonces make it possible to retain the enforcement of CFI and gain the benefit of the return address predictor. Consider the example in Figure 12.3 that shows a program with three nested functions. Functions 1 and 2 have been invoked for the program to

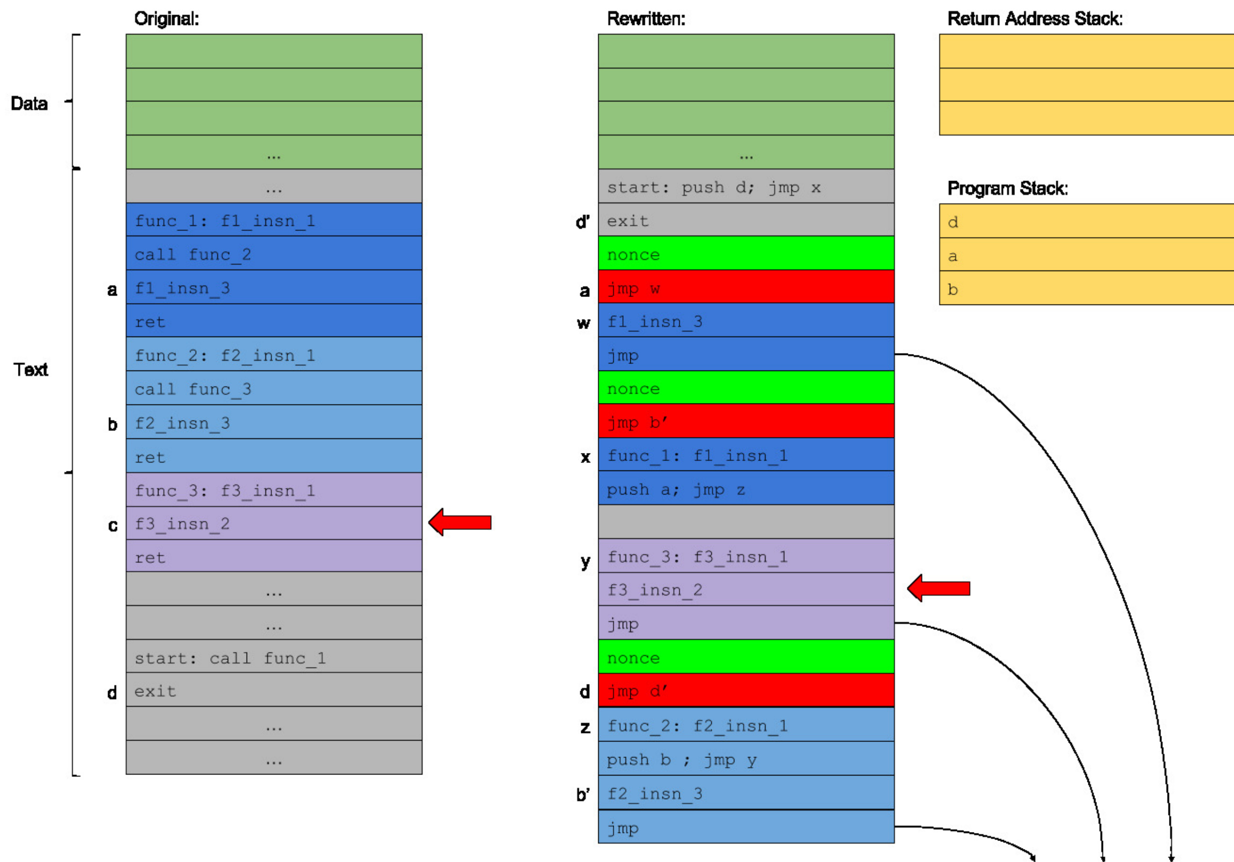

Figure 12.4: The traditional method of SelectiveCFI instrumentation of `ret`s in a program. This enforcement method does not take advantage of the return address predictor.

execute `f3_insn_2`. The contents of the program and return address stacks are shown in gold. The CPU will speculatively begin executing the instructions at *b* in anticipation of the upcoming return instruction. In so doing, the CPU has avoided stalling the processor because the return addresses in the program stack match the addresses in the return address stack as the returns are executed.

Two versions of the SelectiveCFI-enforced version of the same program are shown in Figures 12.4 and 12.5. The version of the program shown in Figure 12.4 is enforced with the traditional SelectiveCFI enforcement mechanism. Because all of the `call`s have been translated to `push`/`jmp` combinations for the sake of making room for the decorations, the return address stack is completely unused. Although not shown in this figure, the enforcement code for the return instructions translates those `ret`s into `jmp`s. See Figure 12.2 and the accompanying explanation for additional details on the traditional SelectiveCFI enforcement instrumentation.

The version of the program shown in Figure 12.5 uses *executable nonces*. There are several differences to notice between the two enforcement mechanisms. First, the `call` instructions remain. The `call` instructions are not converted to a `push`/`jmp` combination. Second, rather than requiring the return addresses to be

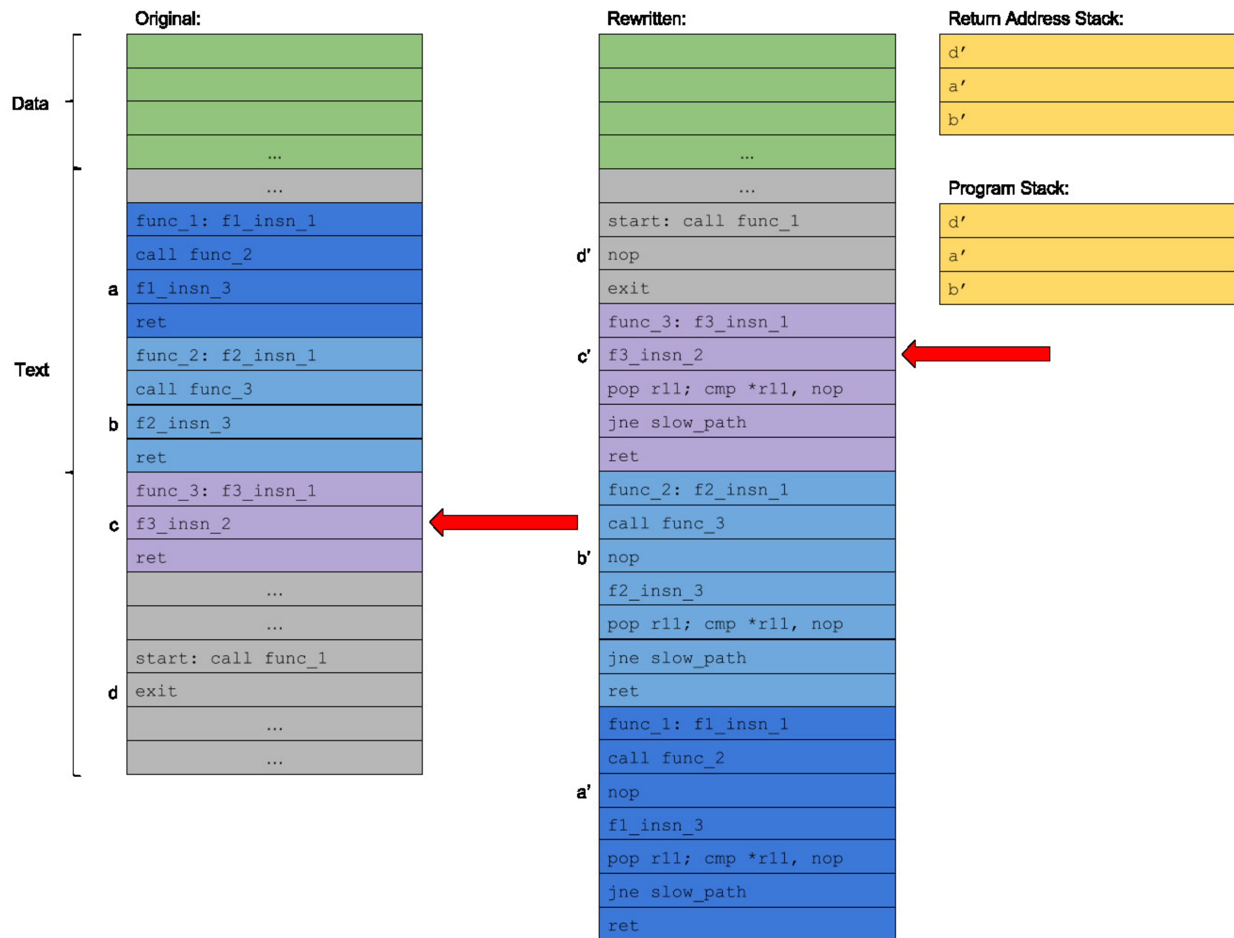

Figure 12.5: The SelectiveCFI instrumentation of `ret`s in a program using executable nonces. This enforcement method is able to take advantage of the return address predictor.

separated from the `call` instructions (and pinned), the instruction after each `call` is a `nop` instruction, an executable nonce. Third, and finally, the enforcement mechanism uses a `ret` rather than an indirect `jmp`. CFI enforcement at each return needs only to verify that the instruction at the address on the top of the program stack is the nonce value. Once the enforcement is done, program execution can continue as normal.

The overall affect of retaining the `call` and `ret` instructions is that the CPU's return address predictor can work efficiently and improve runtime performance (see Section 5.5.1).

12.3 Related Work

In 2005, researchers from the University of Santa Cruz, Princeton University and Microsoft Research presented a security technique to protect insecure applications from being exploited by "powerful adversaries that have full control over the entire data memory of the executing program" [22]. Their work, known as Control-Flow

Integrity, *enforces* a policy on an application to ensure that, at runtime, the program does not deviate from its CFG, as determined during offline, static analysis. In other words, "[t]he CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time."

At runtime, enforcement of CFI means that when a program transfers control, the target of that program control transfer must be a valid target as determined by offline, static analysis. The authors explore two different techniques for actually implementing such a mechanism for programs executing on an x86 processor and explore the tradeoffs for each.

Both techniques involve *instrumenting* the target program by adding data/instructions around the sources and targets of program control flow transfer. In order for their techniques to provide security to as much deployed software as possible, the authors build a technique that is applicable to existing software even in the absence of source code or other metadata. The process for instrumenting a program to add CFI enforcement is based on adding unique identifiers to the program's control flow transfer points.

In the first method, the target program is instrumented at the transfer site using a *label*. The code at the original transfer target location is shifted by the length of the unique identifier and replaced with the bytes of the unique identifier. At the call site, the contents of the memory at the original target address are compared against a copy of the unique identifier embedded within the comparison instruction.

The second method is very similar to the first method, but removes a subtle attack vector. In the first method, because the bytes of the unique identifier are embedded in the comparison instruction that verifies the target, the instruction after those bytes is a valid control transfer target as far as CFI is concerned. To correct this, the second method embeds a slightly modified version of the unique identifier in memory at the program control transfer source, loads that into a register and modifies it before doing the comparison with the unique identifier that marks valid targets.

The authors point out that, unlike other mechanisms to improve software security (e.g., stack canaries, ASLR, etc.), CFI is more than a mitigation. CFI enforcement provides provable guarantees about the state of program execution at any point during its execution, no matter how powerful the adversary. In order to make these claims given either of these two implementations of CFI enforcement, the authors rely on three important assumptions:

1. There can be no overlap in the unique IDs assigned to program call sites/targets.
2. There can be no writable program text memory because the values for comparison are embedded in the program's text and it must be known that those values are immutable throughout execution.
3. There must not be executable data segments which give an attacker the power to write instructions into memory that are properly tagged with a unique ID.

These assumptions are reasonable but do mean that the technique is inapplicable to certain types of software: any programs that do just in time compilation, load code dynamically, etc. While this makes the their technique less than universally applicable, the authors assert that most software is "rather static".

There is a fourth assumption that underlies the static analysis used to build the *a priori* CFG used to determine the valid sets of targets for any given program control transfer source. For any program control transfer source within the program, the statically discovered targets of that transfer must be equivalent. This is not always the case, especially for object-oriented languages where different targets are reached depending on an object's type. Although this presents a real opportunity to defeat CFI enforcement, the solution is relatively straightforward: duplicate code. The authors liken this technique to function inlining and demonstrate that it works to mitigate the particular attack but also admit that it may cause additional overhead at runtime.

To evaluate the performance impact of their CFI enforcement mechanism, the authors tested their implementation using SPEC2000. On average, the filesizes of the instrumented programs increased by 8%. The performance overhead was as high as 45% but averaged 16% across the entire suite.

In order to demonstrate the efficacy of the security properties gained with their CFI implementation, the authors examined "famous" vulnerabilities to determine whether CFI would have provided a successful defense. In the case of the vulnerabilities that relied on bugs in parsing user data that allowed the program to maliciously launch separate applications, CFI does not apply and would not have added additional protection. On the other hand, however, for those vulnerabilities that the authors examined that relied on "pointer subterfuge", their implementation would have prevented the exploit of those bugs.

To prove their assertions, the authors used an industry-standard set of security benchmarks that include several programs vulnerable to shell injection attacks. After application of their CFI enforcement mechanism, none of the 18 programs were vulnerable.

The authors also present case studies where CFI enforcement is used as a primitive to build higher-level security techniques. For instance, the authors use CFI to enforce the code that implements the monitors in an inlined reference monitor. The authors also demonstrate how CFI can be used as a primitive for developers to securely implement software memory access control where access to certain segments of program memory are strictly controlled.

12.3.1 BinCFI

While others implemented CFI using dynamic binary translators (e.g., [169]) or static binary rewriters that required debugging symbols and/or relocation information (e.g., [48]), in 2013 Zhang et al. described "a

technique for applying CFI to stripped binaries on x86/Linux" [251] that was the first to work on "complex shared libraries such as `glibc`" [251].

BinCFI implements their technique in two stages. First, BinCFI disassembles the input program. Second, BinCFI instruments and rewrites the program to enforce the CFI policy.

BinCFI performs disassembly using a combination of linear and recursive methods (see 4.2.1). The input program is subjected to linear disassembly first and the results are refined using recursive methods. Through the combination of these techniques, BinCFI disambiguates code and data and identifies the IBTs.

BinCFI classifies IBTs by categories:

Return Addresses The addresses of instructions immediately after a `call`;

Exception Handler Addresses The address of instructions invoked from the exception handling code;

Symbol Addresses The addresses of exported functions available to users of the program/library if it is loaded dynamically;

Computed Code Addresses The addresses of instructions computed as targets in jump tables;

Code Pointer Values The addresses of functions (or sequences of instructions) embedded in the program/library.

BinCFI implements two separate policies to enforce CFI: return IBT instructions and indirect jumps can validly target Return Addresses, Exception Handler Addresses, Symbol Addresses, Computed Code Addresses and Code Pointer Values; indirect calls and calls from the program-linkage table (PLT) can validly target Symbol Addresses, Code Pointer Constants and Computed Code Addresses.

The enforcement instrumentation uses a closed hashing method to encode the sets of valid targets of an indirect control flow transfer operation in a table. Using TLS to hold the target of the control flow transfer, BinCFI consults the table to determine the validity of the transfer. For targets in the table, control flow is validated; for targets not in the table, the control flow transfer operation is blocked and the program is halted. There is another table, the global translation table, to make BinCFI-protected dynamic libraries compatible with unprotected programs.

Using the programs of the SPEC benchmark suite for evaluation, BinCFI instrumentation introduces 8.54% runtime overhead, 139% filesize overhead and 2.2% RSS overhead during execution. While the small amount of runtime overhead introduced by BinCFI is impressive, the filesize overhead is too great to consider BinCFI for protecting programs deployed in the contexts described in Part I and is greater than the filesize overhead of SelectiveCFI (see Section 12.4.2).

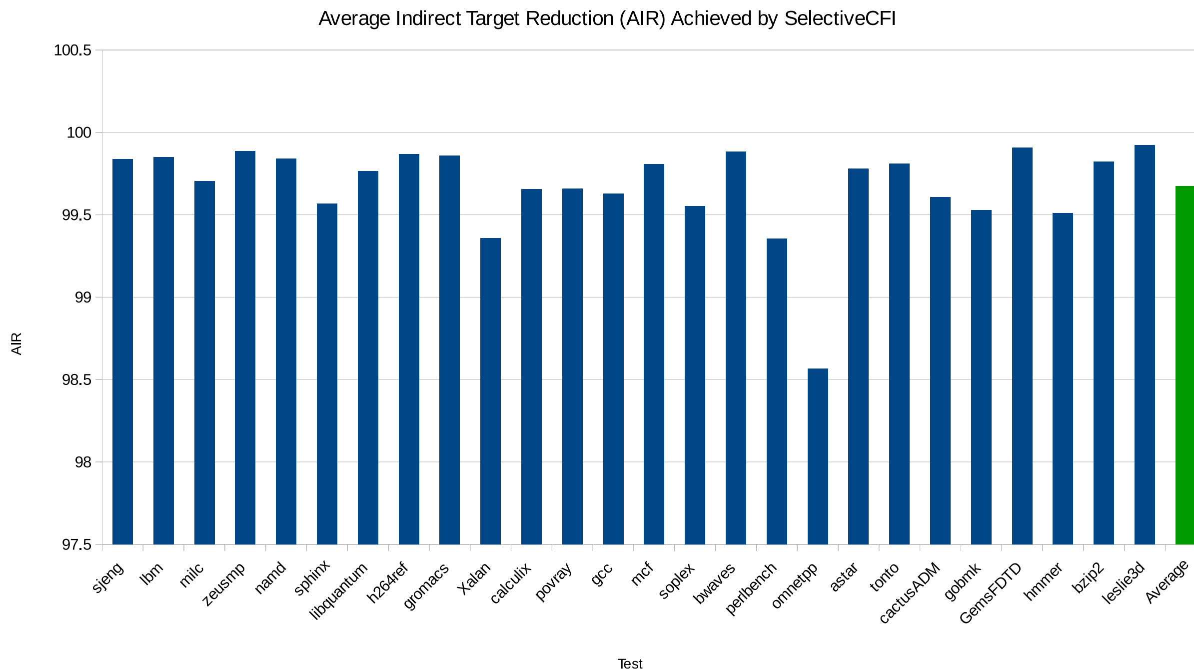

Figure 12.6: AIR for the programs of the SPEC benchmark suite protected with the basic implementation of SelectiveCFI.

Besides lower filesize overhead, SelectiveCFI offers better protection than BinCFI. In their work, Zhang et al. defined the average indirect target reduction (AIR) to quantify the effectiveness of the protection offered by an implementation of CFI [251]. AIR is an average of the reduction in the size of the valid targets for each of the indirect control flow transfer operations in an input program. For example, assume that indirect control flow transfer operation i in an unprotected program can reach T IBTs at runtime but i in the CFI-protected program can only reach t IBTs. In this case, CFI enforcement eliminates $100(1 - \frac{t}{T})$% of potential targets. The AIR for an entire program is defined as

$$\frac{1}{n}\sum_{j=1}^{n}(1 - \frac{|T_i|}{S}) \quad (12.5)$$

where the input program has n indirect control flow transfer operations $i_1 \ldots i_n$, operation i_j validly targets addresses in the set T_j and S is “the number of possible [IBTs] in [the] unprotected program” [251]. For platforms with variable-length instructions that can be as short as a single byte (e.g., the x86 platform), S “is the same as the size of the binary” [251].

For the programs of the SPEC benchmark suite, the average AIR for BinCFI is .9886. As shown in Figure 12.6, the average AIR for SelectiveCFI on the same programs is .9967.

12.3.2 BinCC

Wang et al. described BinCC in 2015 as an improvement on the protection offered by BinCFI [231]. The authors argue that the BinCFI enforcement policies are too permissive and proposed stricter enforcement policies based on code continents.

Code continents are groups of instructions invoked directly by one another. Every instruction in a code continent is known as a node. There are root nodes, boundary nodes and inner nodes. Root nodes are IBTs; boundary nodes are indirect program control transfer operations that target other code continents; and inner nodes are instructions that do not affect the PC or are direct or indirect control transfer operations that target nodes within the code continent.

BinCC enforces intra- and inter-continent CFI policies. The targets of inner nodes are restricted to nodes within the same code continent (the intra-continent policy). The targets of boundary nodes must be root nodes (for indirect calls and jumps) or other boundary nodes (for returns) (the inter-continent policy).

As a result of the construction of code continents, the sets of valid targets for indirect control flow transfer operations in BinCC are more granular than the sets in BinCFI. Besides the difference in the constitution of the sets of valid targets, BinCC is so similar to BinCFI that BinCC uses BinCFI for implementation.

To compare BinCFI with BinCC, Wang et al. presented the AIR for the same set of programs. BinCC improves AIR to .9954 from .9886 in BinCFI. The runtime overhead for BinCC on the programs of the SPEC benchmark suite is 22% and the filesize overhead is 125%. BinCFI's filesize overhead is worse than that of SelectiveCFI, its average AIR is lower than that of SelectiveCFI while BinCFI's runtime overhead is only slightly better than that of SelectiveCFI (see Section 12.4.2). As with BinCFI, the filesize overhead of BinCC makes it inapplicable to the operating contexts described in Part I.

12.3.3 Opaque CFI

Opaque CFI (O-CFI) developed by Mohan et al. in 2015, combines the benefit of CFI with ASLR [152]. O-CFI enforces coarse-grained CFI policies to lower the instrumentation runtime overhead but adds code randomization so that the overall security of the technique matches the security offered by fine-grained CFI implementations.

The core of the O-CFI technique is a static binary rewriter that groups the valid targets of indirect control flow operations near one another. By grouping valid targets together, enforcement can be done by performing a simple bounds check on the target address of every indirect control flow operation. The boundary values for each set of valid targets are stored in memory at an address chosen randomly when the program/library is loaded and are not embedded as immediates. Even if an attacker succeeds in downloading the program

code as it exists in memory at runtime, the table is at a random location and is indistinguishable from other code/data.

As described earlier, the sets of valid targets are not mutually exclusive (see Section 12.2.2) which means that the bounds checks may not guarantee precise CFI enforcement. Code randomization makes up for this imprecision. When a program is loaded into memory at startup, the instructions within the sets of targets are randomized which makes it impossible for an attacker, even one who has the power to read the entire text of the program as it exists in memory after randomization, to know which targets will escape enforcement and can be used in an attack.

Because the enforcement of the CFI policies at runtime are simple bounds checks, its implementation can be highly efficient by using certain processor instructions. On a subset of the programs of the SPEC 2000 benchmark suite,[1] the authors measured O-CFI runtime overhead at 4.7%. In the future, the authors reported that the instructions used in their implementation will be hardware accelerated in future processors and overhead will continue to decrease.

Because it is a hybrid of randomization and CFI, the AIR for O-CFI is an invalid way to compare it with existing techniques. The runtime overhead for O-CFI is lower than that of SelectiveCFI as it was tested for the experiments whose results are presented in this research. However, optimizations to the design of the static rewriter made subsequent to the preparation of this dissertation give reason to believe that the overhead of SelectiveCFI can be improved [96]. The on-disk overhead of O-CFI (137% on average for a subset of the programs of the SPEC benchmark suite) makes it inapplicable to the operating contexts described in Part I.

12.3.4 Object Flow Integrity

One of the problems with CFI is incompatibilities among different instrumentation techniques. A related but more important problem, however, is the inability of an instrumented program to interact with unprotected libraries. The latter is less of a problem for the Linux platform where applications can statically link support libraries than it is on the Windows platform where large system services (windowing, printing, etc.) are only accessible through a common object model (COM) interface [235].

Object Flow Integrity, described in 2017 by Wang et al., solves this problem for the Windows platform and maintains the protection of CFI across invocations of trusted system libraries loaded and invoked dynamically by applications [235].

On the Windows platform, the functions of system libraries are often invoked with objects that control behavior of the function or transfer data. As a result, the functions in third-party code often indirectly invoke

[1] `gzip`, `vpr`, `mcf`, `parser`, `gap`, `bzip2`, `twolf`, `art` and `equake`

the methods of those objects. Because the libraries themselves are trusted and signed by the OSes and cannot be modified, they cannot be instrumented with CFI. As a result, the user application itself may be subject to control flow hijacking even if it is protected with CFI.

Object Flow Integrity solves this problem with read-only proxy objects that enforce CFI policies for the objects passed between the user application and the system libraries. The authors showed the effectiveness of the technique through testing on several well-known Windows applications (e.g., Notepad). In a set of tests automated to emulate interactive use of these applications, the authors reported that their technique incurs negligible runtime overhead (1.82%) but significant filesize overhead (11% to 209%). On a set of microbenchmarks that measure the worst-case execution overhead of Object Flow Integrity (i.e., places where control is transferred back and forth between application and system library), the runtime performance overhead is 32.44%.

It is difficult to compare the performance of Object Flow Integrity with that of SelectiveCFI. The authors do not present results for the applications of the SPEC benchmark suite nor do they present an AIR value. Subsequent to the research presented in this dissertation, SelectiveCFI has been extended to add multi-module support which solves a problem similar to that solved by Object Flow Integrity.

The number of different implementations of CFI demonstrate that the technique is useful for adding protection to potentially vulnerable programs/libraries. The implementations described here each have their own benefits but none meet the constraints of the operating context outlined in Part I. The lower runtime overhead of the techniques described here do provide avenues for future work on the implementation of SelectiveCFI for ways to improve its performance.

12.4 Evaluation

12.4.1 Security

The performance evaluation in this dissertation has relied greatly on the materials built for and collected by the DARPA CGC. The utility of those materials, however, goes beyond performance evaluation.

Evaluation of software security techniques is complicated by the inherent difficulty of reconstructing particular installations/configurations required to reproduce real-world exploits. There are several collections of test cases designed to address this problem (e.g., [66]) but there are still opportunities for additional work in this area. Clark et al., in 2010, wrote about their extensive efforts to simply gather a *list* of vulnerabilities and their descriptions [51].

As described previously, each CB included in the DARPA CGC dataset contains at least one security vulnerability. More importantly, each CB contains a set of inputs that trigger that vulnerability The so-called proof of vulnerability (PoV) is similar to a poller but is designed specifically to trigger the program's vulnerabilities. In combination with the fact that DECREE is an operating environment designed specifically to support reproducibility, the CBs and associated PoVs make the dataset particularly well-suited for security research. In fact, the security research organization Trail of Bits went so far as to say, "[t]he challenge binaries, valid test inputs, and sample vulnerabilities create an industry standard benchmark suite for evaluating: [b]ug-finding tools[,] [p]rogram-analysis tools (e.g. automated test coverage generation, value range analysis)[,] [p]atching strategies [and] [e]xploit mitigations" [17].

CROMU_00009, a CB from the DARPA CGC, is a excellent example of a vulnerable program that can be secured by applying the SelectiveCFI defense. CROMU_00009 is a "RAM-based filesystem [which] provides a simple shell to interact with the filesystem" [182]. The program contains two vulnerabilities: a NULL pointer dereference (CWE-476) and an out-of-bounds write (CWE-787). When exploited, the latter vulnerability gives the attacker the power to overwrite the top entry on the stack which, in turn, allows for control of the instruction pointer.

The two PoVs included with CROMU_00009 exploit the vulnerabilities. One of the PoVs triggers the NULL pointer dereference, a vulnerability and exploit that SelectiveCFI does not prevent. The other PoV triggers the out-of-bounds write and sets the instruction pointer to an address of the attacker's choice.

The PoV controls the instruction pointer by using the out-of-bounds write vulnerability to set the top four bytes of the program stack to `0x0`. At the next function return after the value at the top of the stack is rewritten, the `ret` instruction uses the attacker-chosen value as the target for the transfer of program control. In the vulnerable program, the `ret`, in combination with the `0x0` on the top of the stack, transfers program control to an invalid address. When the PoV is applied to CROMU_00009 protected by SelectiveCFI, the program detects that the `0x0` target is not valid and the attack is thwarted.

The PoV included with CROMU_00009 was rewritten to test whether SelectiveCFI could detect the situation where an attacker exploits the out-of-bounds write to put the address of an existing function, `ExitHandler`,[2] at the top of the stack and, eventually, in the instruction pointer to incorrectly transfer control to that function. When protected with Basic SelectiveCFI, any indirect control transfer operation can validly transfer program control to any IBT (see Section 12.2.1). Therefore, the program protected with Basic SelectiveCFI will not detect the modified attack and will improperly allow transfer of control to the function of the attacker's choice – `ExitHandler` in this case.

[2] `ExitHandler` is a function invoked by CROMU_00009 when the user enters the `exit` command at the shell prompt. It was chosen for this example for didactic purposes.

This scenario is precisely the reason for the implementation of SelectiveCFI with Coloring. Protecting the program with the more granular version of SelectiveCFI restricts the set of valid targets at program control transfer operations to a subset of all the IBTs. The subset is determined statically during the analysis phase (see Section 12.2.1). Upon protecting CROMU_00009 with SelectiveCFI with Coloring, the attack is detected. At the next function return after the stack value is rewritten with the address of the `ExitHandler`, the `ret` instruction uses the attacker-chosen value as the target for the transfer of program control. In the vulnerable program *and* the vulnerable program protected by Basic SelectiveCFI, the `ret`, in combination with the `ExitHandler`'s address on the top of the stack, transfers program control to the function. When the PoV is applied to CROMU_00009 protected by SelectiveCFI with Coloring, the program detects that `ExitHandler` is not a valid target for the `ret` in question and the attack is thwarted.

Examining this particular program from the DARPA CGC dataset demonstrates the power of the SelectiveCFI technique and highlights the differing levels of protection offered by Basic SelectiveCFI and SelectiveCFI with Coloring. Evidence from the DARPA CGC CFE shows that Basic SelectiveCFI is effective at preventing a variety of attacks on vulnerable software [146, 145]. However, applying SelectiveCFI to the other available security testsuites will allow for quantification of difference in effectiveness between the two types of SelectiveCFI.

12.4.2 Performance

Underlying the implementation of SelectiveCFI are runtime checks performed at every point in the program where control flow is transferred. Because of the additional check at each indirect control flow transfer instruction, it is reasonable to expect that program performance will be negatively affected. Further, it is reasonable to expect that the code to implement the runtime protection checks will increase the maximum RSS at runtime and the on-disk size of the statically rewritten program/library. Quantifying how much performance will suffer is important for determining whether SelectiveCFI is efficient enough for system architects to apply the protection to SOUP on high performance server applications, embedded software and everything in between.

CFI Only

Figures 12.7 and 12.8 show the performance impact of SelectiveCFI on the statically rewritten versions of the programs of the SPEC benchmark suite. On average, the performance overhead on Hosts A and B are 1.25x and 1.24x, respectively, which are 5.24% and 1.30% improvements, respectively, over the performance

overhead for the programs of the SPEC benchmark suite rewritten using the default Reassembly algorithms when transformed with only the Null Transformation (see 5.3).

Because of the execution of additional conditionals at each indirect program control flow transfer instruction, a performance *improvement* is unexpected. There are three reasons for this surprising result. First, most of the time executing the SPEC benchmark programs is in computational kernels that do not contain protected indirect program control flow transfer instructions. Therefore, the impact of the additional checks at indirect program control transfer instructions is minimal. If this were the only explanation, it would be reasonable to expect equivalent performance but not an improvement.

The second reason involves the hardware return address stack. As described extensively above, the hardware optimizes for matches between `call`s and `ret`s. The Null Transformation fixes most of the `call`s and converts them to `push`/`jmp`s but leaves all `ret`s in place. This mismatch defeats the return address stack optimization and hurts performance. Therefore, the baseline for the comparison in this case is pessimistic.

The SelectiveCFI implementation also converts `call` instruction into the combination of `push`/`jmp`s but, as opposed to the Null Transformation, converts `ret` instructions into `pop`/`jmp` combinations to check the safety of the potential target. Not only does this change neutralize the performance penalty of mismatched `call`/`ret`s, it actually improves performance by taking advantage of the hardware's branch predictor.

The third and final reason involves a limitation of the Basic SelectiveCFI implementation. The fastpath code used to check the nonce decorations at a target uses register `r11`. That register is always dead across function calls so using it to check the value of the nonces when returning from functions is always safe. However, that register is not always dead across jumps. Therefore, this implementation can only test the nonce values of indirect jumps when static analysis can determine that the register is dead at the time of the check. The current implementation of the static analysis used to determine whether the register is dead at the indirect jumps is conservative. So conservative, in fact, that almost no indirect jump is protected (0.159% and 0.154% of the indirect jumps are protected on Hosts A and B, respectively). Therefore, the performance penalty for checking the safety of the more than 2000 indirect jumps in the programs of the SPEC benchmark suite does not factor into the overhead calculations. These three reasons explain the performance improvement seen on Hosts A and B.

One of the priorities from the beginning for the design and architecture of the static binary rewriter described in this dissertation was for its algorithms to emit reassembled programs/libraries with minimum on-disk overhead (see Part I). In the same way, SelectiveCFI cannot add excessive on-disk overhead to the protected programs/libraries if the goal is to build a tool that security architects can deploy on embedded platforms. Adding nonces and code to enforce that indirect transfers target only decorated targets will necessarily increase the on-disk filesize but it is important to know by how much the size will increase.

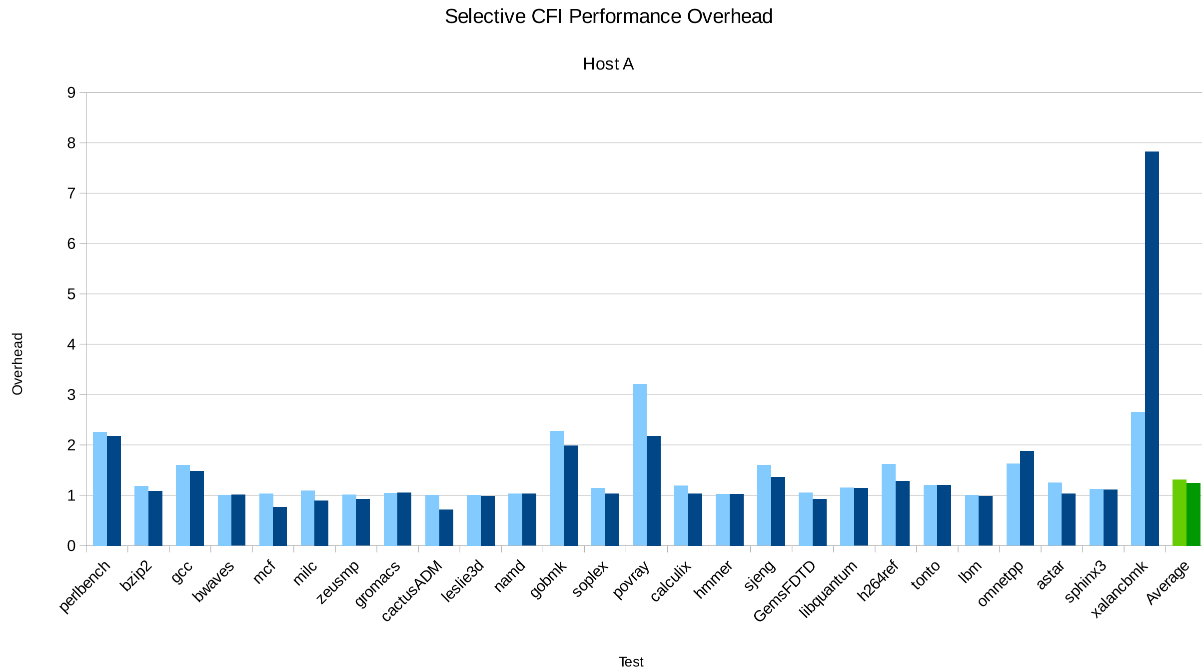

Figure 12.7: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host A. The results for the applications protected with the basic version of SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase when transformed with the Null Transformation are shown on the left, in light blue and light green.

Figures 12.9 and 12.10 show the impact of SelectiveCFI on the on-disk size of the statically rewritten versions of the programs of the SPEC benchmark suite. On Host A, the programs of the SPEC benchmark suite protected by SelectiveCFI have a 1.15x on-disk size overhead when compared to their native counterparts. That is 5.76% larger than the applications of the SPEC benchmark suite rewritten using the default algorithms of the Reassembly phase transformed using the Null Transformation. On Host B, the programs of the SPEC benchmark suite protected by SelectiveCFI have a 1.14x on-disk size overhead when compared to their native counterparts. That is 6.13% larger than the applications of the SPEC benchmark suite rewritten using the default algorithms of the Reassembly phase transformed using the Null Transformation.

As discussed in Section 12.2.2, the process of protecting a program using SelectiveCFI introduces three sources of on-disk overhead: the nonces themselves and the fast- and slow-path code for checking nonce values at each indirect program control transfer instruction.

It is possible to calculate precisely the on-disk overhead for the fast-path code for checking nonce values at each indirect program control transfer instruction. For the implementation on the x86 platform, each protected `ret` adds 15 bytes of overhead and each protected `jmp` or `call` adds 16 bytes. Figure 12.11 shows the percentage of the marginal size on-disk overhead of programs secured with the Basic SelectiveCFI as

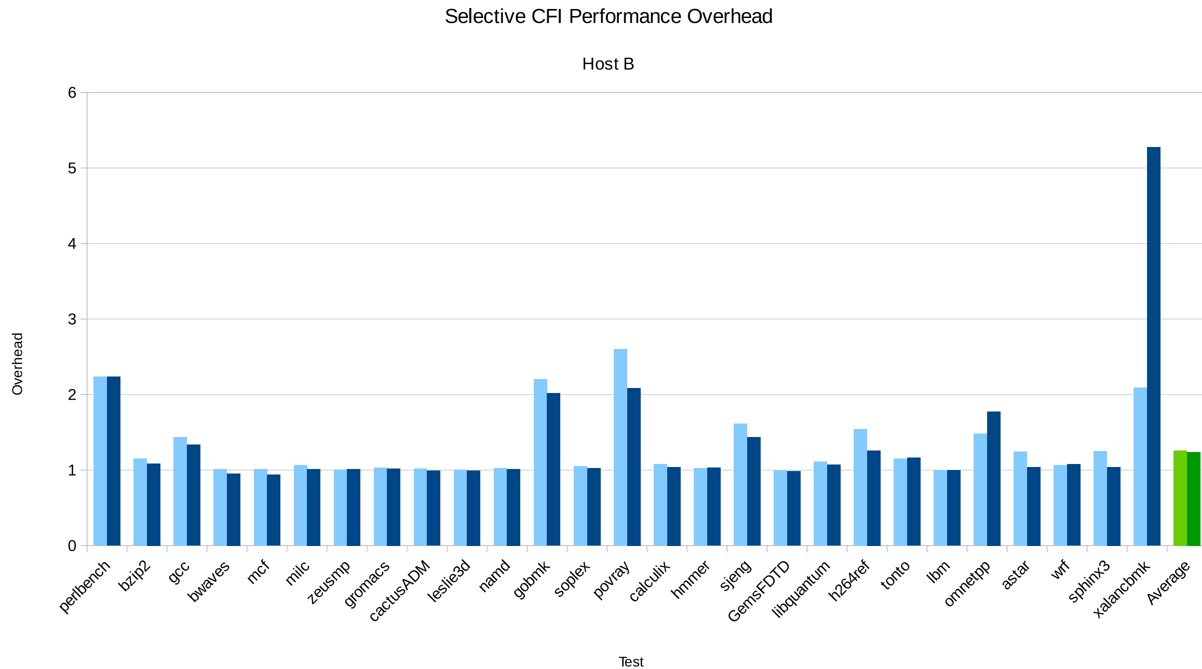

Figure 12.8: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host B. The results for the applications protected with the basic version of SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase when transformed with the Null Transformation are shown on the left, in light blue and light green.

compared to the programs transformed with the Null Transformation attributable to the fast-path code for checking nonce values at each indirect program control transfer instruction. The `leslie3d` benchmark is not included because its on-disk size when protected with the Basic SelectiveCFI is less than its size when simply statically rewritten with the Null Transformation.[3] It is evident from these data that for most of the programs of the SPEC benchmark suite, a majority of the additional overhead introduced by the SelectiveCFI implementation is due to the fast-path code for checking the security of the targets at indirect program control transfer instructions. This result demonstrates that there may be a tradeoff between placing a fast-path at each such instruction and consolidating the target safety code in a single place in the program. Allowing the security architect to specify this as an option to the SelectiveCFI deployment is the subject of future work.

As described in Section 12.2.2, the safety of transfers that target instructions that can be decorated with nonces is checked efficiently with only a few instructions. When a target cannot be decorated with a nonce, the safety of a transfer to the target must be determined using code along a slow path. Therefore, the percentage of targets that cannot be decorated with nonces has a direct impact on the runtime performance of the program/library protected with SelectiveCFI. Figures 12.12 show the percentages of time that targets

[3]This occurs because of the non-determinism inherent in the default algorithms of the Reassembly phase (see Section 4.4).

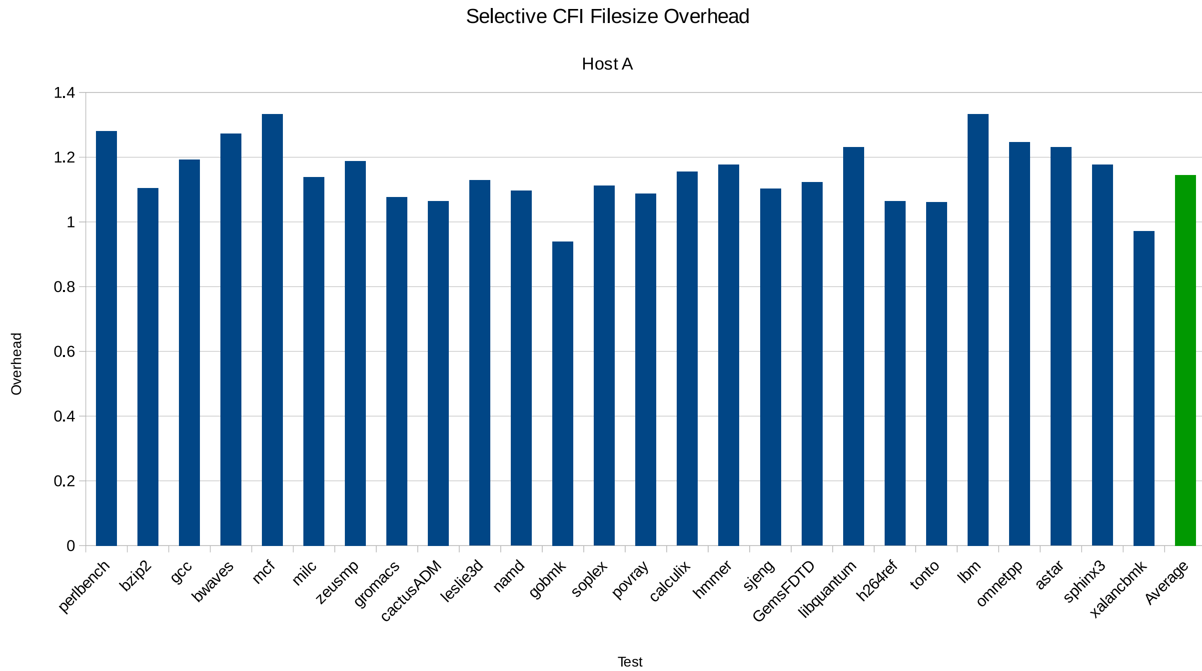

Figure 12.9: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host A.

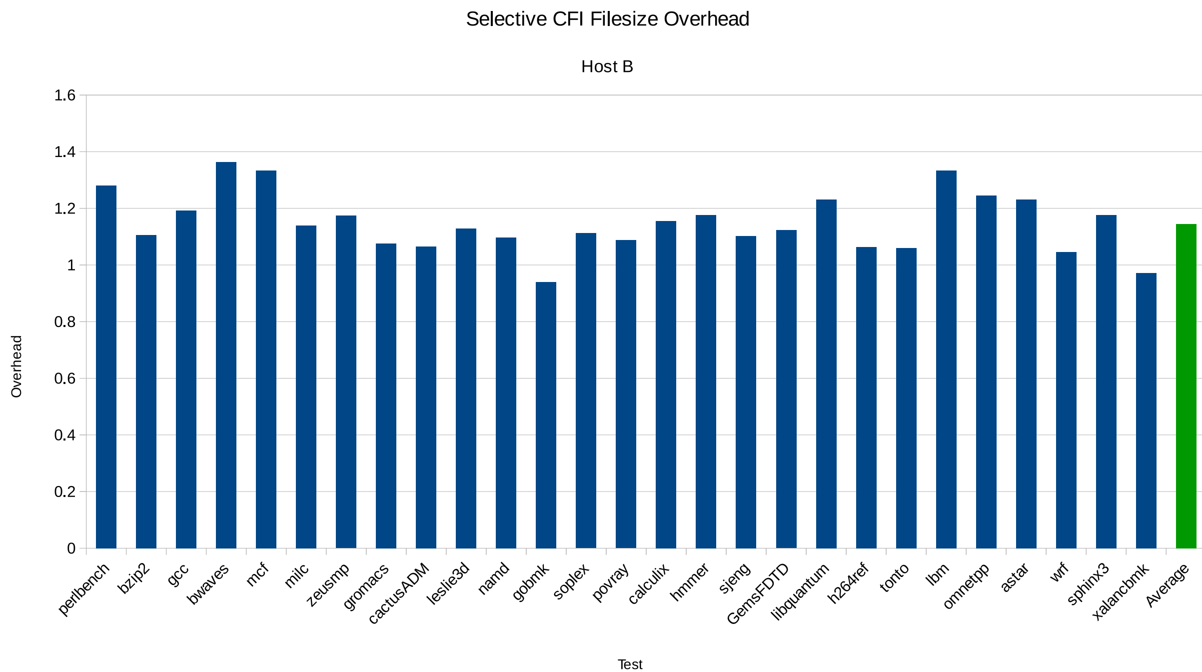

Figure 12.10: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using the basic version of SelectiveCFI. These results are for Host B.

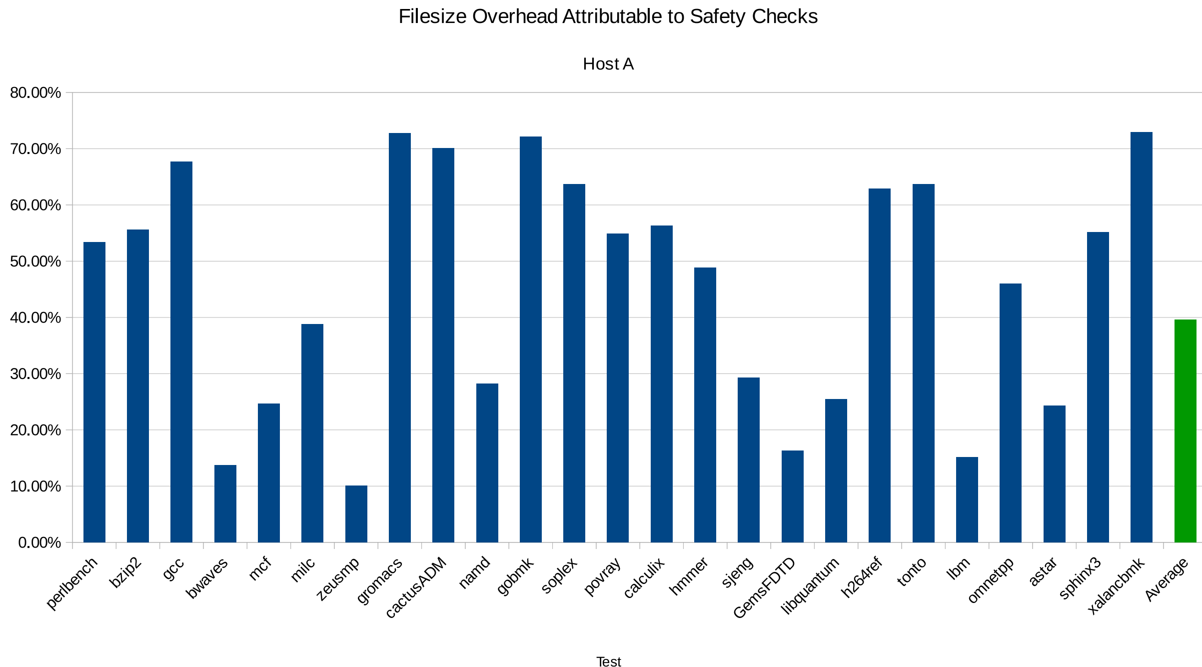

Figure 12.11: Percentage of the marginal overhead of programs secured with Basic SelectiveCFI as compared to programs transformed with the Null Transformation attributable to the code required to check the safety of targets.

in the programs of the SPEC benchmarks suite can be decorated with nonces. On average, targets can be decorated more than 97% of the time on both Hosts A and B. The ability to decorate targets is based on the layout of the program/library which is governed by the compiler. Because the compilers used on both hosts for these experiments is identical, it is expected that the placement ratios will be nearly identical.

CFI Without Protecting Safe Indirect Transfers

As described extensively in Section 12.2.1, the possibility of gaining the security of CFI without having to check the safety of every indirect program control transfer opens up the chance for lessening the performance overhead associated with the security gained with the technique. One of the options for decreasing the number of run-time checked indirect program control transfers is to not protect `ret`s from safe functions (see Section 12.2.1) or indirect jumps (see Section 12.2.1). The results below explore the performance impact of skipping safety checks on `ret`s from functions statically deemed safe. As a shorthand, this implementation of SelectiveCFI will be referred to as "Safe SelectiveCFI".

The evaluation of the performance of Safe SelectiveCFI using the SPEC benchmark suite used the alternate method for protecting the return indirect control transfer operations (see Section 12.2.2). The choice of the alternate method bypasses the hardware's return address predictor method and comparing the results in

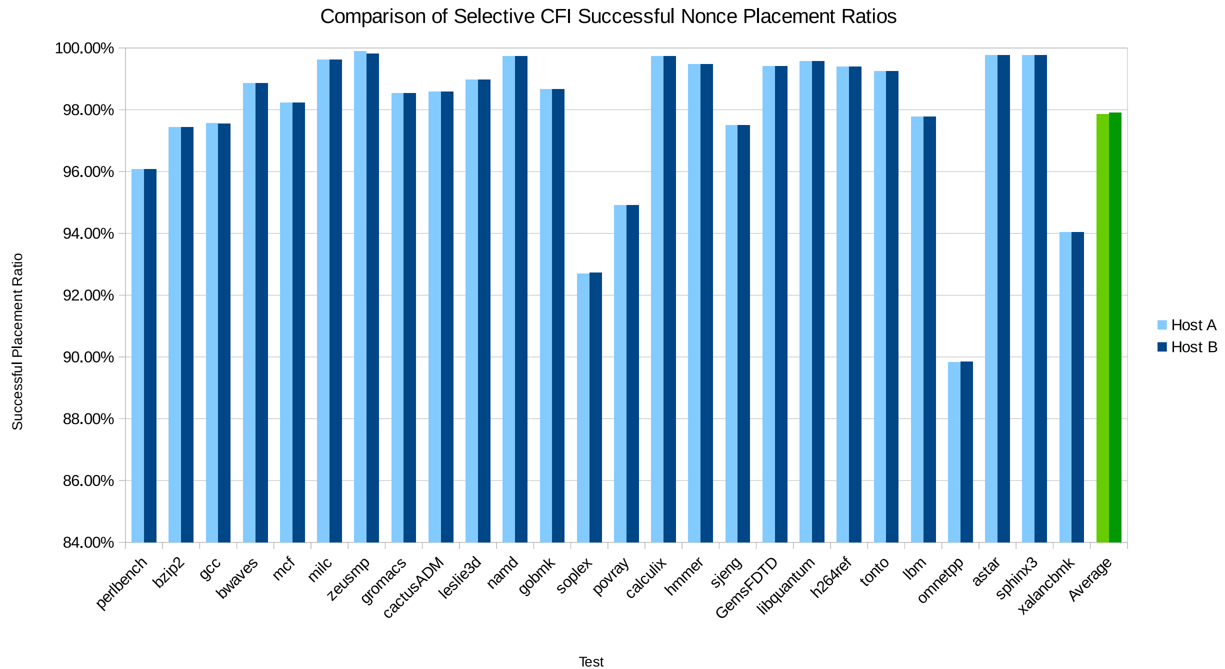

Figure 12.12: Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with the basic implementation of SelectiveCFI on Hosts A and B.

this section with the results in Section 12.4.3 will demonstrate the relative merits of the two methods of instrumentation.

Figures 12.13 and 12.14 show the performance impact of Safe SelectiveCFI on the statically rewritten versions of the programs of the SPEC benchmark suite. On average, the performance overhead on Host A is 1.21x which is a 2.48% improvement for the programs of the SPEC benchmark suite protected with the Basic SelectiveCFI. On average, the performance overhead on Host B is 1.23x which is an improvement of 1.03% compared with the performance overhead for the programs of the SPEC benchmark suite protected with the Basic SelectiveCFI.

Again, one of the priorities from the beginning for the design and architecture of the static binary rewriter described in this dissertation was for its algorithms to emit reassembled programs/libraries with minimum on-disk overhead (see Part I). Figures 12.15 and 12.16 show the impact of Safe SelectiveCFI on the on-disk size of the statically rewritten versions of the programs of the SPEC benchmark suite. The on-disk size overhead for Hosts A and B are 1.15x and 1.14x, respectively, when compared to their native counterparts. That is a difference of less than 1% than the applications of the SPEC benchmark suite protected with the Basic SelectiveCFI implementation on both hosts.

Figure 12.17 shows a comparison of the percentage of the functions of the programs in the SPEC benchmark

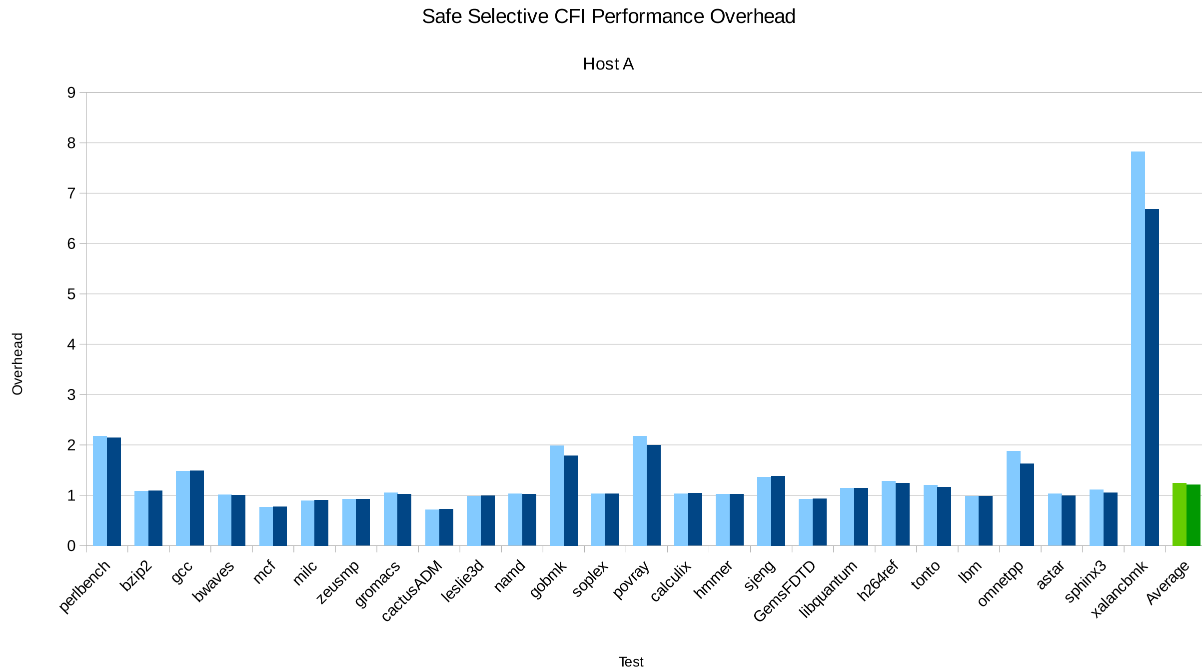

Figure 12.13: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host A. The results for the applications protected with Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green.

suite that the CFI analysis statically deems safe. The CFI analysis implemented on Host A is exactly the same as the analysis on Host B and the analysis operates on program binaries directly. Because Hosts A and B compile the programs of the SPEC benchmark suite with exactly the same compiler and libraries and the analysis is deterministic, it is reasonable to conjecture that the percentage of functions statically deemed safe will be almost identical on Hosts A and B. The percentage of the functions of the programs of the SPEC benchmark suite that the CFI analysis statically deems safe is identical on both hosts – 17.27%.

There is no correlation between the percentage of safe functions in an individual benchmark and that program's performance improvement when secured with Basic SelectiveCFI compared to Safe SelectiveCFI implementations ($r^2 = 0.16$). As described in Section 10.5.2, this result is not surprising. A relative performance improvement should correlate with the number of times that returns from safe functions occur at runtime and not simply with the number of returns from safe functions in a particular program. The results indicate that `omnetpp`, `povray` and `gobmk` execute at runtime the most returns from functions statically deemed safe.

As described in Section 12.2.2, the safety of transfers that target instructions that can be decorated with nonces is checked efficiently with only a few instructions. When a target cannot be decorated with a

Figure 12.14: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI without checking the safety of `ret` targets in functions statically deemed safe. These results are for Host B. The results for the applications protected with Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green.

nonce, the safety of a transfer to the target must be determined using code along a slow path. Therefore, the percentage of targets that cannot be decorated with nonces has a direct impact on the runtime performance of the program/library protected with Safe SelectiveCFI. Figures 12.18 show the percentages of time that targets in the programs of the SPEC benchmarks suite can be decorated with nonces. On average, targets can be decorated more than 97% of the time on both Hosts A and B. The ability to decorate targets is based on the layout of the program/library which is governed by the compiler. Because the compilers used on both hosts for these experiments is identical, it is expected that the placement ratios will be nearly identical.

CFI with Coloring

SelectiveCFI with coloring adds granularity to the security protections offered by the CFI instrumentation at the expense of decorating IBTs with more nonces (see 12.2.2). Because there are additional nonces to be placed, it is reasonable to expect that the ratio of those nonces successfully placed will decrease and the filesize overhead may increase; with a decrease in the ability to place decorations, it is possible that execution overhead will increase as the determination of the safety of indirect program control transfers more often takes the slow path.

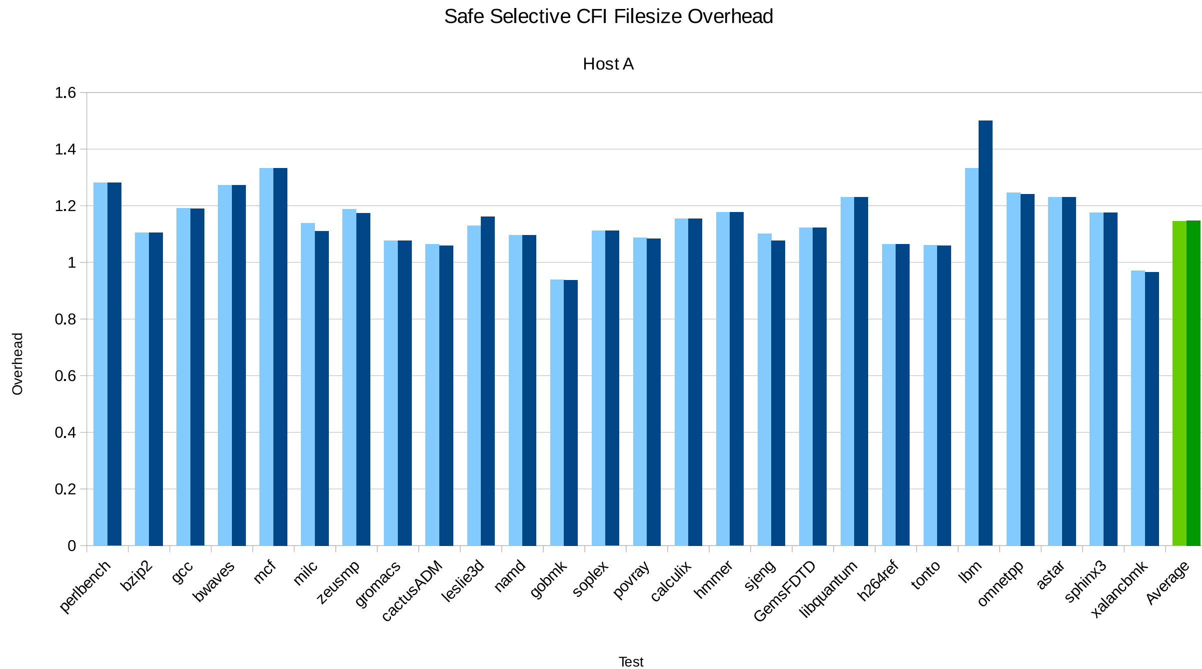

Figure 12.15: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host A. The results for the applications protected with the Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green.

Figures 12.19 and 12.20 show the performance impact of SelectiveCFI with Coloring on the statically rewritten versions of the programs of the SPEC benchmark suite. On average, the performance overhead on Host A is 1.39x which is a 10.57% impairment for the programs of the SPEC benchmark suite protected with the Basic SelectiveCFI. On average, the performance overhead on Host B is 1.37x which is an impairment of 9.70% compared with the performance overhead for the programs of the SPEC benchmark suite protected with the Basic SelectiveCFI.

Again, one of the priorities from the beginning for the design and architecture of the static binary rewriter described in this dissertation was for its algorithms to emit reassembled programs/libraries with minimum on-disk overhead (see Part I). Figures 12.21 and 12.22 show the impact of SelectiveCFI with Coloring on the on-disk size of the statically rewritten versions of the programs of the SPEC benchmark suite. The on-disk size overhead for Hosts A and B are 1.19x and 1.18, respectively, when compared to their native counterparts. That is 2.87% and 2.53% increase, respectively, in on-disk overhead when compared with the applications of the SPEC benchmark suite protected with the Basic SelectiveCFI implementation.

As described in Section 12.2.2, the safety of transfers that target instructions that can be decorated with nonces is checked efficiently with only a few instructions. When a target cannot be decorated with a

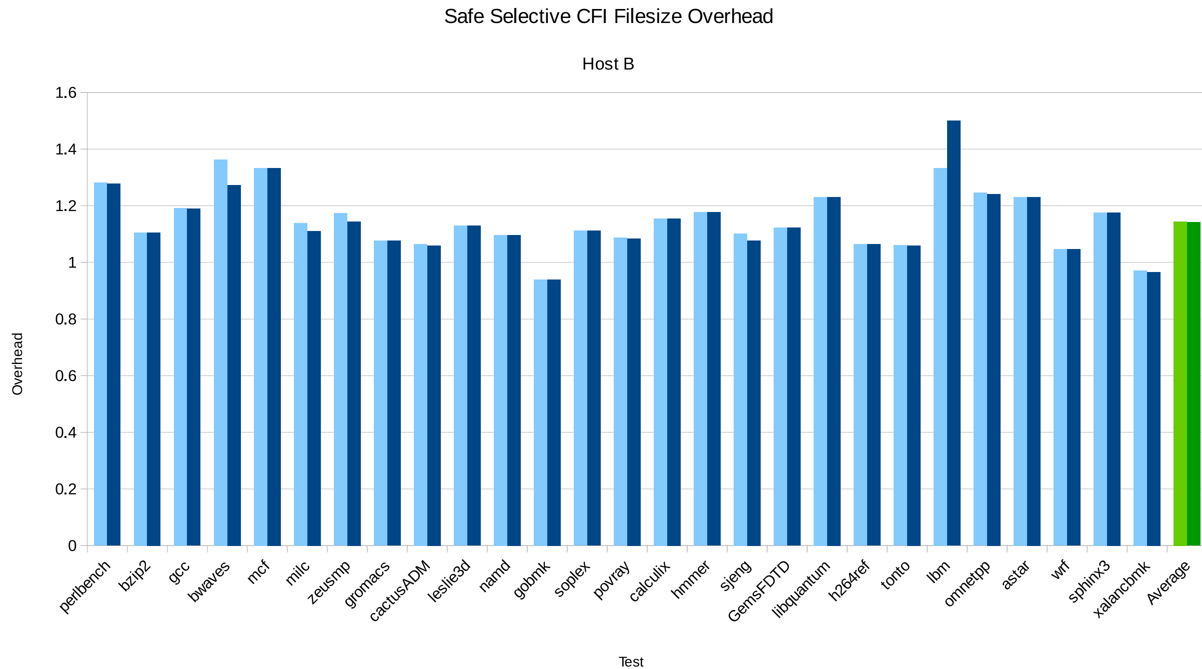

Figure 12.16: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using Safe SelectiveCFI. These results are for Host B. The results for the applications protected with the Safe SelectiveCFI are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green.

nonce, the safety of a transfer to the target must be determined using code along a slow path. Therefore, the percentage of targets that cannot be decorated with nonces has a direct impact on the runtime performance of the program/library protected with SelectiveCFI with Coloring. Figures 12.23 show the percentages of time that targets in the programs of the SPEC benchmarks suite can be decorated with nonces. On average, targets can be decorated 93.81% and 93.98% of the time on both Hosts A and B respectively. The ability to decorate targets is based on the layout of the program/library which is governed by the compiler. Because the compilers used on both hosts for these experiments is identical, it is expected that the placement ratios will be nearly identical except for the fact that SelectiveCFI with Coloring uses significantly more nonces than the Basic SelectiveCFI implementation (more than 23% and 4x more successfully and unsuccessfully placed nonces, respectively). The ratio of successful placements of nonces on the SelectiveCFI with Coloring implementation is 4.33% and 4.19% worse on Hosts A and B, respectively, when compared with the successful placement ratios of the Basic SelectiveCFI implementation.

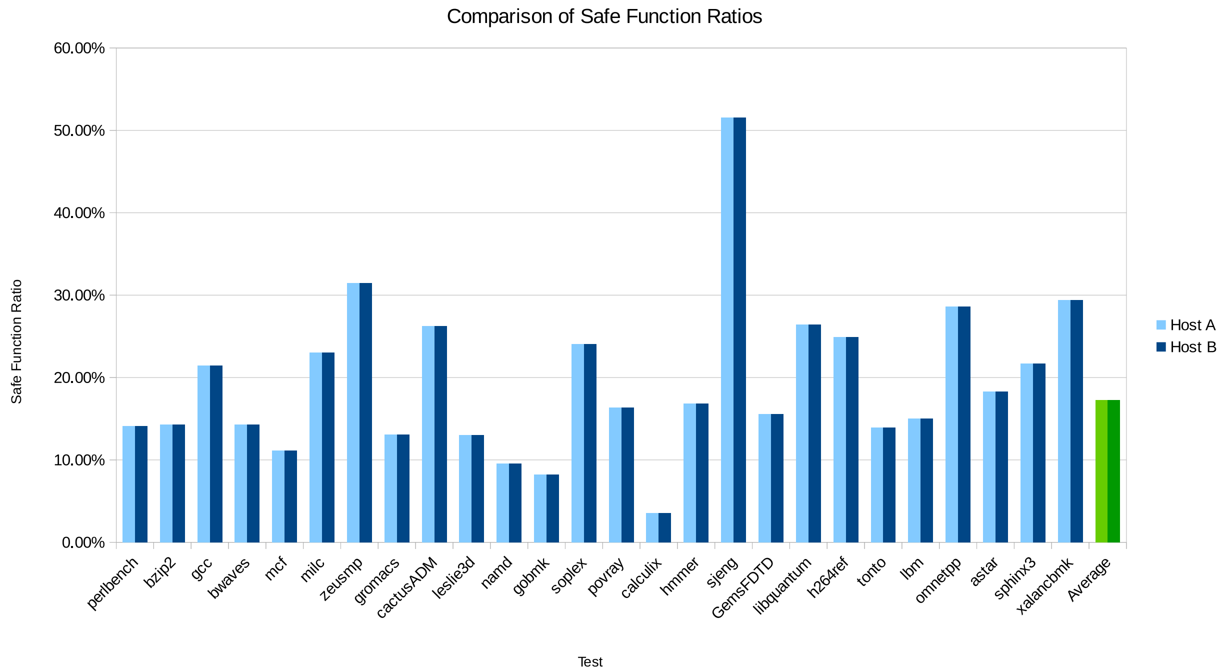

Figure 12.17: Comparison of the ratios of the functions of the applications of the SPEC benchmark suite statically deemed safe on Hosts A and B.

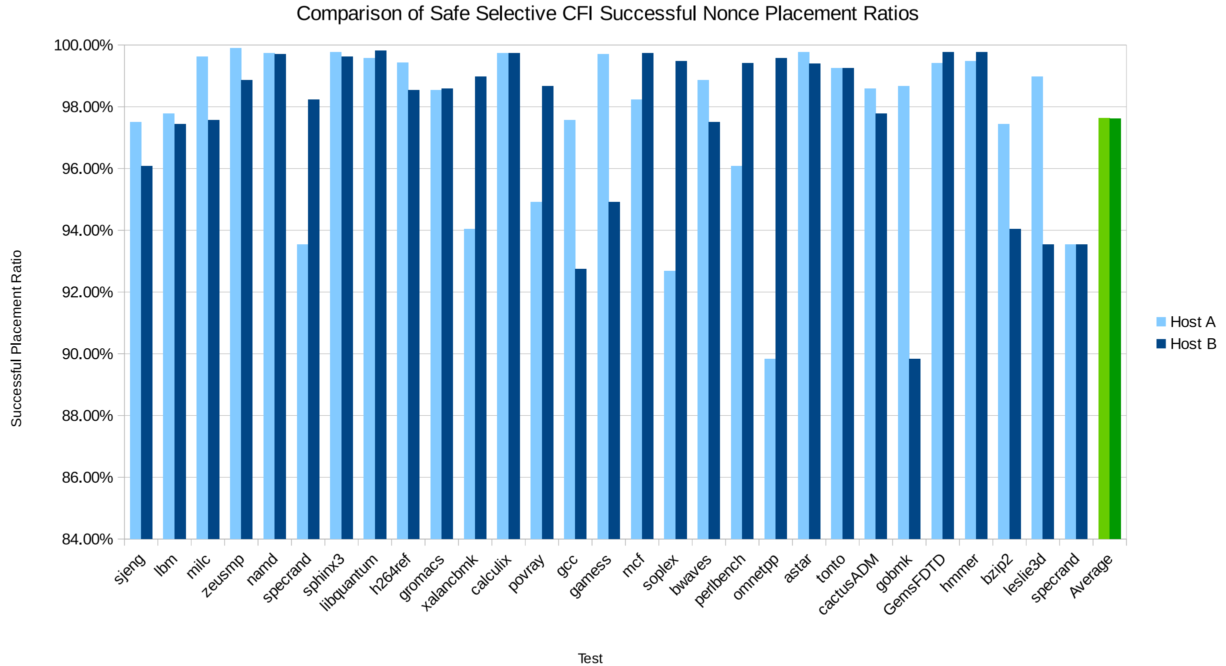

Figure 12.18: Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with Safe SelectiveCFI on Hosts A and B.

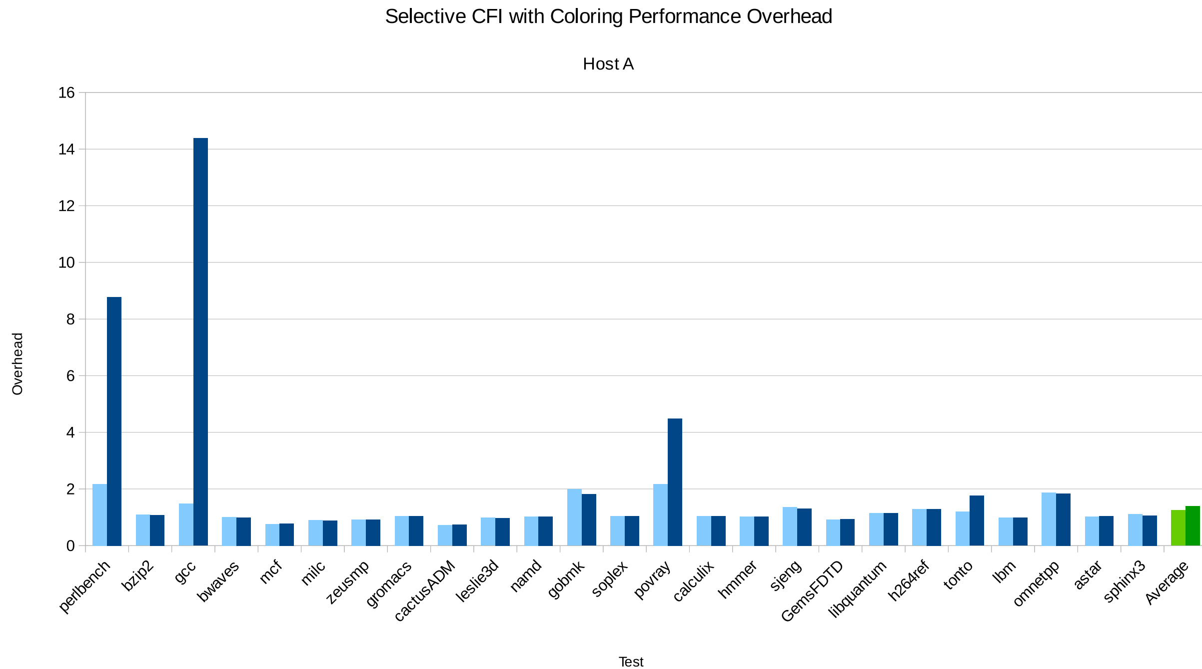

Figure 12.19: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green.

12.4.3 CGC

The more than 140 CBs in the CGC dataset provide another way to assess the impact of SelectiveCFI on the overall performance of protected programs.[4] As discussed earlier (see Section 5.6), the CGC environment and the tools contained therein provide an ideal laboratory environment for test and measurement. Section 5.6 provides background information on the CGC environment, its tools and the terms used in this section.

CFI Only

The first set of experiments measures the performance impact of the basic implementation of SelectiveCFI on the CBs of the CGC (i.e., no coloring). Figure 12.24 shows the availability scores of the CBs in the test dataset. On average, the availability score is 77.04%. That is a 8.29% decrease in availability when compared to the CBs in the test dataset when statically rewritten using the default algorithms of the Reassembly phase.

Figure 12.25 shows the execution overhead for the CBs in the test dataset. On average, the execution overhead is 4.36%. That is almost a 42% increase in execution overhead when compared to the CBs in the test dataset when statically rewritten using the default algorithms of the Reassembly phase (2.53%). This

[4]For this experiment, 8 of the Challenge Binaries were omitted because the transformed versions failed to execute in one or more of the configurations under test.

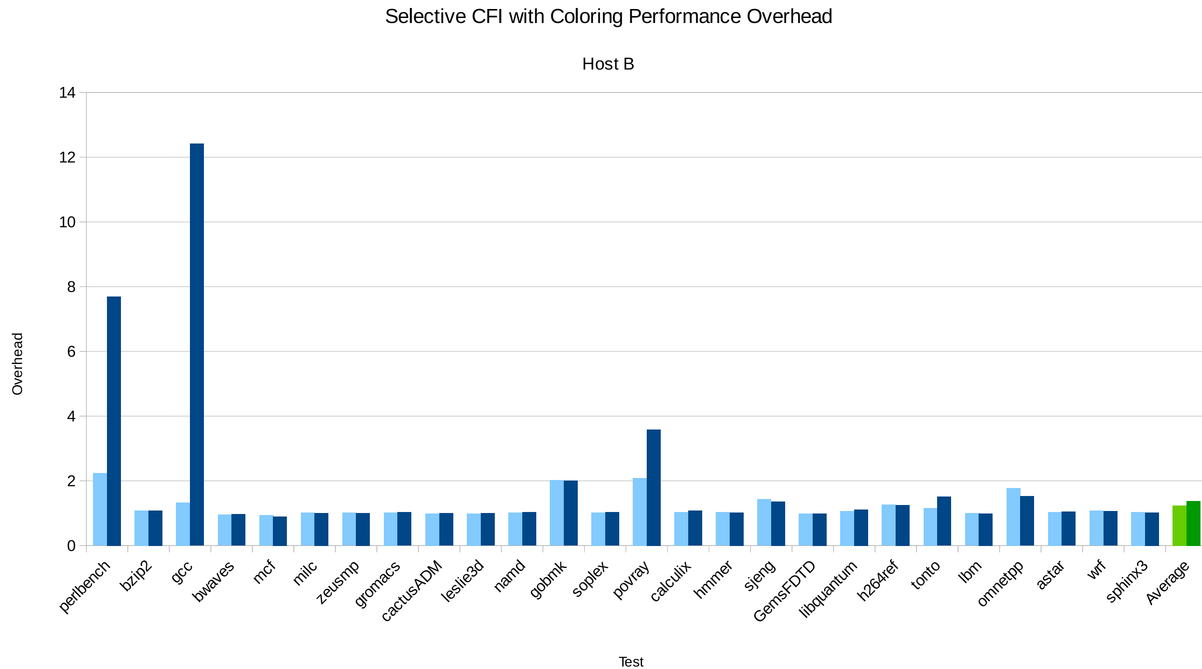

Figure 12.20: Performance overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host B. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are shown on the left, in light blue and light green.

increase in execution overhead is unexpected when compared to the performance improvements seen on Hosts A and B when the programs of the SPEC benchmark suite were protected with Basic SelectiveCFI as described in Section 12.4.2. The earlier results were partially explained by the relatively large amount of time spent in computational kernels in the applications of the SPEC benchmark suite. The CBs are not computationally bound in the same way and execute more indirect program control transfers during execution which increases the overhead as a result of the SelectiveCFI transformation.

Figure 12.26 shows the runtime memory overhead for the CBs in the test dataset. On average, the runtime memory overhead is 12.02%. That is almost a 20% increase in runtime memory overhead when compared to the CBs in the test dataset when statically rewritten using the default algorithms of the Reassembly phase (9.70%).

Figure 12.27 shows the on-disk overhead for the CBs in the test dataset. On average, the on-disk overhead is 9.08%. That is almost an 18% increase in on-disk overhead when compared to the CBs in the test dataset when statically rewritten using the default algorithms of the Reassembly phase (7.47%).

In the same way that the ability to decorate a target with a nonce plays an important role in the performance of the programs of the SPEC benchmark suite protected by the SelectiveCFI transformation,

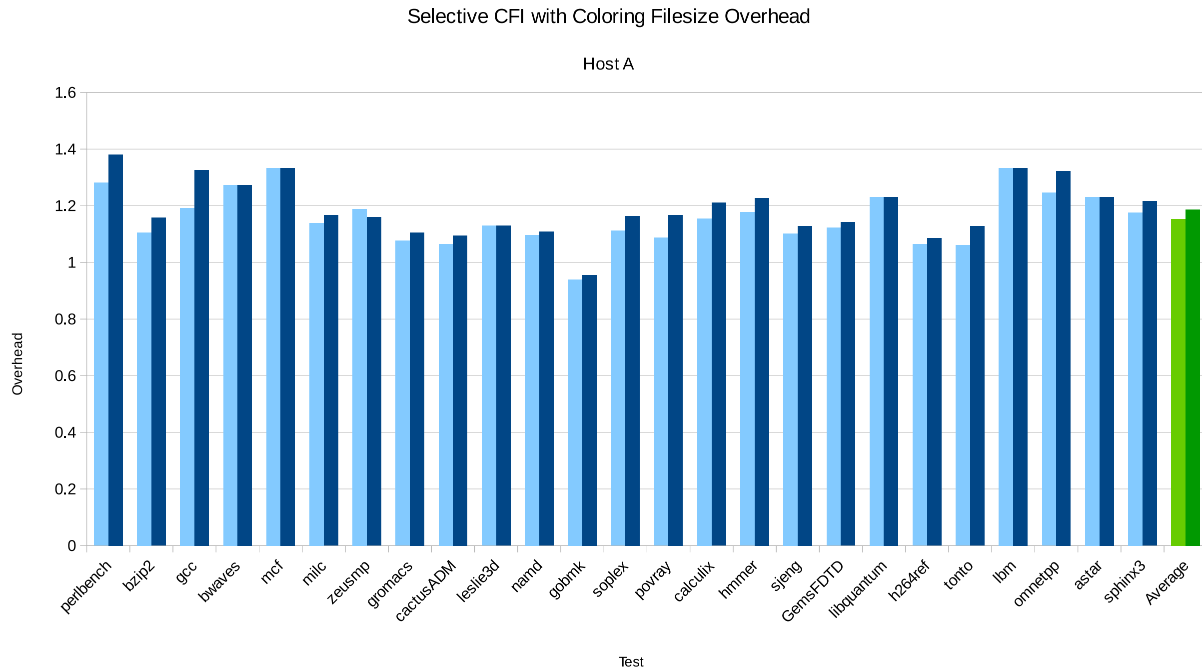

Figure 12.21: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with the SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green.

the ability to decorate targets in the CBs is a factor in the execution overhead and, therefore, availability scores of the RCBs. Figure 12.28 shows the rate at which targets can be decorated in the rewritten versions of the CBs in the test dataset. On average, the decoration percentage is 97.26%. This decoration ration is very similar to the decoration ratio for the applications of the SPEC benchmark suite (97.87% and 97.91% on Hosts A and B, respectively).

CFI Without Protecting Safe Indirect Transfers

As described in Section 12.2.1, the possibility of gaining the security of CFI without having to check the safety of every indirect program control transfer opens up the chance for reducing the performance overhead associated with the security gained with the technique.

The evaluation of the performance of Safe SelectiveCFI using the CGC dataset used the basic method for protecting the return indirect control transfer operations (see Section 12.2.2). The choice of the basic method means that the hardware's return address predictor will play a role in the performance. A comparison between the results presented in this section will be comparable to the results in Section 12.4.2 and will demonstrate the relative merits of the two methods of instrumentation.

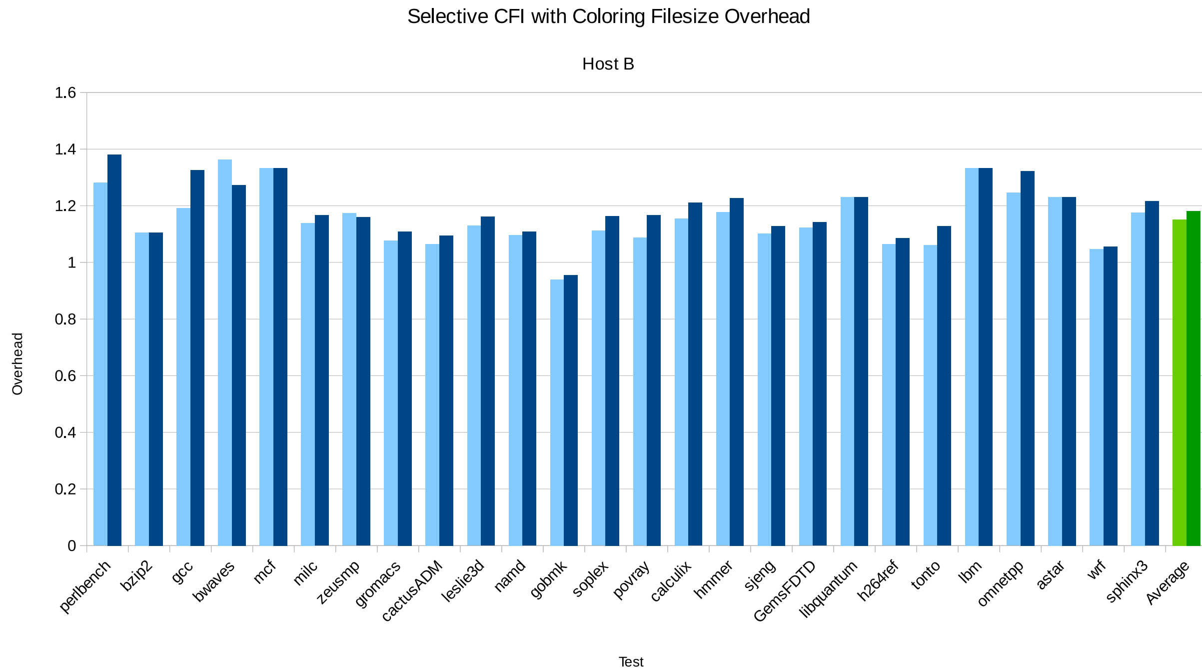

Figure 12.22: Filesize overhead of the applications in the SPEC2006 benchmark suite when protected using SelectiveCFI with Coloring. These results are for Host A. The results for the applications protected with SelectiveCFI with Coloring are shown on the right, in dark blue and dark green. The results for the applications protected with Basic SelectiveCFI are on the left, in light blue and light green.

Figure 12.29 shows the availability scores of the CBs in the test dataset when secured with Safe SelectiveCFI. On average, the availability score is 75.81%, a decrease in availability when compared with the Basic CFI implementation/instrumentation. This result is inconsistent with the hypothesis that instrumenting fewer indirect program control transfers would lead to a performance improvement. Further experiments show that this decrease in availability can be attributed to the execution overhead, a fact which is explained further below.

Figure 12.30 shows the runtime memory overhead for the CBs in the test dataset when protected with Safe SelectiveCFI. On average, the runtime memory overhead is 11.23%. That is over a 7% decrease in runtime memory overhead compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation (12.02%).

Figure 12.31 shows the on-disk overhead for the CBs in the test dataset when protected with Safe SelectiveCFI. On average, the on-disk overhead is 8.83%. That is over a 2.75% decrease in on-disk overhead compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation 9.08%).

Figure 12.32 shows the execution overhead for the CBs in the test dataset when protected with Safe SelectiveCFI. On average, the execution overhead is 7.43%. That is a 41.32% increase in execution overhead

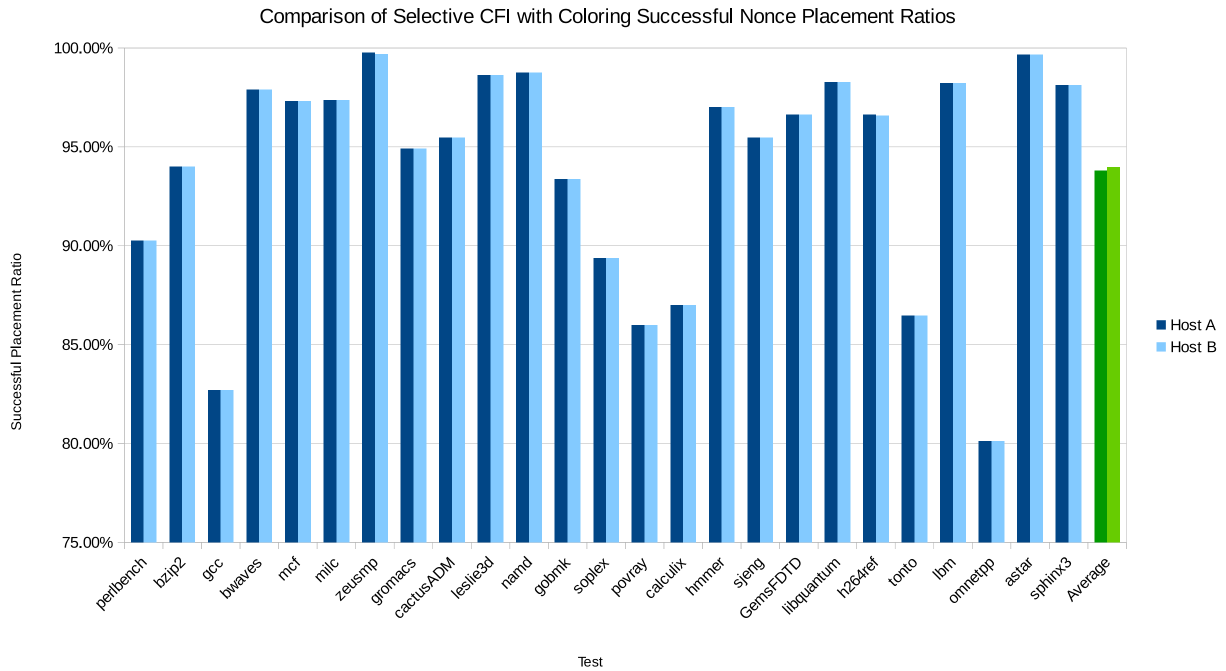

Figure 12.23: Comparison of the ratios of the successfully placed nonces for the applications of the SPEC benchmark suite protected with Safe SelectiveCFI on Hosts A and B.

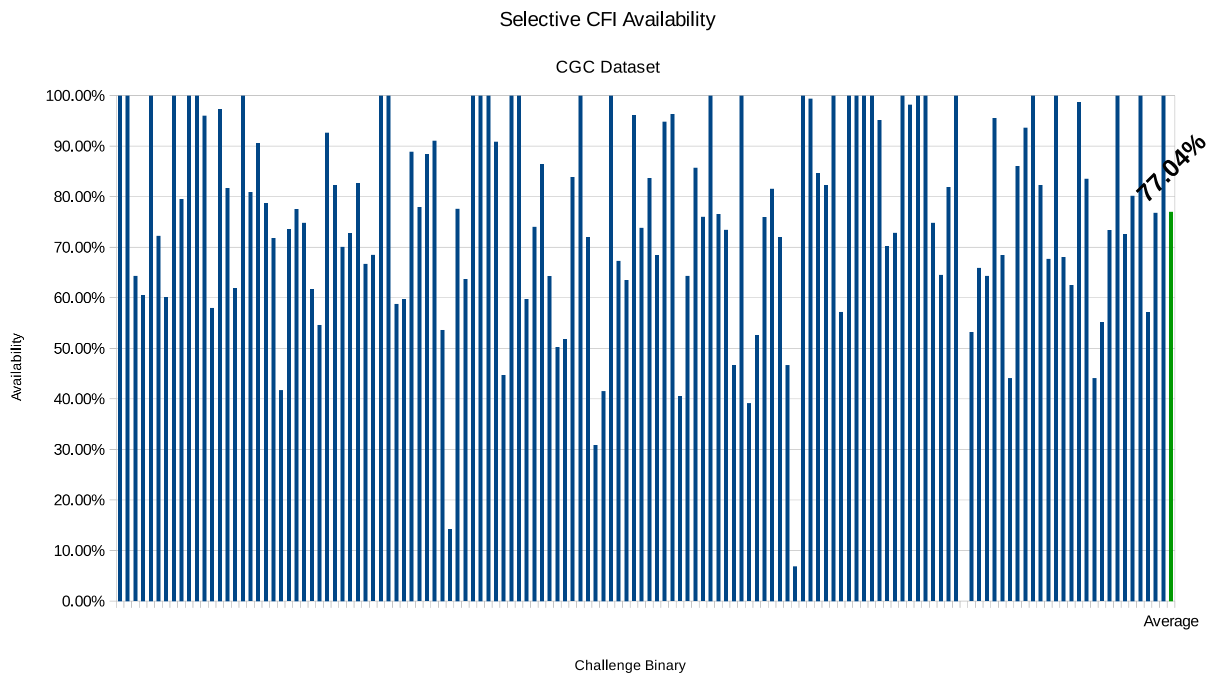

Figure 12.24: Availability of the Challenge Binaries protected with the Basic Selective CFI implementation.

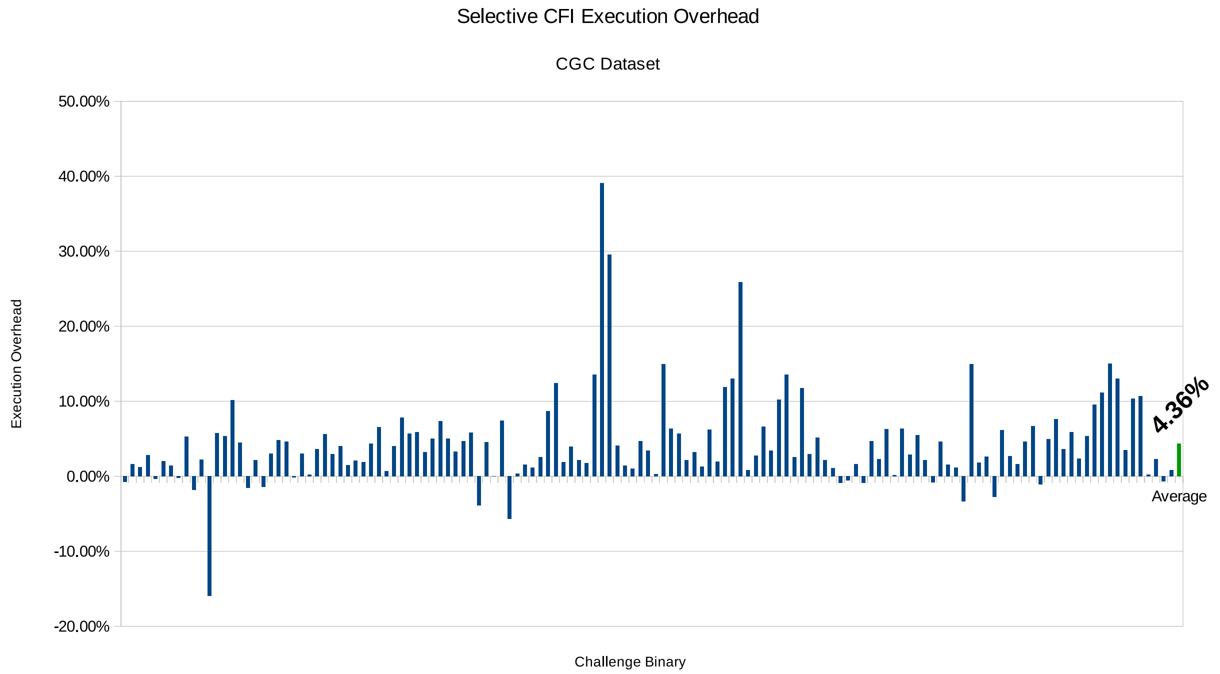

Figure 12.25: Execution overhead of the Challenge Binaries protected with the Basic Selective CFI implementation.

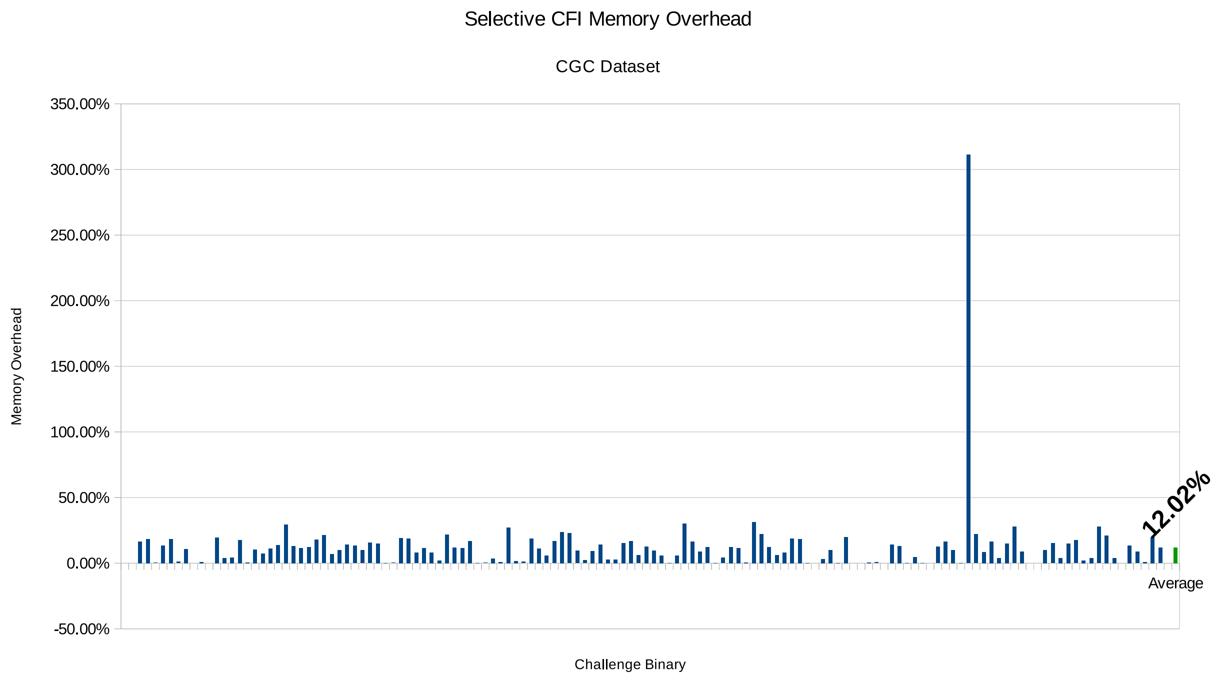

Figure 12.26: Memory overhead of the Challenge Binaries protected with the Basic Selective CFI implementation.

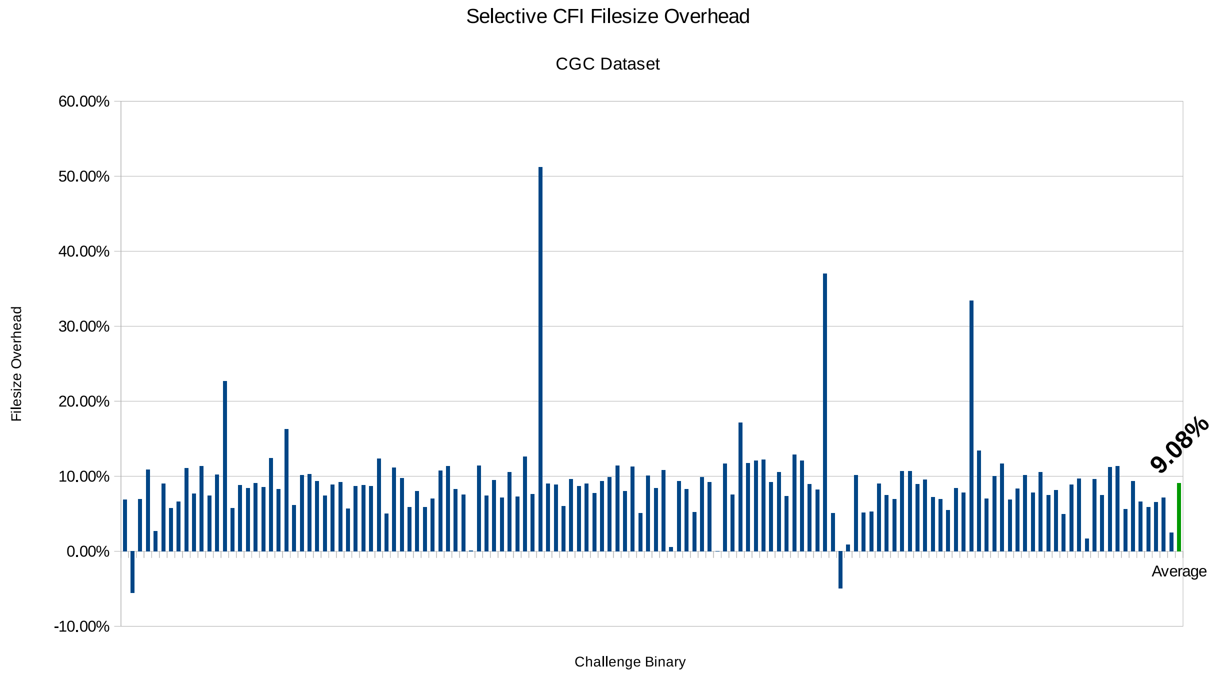

Figure 12.27: Filesize overhead of the Challenge Binaries protected with the Basic Selective CFI implementation.

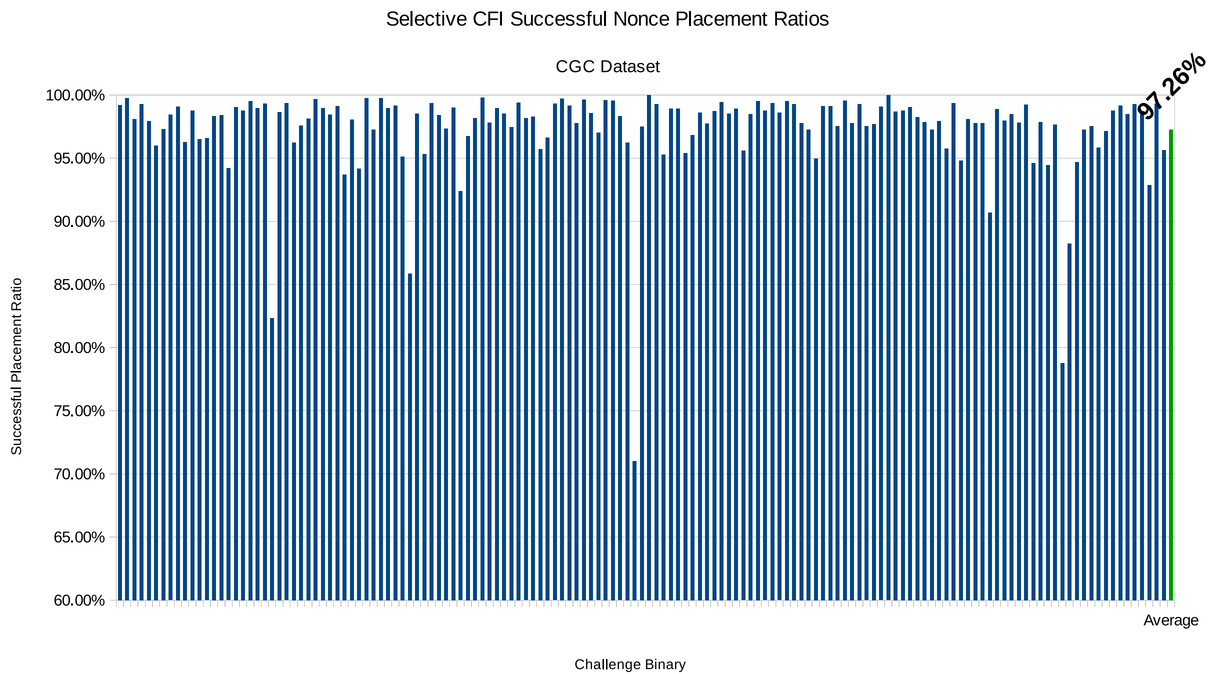

Figure 12.28: Percentage of successful placement of nonces in Challenge Binaries protected with the Basic SelectiveCFI implementation.

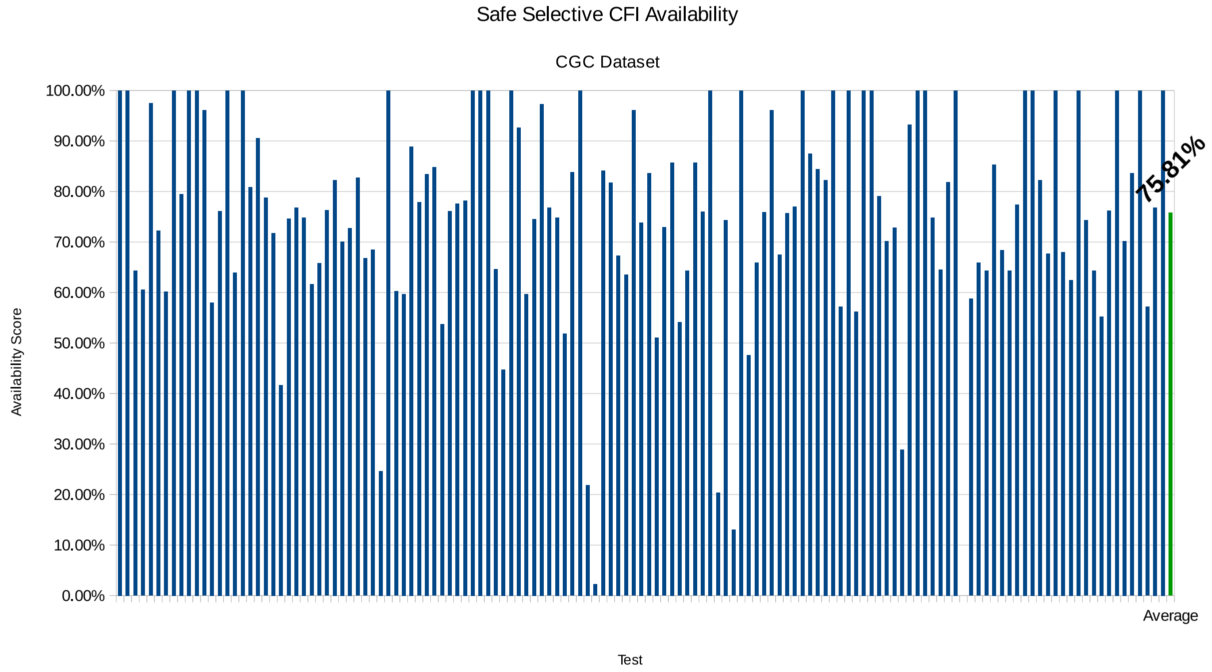

Figure 12.29: Availability of the Challenge Binaries protected with the Safe Selective CFI implementation.

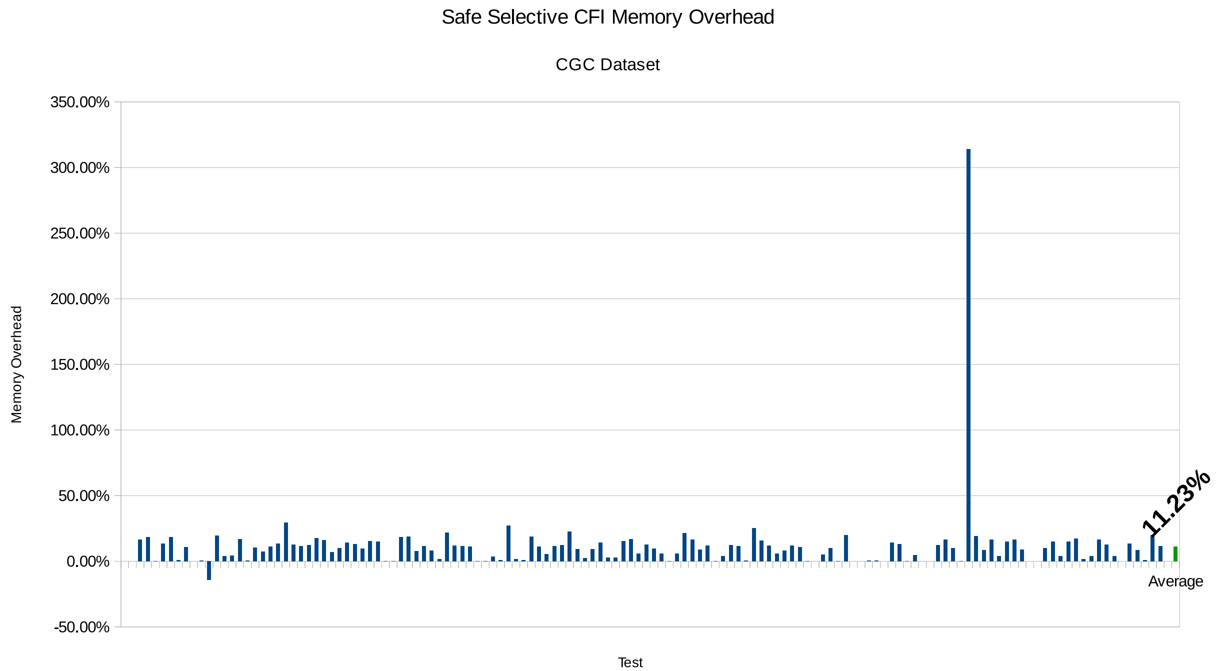

Figure 12.30: Memory overhead of the Challenge Binaries protected with the Safe Selective CFI implementation.

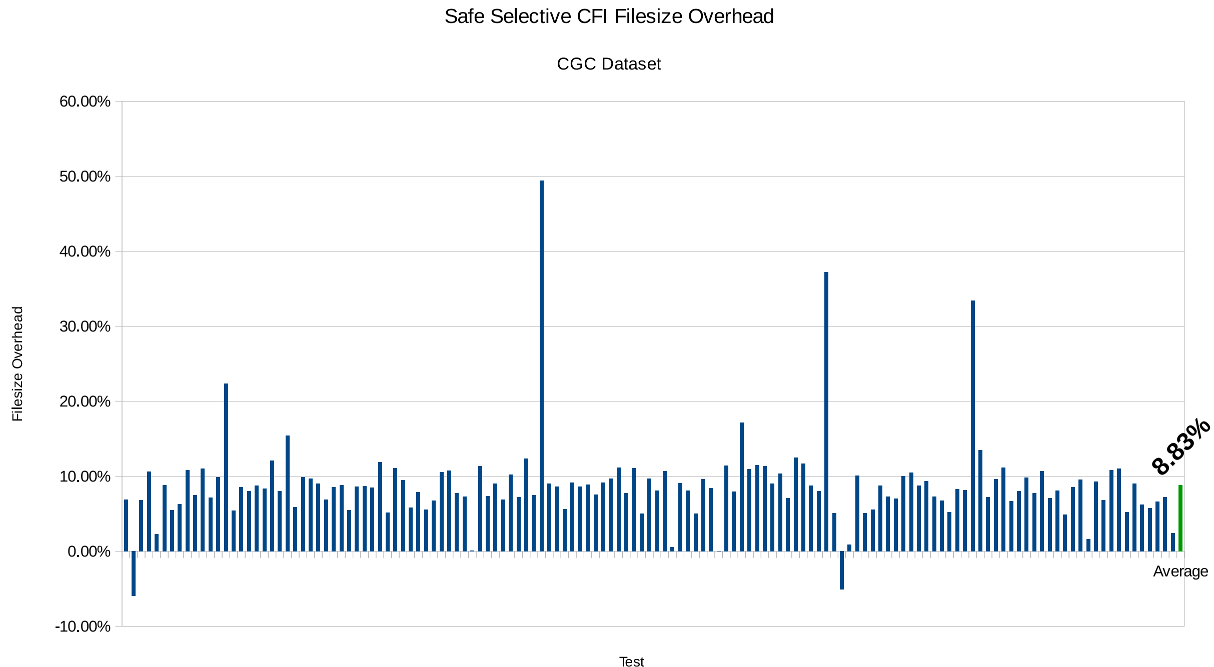

Figure 12.31: Filesize overhead of the Challenge Binaries protected with the Safe Selective CFI implementation.

when compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation (4.36%).

The results for the on-disk and runtime memory overhead demonstrate that instrumenting fewer indirect program control transfers does have a significant benefit on at least some metrics of performance. However, a superficial interpretation of the data indicates that execution overhead is not one of those metrics that show an improvement.

The key to understanding the additional execution overhead is, again, the return address stack. The choice of the basic method of instrumenting for safety means that there is a mismatch between `ret`s and (nonexistent) `call`s which introduces a penalty compared to the alternate method. The performance improvement gained by not protecting 27.92% (Figure 12.33), on average, of functions in the CBs of the test dataset is, therefore, greatly outweighed by the penalty from not taking full advantage of the hardware's optimizations.

Comparing the performance impairment shown in these results compared to the performance improvements seen in the results from the SPEC benchmark suite are a stark example of the impact of the return address stack on performance (Section 12.4.2). The data show that alternate method is better for performance despite the fact that more instructions are executed.

Again, in the same way that the ability to decorate a target with a nonce plays an important role in the performance of the programs of the SPEC benchmark suite protected by the SelectiveCFI transformation, the

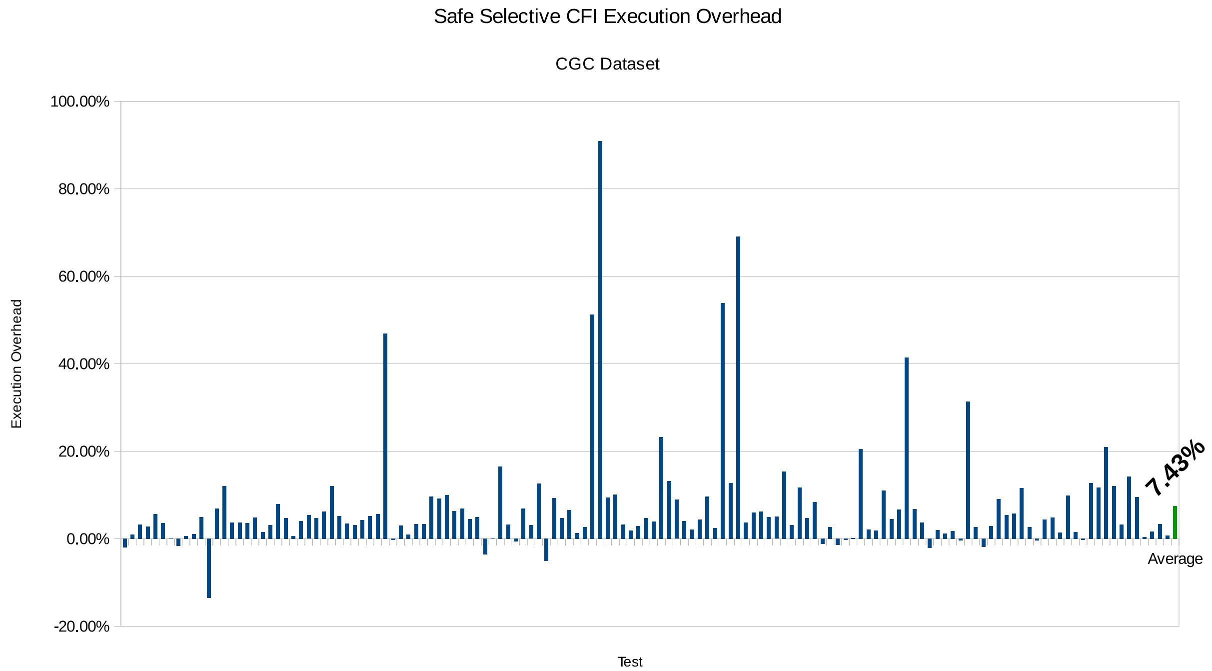

Figure 12.32: Execution overhead of the Challenge Binaries protected with the Safe Selective CFI implementation.

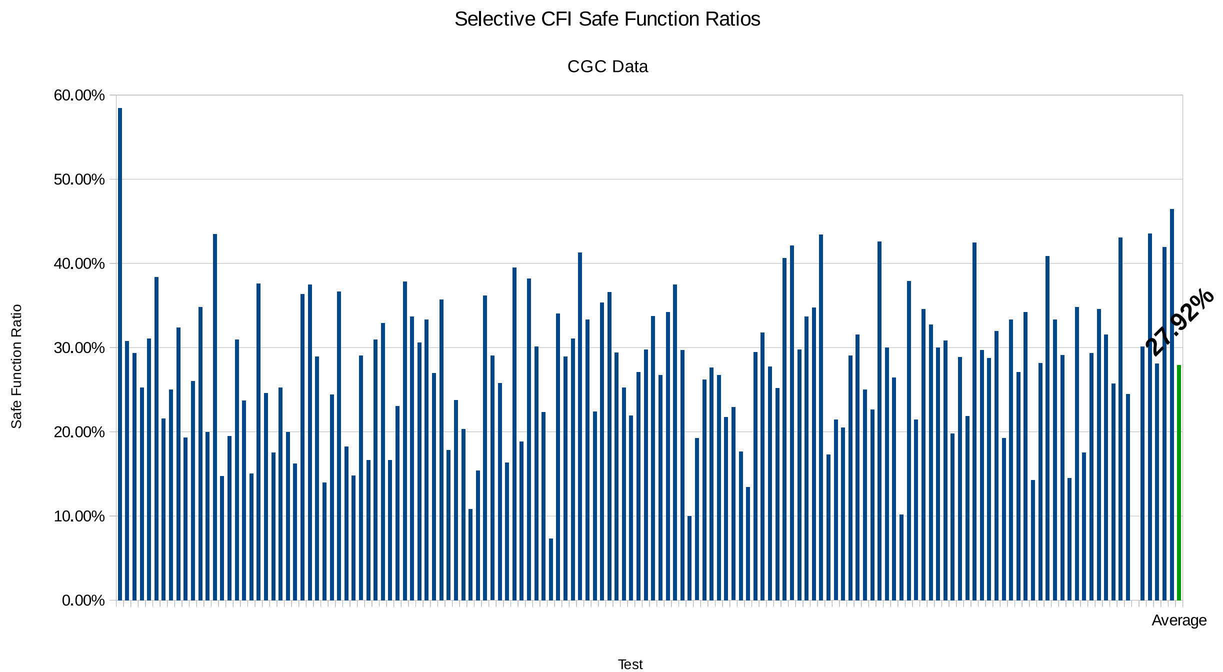

Figure 12.33: Percentage of functions in the Challenge Binaries statically deemed safe by the SelectiveCFI analysis.

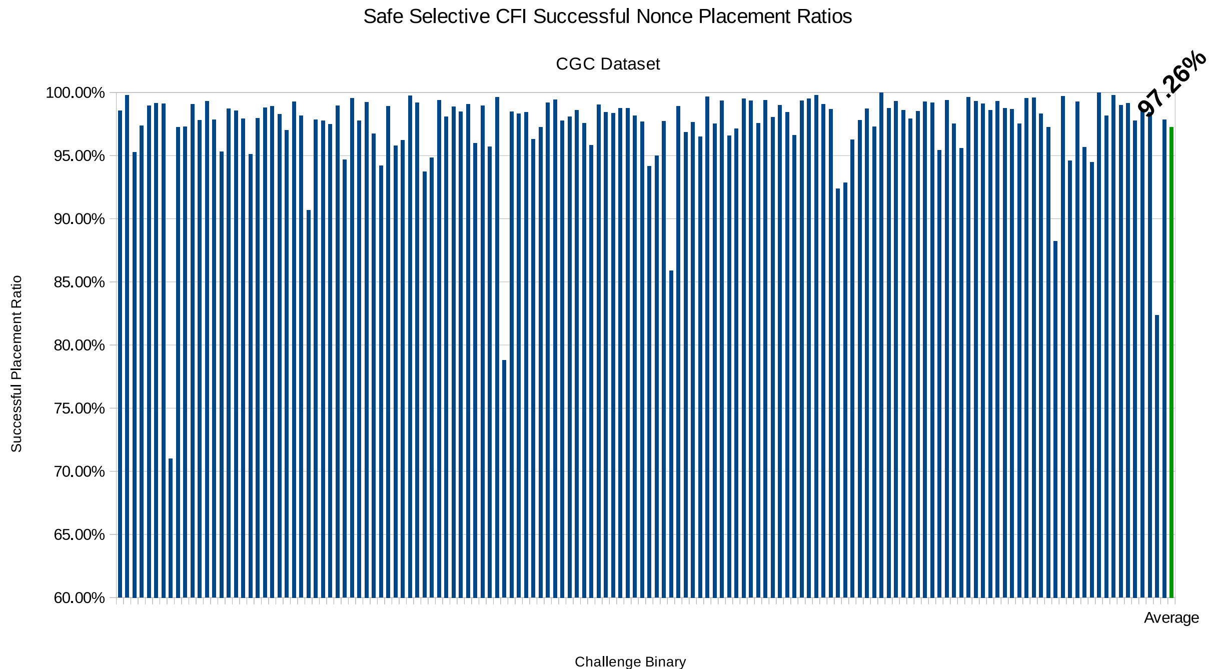

Figure 12.34: Percentage of successful placement of nonces in Challenge Binaries protected with the Safe SelectiveCFI implementation.

ability to decorate targets in the CBs is a factor in the execution overhead and, therefore, availability scores of the RCBs. There is no reason to believe that not protecting safe indirect transfers will change the decoration ratios. The results shown in Figure 12.34 confirm this hypothesis. On average, the decoration percentage is 97.26% and is exactly the same as the decoration ratio for the Basic SelectiveCFI implementation.

CFI with Coloring

SelectiveCFI with coloring adds granularity to the security protections offered by the CFI instrumentation at the expense of additional decorations around targets (see 12.2.2). Because there are additional decorations to be placed, it is reasonable to expect that the decoration ratio will decrease and the filesize overhead may increase. If there is a decrease in the ability to place decorations, it is possible that execution overhead will increase as the determination of the safety of indirect program control transfers more often takes the slow path.

Figure 12.35 shows the availability scores of the CBs in the test dataset. On average, the availability score is 75.05%, a decrease in availability when compared with Basic SelectiveCFI implementation/instrumentation (77.04%). This result is consistent with the hypothesis that decorating indirect targets with sets of nonce values rather than a single nonce value would lead to an overall performance impairment. Further experiments

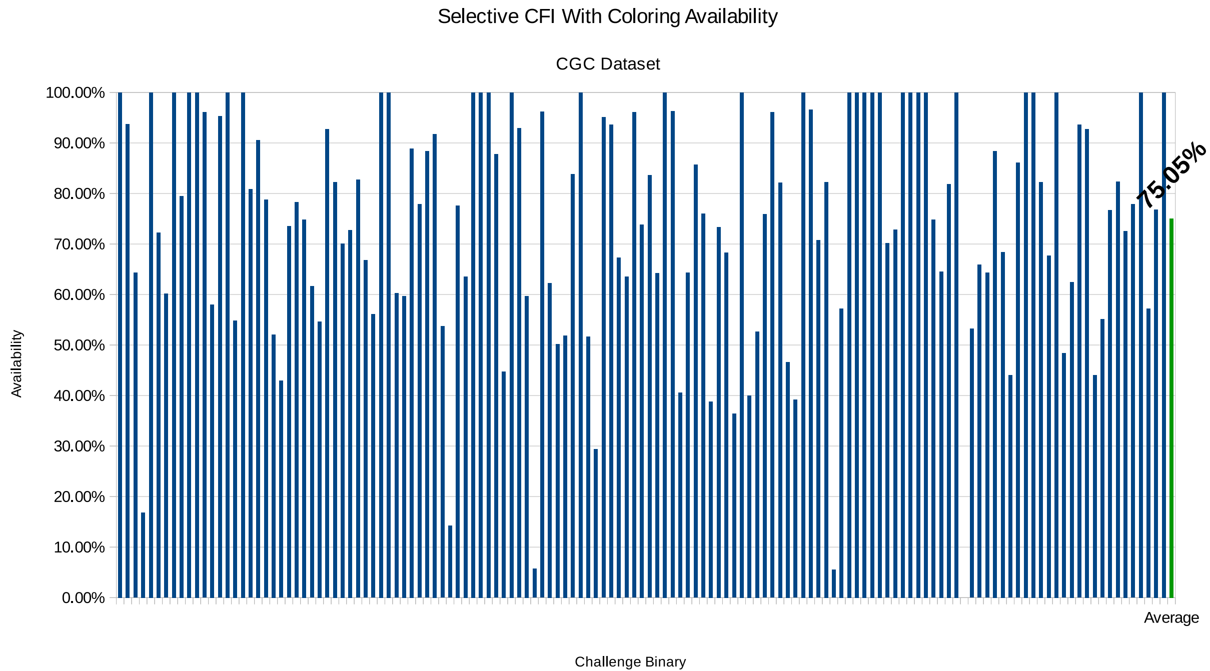

Figure 12.35: Availability of the Challenge Binaries protected with the Selective CFI with Coloring implementation.

show just how much each individual metric of performance is impacted by the increased nonces needed for the additional security granularity.

Figure 12.36 shows the runtime memory overhead for the CBs in the test dataset. On average, the runtime memory overhead is 12.61%. That is a 4.70% increase in runtime memory overhead compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation (12.02%).

Figure 12.37 shows the on-disk overhead for the CBs in the test dataset. On average, the on-disk overhead is 9.36%. That is over a 2% increasing on-disk overhead compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation (9.08%).

Figure 12.38 shows the execution overhead for the CBs in the test dataset. On average, the execution overhead is 3.66%. That is almost a 3% increase in execution overhead when compared to the CBs in the test dataset when protected with the Basic SelectiveCFI instrumentation (3.54%).

Figure 12.39 shows the percentages of nonces that were successfully placed at their targets. The data support the hypothesis that because SelectiveCFI with Color employs additional decorations, the successful placement ratio will decline. On average, the placement ratio for all the nonces needed for SelectiveCFI with Coloring is 96.28%. For the Basic SelectiveCFI, the ratio is 97.26%. In other words, there is a 1.02% decrease in the likelihood of successfully placing nonces in the SelectiveCFI with Coloring than with the

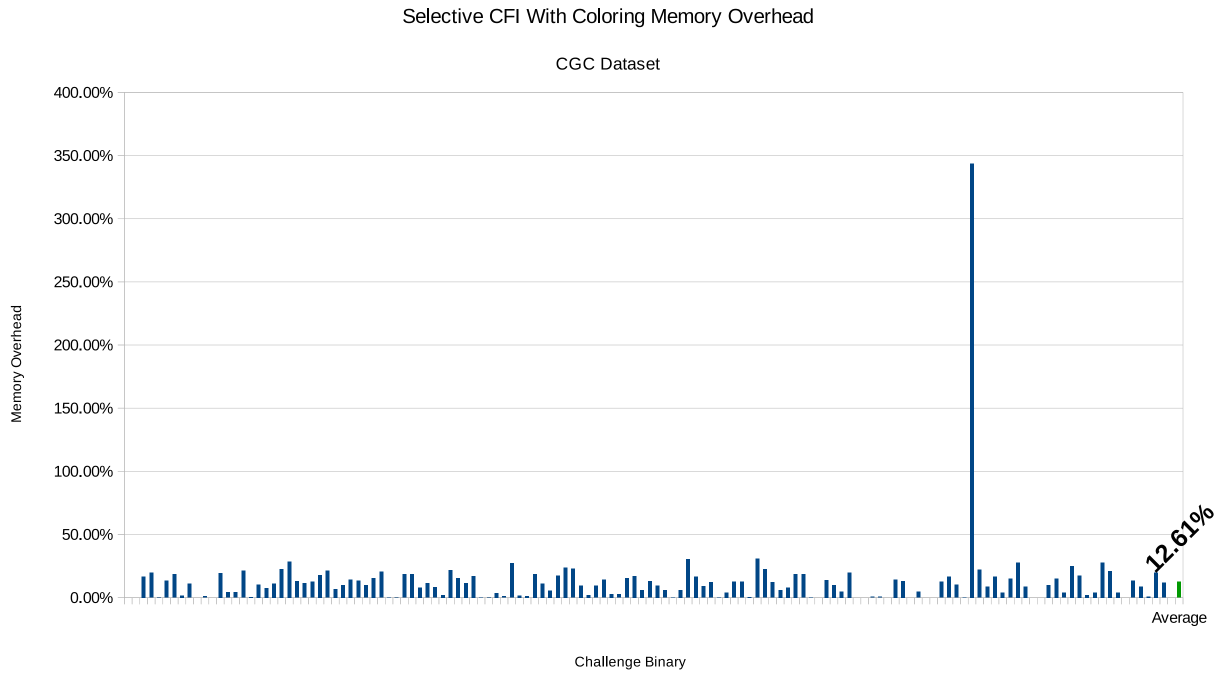

Figure 12.36: Memory overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation.

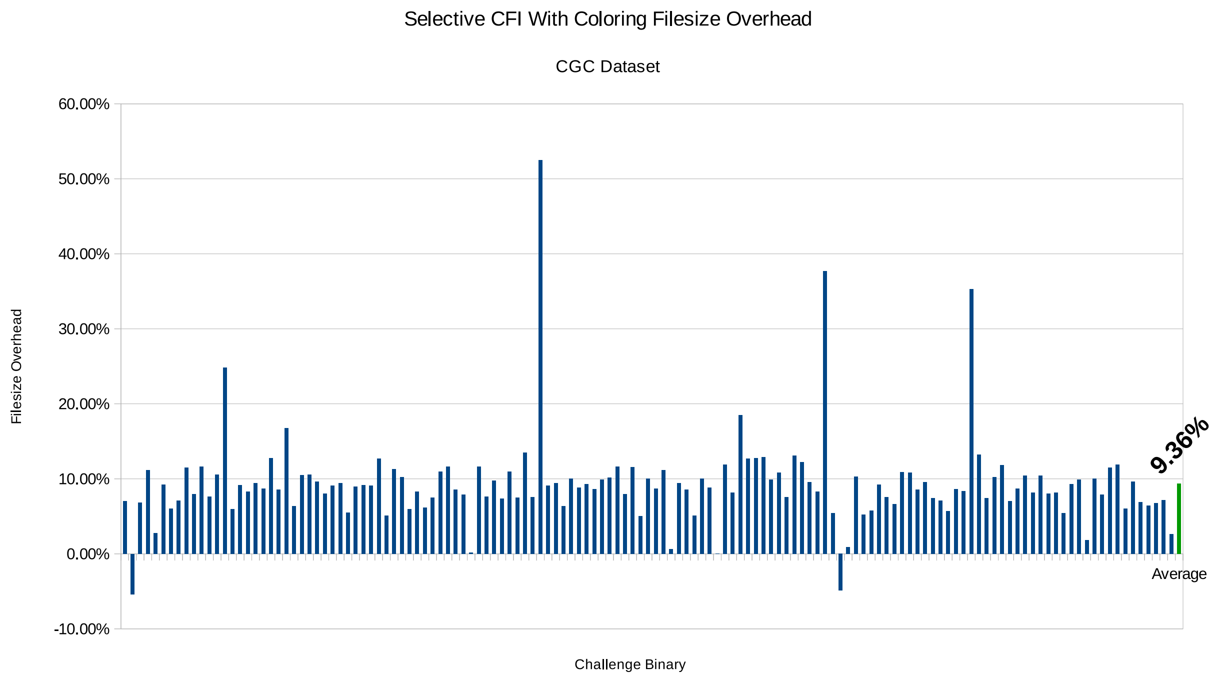

Figure 12.37: Filesize overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation.

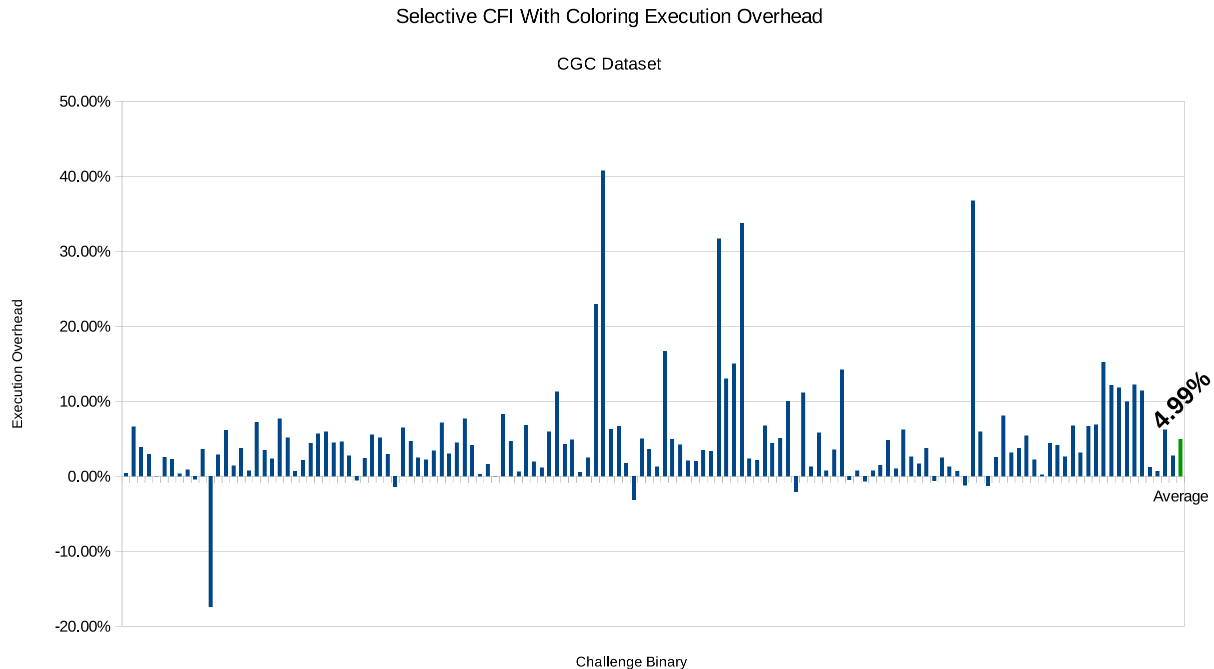

Figure 12.38: Execution overhead of the Challenge Binaries protected with the Selective CFI with Coloring implementation.

Basic SelectiveCFI implementation.

The placement ratio does not tell the whole story. The absolute number of nonces used by Selective CFI with Coloring also plays a critical role in determining performance. There are 1.12x as many placed nonces used by SelectiveCFI with Coloring as there are with the Basic SelectiveCFI implementation. There are 2.07x as many nonces that cannot be placed in SelectiveCFI with Coloring as there are with Basic SelectiveCFI.

12.5 Conclusion

CFI is one of several generic security protections that, although not 100% effective in every situation, greatly improve the baseline security posture of a majority of software and can be deployed proactively to protect against the flaws inevitably latent in all software.

SelectiveCFI is implemented as a User-specified transformation deployed using the prototype implementation of the algorithms and architecture of the static binary rewriter described in this dissertation (Zipr) to apply CFI to SOUP. The results from experimentation with Basic SelectiveCFI and SelectiveCFI with Coloring show that it incurs minimal overhead [216, 11] and adds security to vulnerable programs. Using SelectiveCFI with Coloring adds stronger protection to a program at the expense of increased runtime performance and on-disk size overheads.

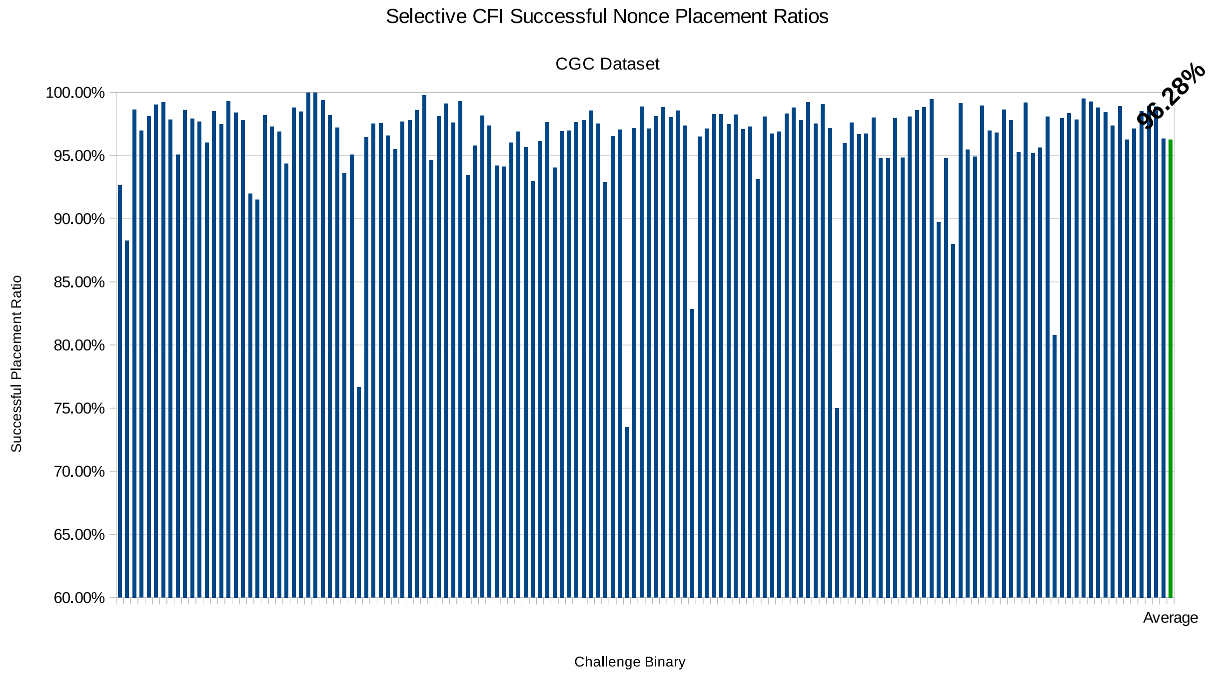

Figure 12.39: Percentage of successful placement of nonces in Challenge Binaries protected with the SelectiveCFI with Coloring implementation.

Finally, SelectiveCFI is another example of the power of the static binary rewriter architecture and design described in this dissertation to transform SOUP to improve its security and reliability.

Chapter 13

Conclusion

The notion that software runs the modern world is generally accepted and humans interact with software to conduct most of the tasks in their day-to-day lives. When people rely on software without realizing it, the stakes are high. Software controls the power grid, dams, airplanes, cars, surgery, the economy, national defense and so on.

"Computing systems in which the consequences of failure are very serious are termed safety-critical" [242] and their disruption, whether caused by the malicious activity of an attacker, the nature of the complexity of the system, operator error, etc., can cost humans their lives or livelihoods. The public would greatly benefit from a *general* tool that can be used to improve the security, safety and reliability of these systems. Chapter 2 expanded on society's need for a retargetable, static binary rewriter that gives system designers the power to apply *post hoc* transformations that improve the security and reliability of SOUP without excessive size and performance overhead.

The literature surveyed in this dissertation demonstrates the reasons why such a tool is difficult to find, much less build (Chapter 6). When work on the design, algorithms and architecture of the static binary rewriter described in this dissertation began in late 2014, it was an open question whether a static binary rewriter could operate on SOUP and reliably generate rewritten programs/libraries. Nor was it clear that a static binary rewriter could rewrite SOUP without having to preserve a copy of the original code alongside the modified version in the statically rewritten output (thus creating an artifact with significant on-disk and runtime memory overhead) or that a static binary rewriter for SOUP could build output programs/libraries that execute with a reasonable amount of runtime overhead.

The design, architecture and algorithms presented, implemented and evaluated in this work definitively answer those questions. The results from this dissertation show that building a static binary rewriter for

SOUP that can be used to improve the security, safety and reliability of critical software systems is difficult but not impossible. The architecture and algorithms described herein can be combined to build a static binary rewriter that meets a strict set of requirements: it can operate on software that runs on several different OSes (e.g., Linux, UNIX, Windows, etc.) and hardware platforms (e.g., servers, mainframes, desktops, embedded devices, etc.); it can operate on software whose source code is unavailable or whose provenance is unknown; and it can improve software whose operating context strictly limits overhead (memory, time, power).

When combined, the algorithms and architecture described in this dissertation form the basis of a static binary rewriter that gives system designers the power to apply *post hoc* transformations that improve the security and reliability of SOUP without excessive size and performance overhead. These capabilities, in turn, give developers the tools they need to improve the software that drives daily life, irrespective of its provenance or operating context.

Chapter 3 formalized a schema for assessing the correctness of programs/libraries statically rewritten using the architecture and algorithms described and evaluated in this dissertation. The chapter also presented the limitations of such a conception. The definition of correctness parallels the definition of iO presented in Barak et al.'s. An indistinguishability obfuscator prevents an adversary who has unlimited access to program Q which is either $A' = iO(A)$ or $B' = iO(B)$ where A and B are programs that compute the same function f from determining whether Q is A' or B'. Using the framework from Chapter 3 for assessing the correctness of the algorithms and architecture presented herein is beyond the scope of this dissertation with the exception of an explicit disclaimer that matching the original program/library's input and output on a set of test inputs/outputs with the statically rewritten program/library's inputs and outputs is sufficient to prove the correctness of a statically rewritten program/library.

Nevertheless, the formal definition of correctness presented in Chapter 3 advances the entire field of research into building static binary rewriters by acting as a roadmap to the construction of the set of algorithms comprising the ideal static binary rewriter. The closer any particular implementation comes to generating formally correct artifacts, the more likely system architects will be to deploy such a tool. Furthermore, a method for comparing between an implementation and the theoretical can bridge the gap with other areas of computer science that can be usefully incorporated into the research on static binary rewriters – e.g., deploying proof-carrying code in the statically rewritten program/library for self-verification [155] or invariant preservation between original and statically rewritten program/library [71].

Part II discussed the fundamental architecture and algorithms underlying the static binary rewriter. The basis of the static binary rewriter is a pipeline of phases: IR Construction, Transformation and Reassembly. IR Construction begins with disassembly which involves the use of a combination of linear and recursive disassemblers to disambiguate code from data and convert the binary input program/library to be rewritten

into a sequence of instructions. The IR Construction phase concludes by converting that sequence of instructions into a CFG that represents the semantics of the execution of the input program/library. The Transformation phase prepares the IR for reassembly and offers the user the opportunity to arbitrarily modify the IR through a user-friendly API. Finally, the Reassembly phase converts the (modified) IR back into a program/library that can be executed on the same platform as the original program/library without additional support libraries or a runtime environment.

Chapter 5 presented the results of an evaluation of Zipr, a prototype implementation of a static binary rewriter based on the architecture and algorithms described in the previous chapters. Zipr was developed solely to evaluate programs/libraries statically rewritten using the algorithms and architecture outlined in this dissertation and assess the flexibility of those algorithms to allow users to modify programs/libraries to include features that enhance the security and reliability of SOUP.

To assess whether the design and architecture of the static binary rewriter is practical, Zipr was applied to several large, real-world software applications and libraries. *glibc*, *libjvm*, Apache, and GNU Core Utilities were rewritten using Zipr and the Null Transformation to produce semantically equivalent statically rewritten programs and libraries. Based on the results of tests included with the software packages *glibc*, *libjvm* and GNU Core Utilities, Zipr generated statically rewritten versions that exhibited consistent behavior with their native counterparts. Testing with the external test framework Jmeter confirmed that the statically rewritten version of Apache functioned equivalently to the native version with respect to the tested behavior.

The performance evaluation relied on the programs of the SPEC 2006 benchmark suite and the DECREE platform developed for the DARPA CGC. For the experiments performed using the SPEC benchmark suite, the results included measurements of the changes in the overall performance overhead, on-disk overhead, maximum RSS, number of minor page faults for accessing program code and instruction cache usage. Measurement of the overall change in performance identified whether the statically rewritten programs/libraries perform as well as (or even better than) the originals. The other specific measurements pinpointed the reason for the change in the overall performance and identified areas for optimization.

For experiments performed using the DECREE platform and the CBs of the DARPA CGC dataset, the results included quantification of the Availability score and changes in the on-disk size, runtime memory usage and execution time. Like overall performance overhead for SPEC, the Availability score was used as a proxy for assessing the general performance of a statically rewritten program. The Availability score is a composite metric that takes into account *acceptable* levels of overhead but penalizes statically rewritten programs that exceed those levels. Again, the other, more precise metrics, helped pinpoint the reason for changes in the Availability and identified areas where the algorithms and architecture of the static binary rewriter presented in this dissertation could be improved to emit more efficient rewritten output.

Again, the design, architecture and algorithms described in Chapter 4 and the results of the evaluation of their prototypical implementation presented in Chapter 5 answered important open questions in the field of research into static binary rewriters. The work presented in Part II answers affirmatively the questions of whether it is possible to create a static binary rewriter for SOUP that generates artifacts that a) function reliably, b) do not require a copy of the original program/library code, and c) do not execute with unacceptable performance overhead. Furthermore, certain decisions made and solutions built during the design and implementation of the algorithms and architecture of the static binary rewriter presented herein (e.g., pinned addresses for immovable IBTs, sleds for handling dense code references) advanced the state-of-the-art and solved problems inherent in the development of a static binary rewriter that allows users to perform arbitrary transformations of the input program and generate rewritten programs/libraries that will operate in conditions with very strict performance criteria.

Although performance evaluation results indicate that the on-disk and runtime performance overhead of the programs/libraries rewritten with the basic algorithms, architecture and design presented in Part II is reasonable, the results also highlighted areas for improvement. In particular, three potential avenues for optimization were discovered thanks to the results of the evaluation. Part III described these optimizations to the fundamental architecture and algorithms presented in Part II. The algorithms presented optimized code layout and experiments evaluated the impact of those algorithms on the performance of statically rewritten programs/libraries. Alongside the description of each optimization was an overview of the historical research and related work associated with that technique.

The goal of the Locality Layout algorithm was to improve runtime performance of the statically rewritten programs/libraries by taking advantage of temporal/spatial locality. The optimization was based on the insight that the statically rewritten program/library will generate fewer page faults if the target code of instructions at pinned addresses is on the same page as those addresses and that a reduction in page faults will translate to improved performance at runtime. This optimization led to an overall performance improvement of approximately 2.5% for the applications of the SPEC benchmark suite and more than a 4% increase in the average Availability score for the CBs of the DARPA CGC dataset. The Locality Layout algorithm improved the instruction cache usage for statically rewritten programs of the SPEC benchmark suite by more than 17%. The Locality Layout algorithm decreased the number of CBs in the CGC dataset negatively impacted by excessive memory use at runtime. While the Locality Layout algorithm improved overall performance and certain metrics, it did not provide significant improvement on others. Analysis of the results spurred the creation of a refined version of the optimization.

Like the Locality Layout algorithm, the goal of the Profile Layout algorithm was to improve the overall performance of statically rewritten programs/libraries. The contribution of the Profile Layout algorithm to the

field was based on the premise that knowledge of the most common path of the execution of a program/library can be used during reassembly to build statically rewritten artifacts with a high degree of temporal/spatial locality. The optimization uses a novel algorithm based on a bid/sell auction mechanism for performing code placement that was originally designed to take profile data for input but was shown, surprisingly, to perform just as well when using default values as it did when using actual profile data.

For the applications of the SPEC benchmark suite, the Profile Layout algorithm improved overall performance by between 4.4% and 5.2% relative to the Locality Layout algorithm which itself was already an increase over the default algorithms of the Reassembly phase. The instruction cache usage improved by over 12% for applications of the SPEC benchmark suite relative to the Locality Layout algorithm but the other performance metrics showed no significant change. Improvements were also seen in the average availability score for the CBs of the DARPA CGC dataset with respect to the Locality Layout algorithm – more than a 7% increase. The Profile Layout algorithm decreased the number of CBs in the CGC dataset negatively impacted by excessive memory use at runtime.

Given the importance of the on-disk overhead in the context of the operating environment described in Part I, yet a third optimization was developed and tested. This optimization, the Relax Layout algorithm, targeted the on-disk size overhead directly and the runtime memory use indirectly. The Relax Layout algorithm combines the bid/sell auction technique for code placement from the Profile Layout algorithm with an algorithm for using the smallest possible branch instructions available on a target platform to implement links. This optimization is particularly helpful on architectures with variable-length instructions where different versions of branch instructions may be able to reach different parts of a program.

Thanks to this optimization, the on-disk filesize overhead of the statically rewritten versions of the applications of the SPEC benchmark suite decreased by more than 2% percent; the Relax Layout algorithm was the only one of the optimizations that significantly changed the on-disk overhead. The page fault overhead for the applications of the SPEC benchmark suite statically rewritten using the Relax Layout algorithm decreased by more than 5%; again, the Relax Layout algorithm was the only one of the optimizations that significantly improved the page fault overhead for the applications for the SPEC benchmark suite. However, it was shown that overall performance of applications of the SPEC benchmark suite rewritten using the Relax Layout algorithm was worse than that of the applications statically rewritten using the Profile Layout algorithm because of dollops sliding from their ideal location during relaxation.

The overall Availability of the statically rewritten versions of the CBs of the DARPA CGC dataset improved thanks to the Relax Layout algorithm relative to the Profile Layout algorithm which itself was an improvement over the Profile Layout algorithm – a 1.29% percent increase. The Relax Layout algorithm also decreased the number of CBs in the CGC dataset negatively impacted by excessive memory use at runtime.

The Locality and Profile Layout algorithm target overall performance while the Relax Layout algorithm targets the on-disk and runtime memory overhead. Based on the work in this dissertation, it is possible to conclude that researchers must consider temporal and spatial locality when creating any static binary rewriter. Comparing results for the Locality and Profile Layout algorithms with the Relax Layout algorithm showed that, in the presence of IBTs that cannot be moved (i.e., pinned addresses), not just any temporal/spatial locality improves performance. Simply building a rewritten program/library with smaller code size (the way that the Relax Layout algorithm does) that otherwise would improve locality must still recognize the fact that immovable IBTs dictate program control flow and take that into consideration when performing code placement to achieve the goal of building a static binary rewriter that generates artifacts with minimal runtime performance overhead. Combining the small code size of programs rewritten using the Relax Layout algorithm and the low runtime performance overhead of programs/libraries rewritten using the Profile Layout algorithm is the subject of future work and will, no doubt, produce additional insights valuable to the entire field of researchers working to develop static binary rewriters.

The evaluation of the optimizations showed not only that improvements in the efficiency of the statically rewritten programs and libraries were possible but that the attention to modular design and architecture of the static binary rewriter presented in this dissertation was important. Depending on the operating environment of the user deploying the static binary rewriter on his/her program, he/she can choose which of the optimizations to employ and gain their benefits without being forced into a particular tradeoff.

Part IV described three different practical applications for a retargetable, static binary rewriter.

Chapter 10 described the design and implementation of Mixr, a runtime rerandomization moving target defense that is an improvement on ASLR and protects vulnerable programs from return-to-libc, ROP and BROP attacks.

Within the context of the design and implementation of Mixr, a threat model and vocabulary were defined to make it easier to describe information leak attacks. The vocabulary and threat model will be useful to the general field of software security research by making it easier for security architects to evaluate and quantify the threat of information leak attacks and assess the efficacy of security solutions. *Oracles*, *queries*, *invalidation points* and the *attack window* all provide a means for systematically describing this class of attacks and the tools that can combat adversaries.

Mixr can be deployed on SOUP according to the needs of the security architect to protect vulnerable software against information leak attacks. Mixr allows the user to choose how often to rerandomize its address space and with what granularity to do so. The defense comes without the need for access to the vulnerable program/library's source code, a modified compiler, a modified linker or a modified kernel. This combination of features makes it unique among other existing runtime rerandomization techniques. In addition to offering

the security of runtime rerandomization, Mixr could be used to inject noise into the operation of sensitive programs and increase the difficulty of cracking software with side-channel attacks.

Chapter 11 described the design and implementation of DCR, an improvement on stack canaries, the powerful, effective method for protecting programs against attacks targeting stack overflow vulnerabilities.

The chapter began by describing the type of attack defended by DCR in terms of the vocabulary defined in Chapter 10 demonstrating the utility of that taxonomy. It continued with a description of the DCR defense and showed how it improved upon the traditional but successful defense of software using stack canaries by demonstrating the DCR prevents the state-of-the art BROP attack while stack canaries do not. Like Mixr, DCR is a highly-customizable technique that the security architect can deploy according a specific risk tolerance.

Chapter 12 described an implementation of enforcement of control-flow integrity, a technique that protects against program hijack at runtime using a retargetable, static binary rewriter.

Although SelectiveCFI showed no runtime performance improvement with respect to other CFI implementations, the discussion of SelectiveCFI showed how it is an improvement over existing work. First, it works on SOUP and improves on existing CFI implementation's AIR, a metric proposed by Zhang et al. for quantifying the effectiveness of the protection offered by an implementation of CFI [251]. Second, it has low on-disk overhead. Third, and finally, it is customizable according to the needs of the security architect and the environment in which it will be deployed – the architect can choose to use fine- or coarse-grained instrumentation or bypass protection entirely on safe functions. This combination makes it unique among the existing implementations of CFI. Furthermore, work presented subsequent to the preparation of this dissertation demonstrate the ability to lower the overhead of the static binary rewriter itself which lowers the overhead of programs protected with SelectiveCFI.

The ability to protect against information leaks and control-flow hijack attacks without tying the security architect to a specific set of choices make Mixr, DCR and SelectiveCFI worthy of discussion independent of their utility in demonstrating the power of the architecture, algorithms and design of the static binary rewriter presented in this dissertation. The combination of their ability to protect SOUP while affording the user the chance to deploy the security solution according to his/her risk assessment make Mixr, DCR and SelectiveCFI unique among existing work.

The architecture and algorithms presented in this dissertation and implemented for testing and evaluation in Zipr prove that the creation of a static binary rewriter that can be used to improve to apply *post hoc* transformation to SOUP that improve the security, safety and reliability of critical software systems is difficult but not impossible.

Appendices

Appendix A

Locality Layout Algorithm Evaluation

The overall assessment of the importance of these results was discussed in Section 7.4. The details of the tests results that informed those assesments are presented in this chapter. The applications of the SPEC benchmark suite and the CBs of the CGC dataset were evaluated in order to assess whether the Locality Layout algorithm achieves the goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs.

A.1 SPEC

Figures A.1 and A.2 show the difference in performance for each of the applications of the SPEC benchmark suite when those programs were rewritten using the Locality Layout algorithm as opposed to the default algorithms of the Reassembly phase.

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm on Host A was 1.288x. This compares favorably to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the default algorithms of the Reassembly phase (1.323x). In fact, it is a 2.666% improvement.

On average, the performance overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm on Host B was 1.253x. This compares favorably to the average overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the default algorithms of the Reassembly phase (1.285x). It is a 2.539% improvement.

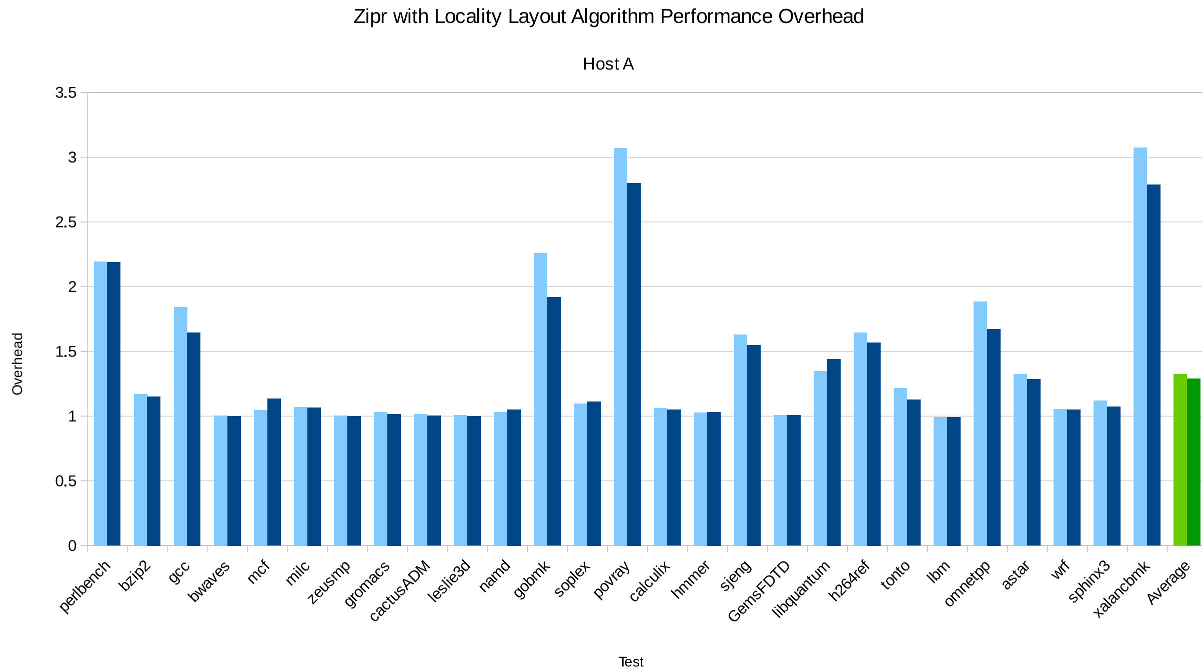

Figure A.1: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

A.2 Memory Usage

Figures A.3 and A.3 show the difference in the RSS overhead for each of the applications of the SPEC benchmark suite when those programs were rewritten using the Locality Layout algorithm as opposed to the default algorithms of the Reassembly phase.

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm on Host A was 1.010x. This overhead is roughly equivalent to the average RSS overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the default algorithms of the Reassembly phase (1.012x). It is just under a fifth of a percent.

On average, the RSS overhead of the programs of the SPEC2006 benchmark suite statically rewritten with the Locality Layout algorithm on Host B was 1.014x. This is roughly equivalent to the average RSS overhead of the applications of the SPEC2006 benchmark suite statically rewritten with the default algorithms of the Reassembly phase (1.012x). In this case the RSS overhead for the benchmark applications rewritten using the “optimized” layout algorithm got worse but the difference was within a quarter of a percent.

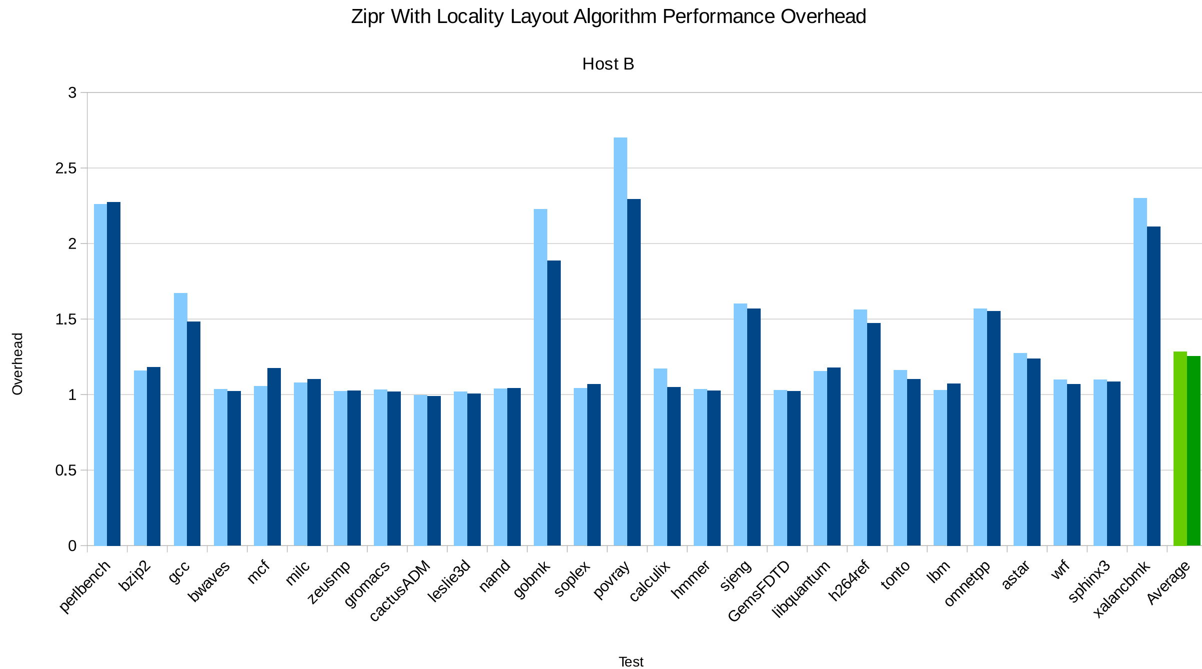

Figure A.2: Performance overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

Taken together, the results for Hosts A and B demonstrate that the Locality Layout algorithm did not significantly improve the RSS overhead of the programs of the SPEC2006 benchmark.

Figures A.5, A.6 show the page fault overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. On Host A, the overhead ranged from 1.22% to 287.41%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.291x more page faults than the native versions. That compares favorably (a .85% improvement) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using the default algorithsm of the Reassembly phase (1.302x). On Host B, the overhead ranged from less than 1.14% to 270.12%. On average, the statically rewritten versions of the SPEC benchmark programs caused 1.283x more page faults than the native versions on Host B. That compares favorably (a .78% improvement) to the page fault overhead of the applications of the SPEC benchmark suite statically rewritten using the default algorithms of the Reassembly phase (1.293x).

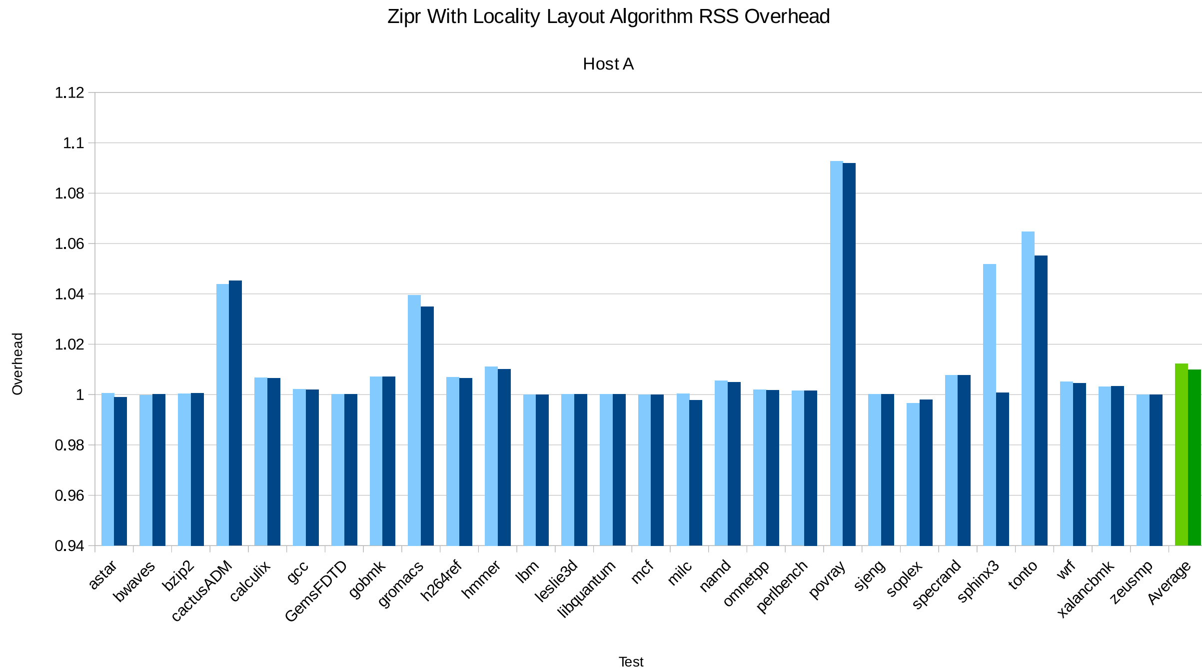

Figure A.3: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

A.3 Instruction Cache Usage

As discussed in Section 5.5.3, the instruction cache is an important part of a host's hardware designed to improve performance. Using an algorithm like the Locality Layout algorithm that reassembles statically rewritten programs and libraries to take advantage of temporal locality will improve the statically rewritten program's use of the instruction cache.

Figures A.7 and A.8 show results that prove this assertion. The For Hosts A and B, the versions of the programs of the SPEC benchmark suite statically rewritten using the Locality Layout algorithm incur 4.32x and 4.66x as many instruction cache misses as their native counterparts, respectively. Those overheads are improvements of more than 17% for both hosts when compared to the instruction cache miss overhead of the programs of the SPEC benchmark suite statically rewritten using the default algorithms of the Reassembly phase. The improvements are significant and contribute to the overall performance improvement discussed earlier.

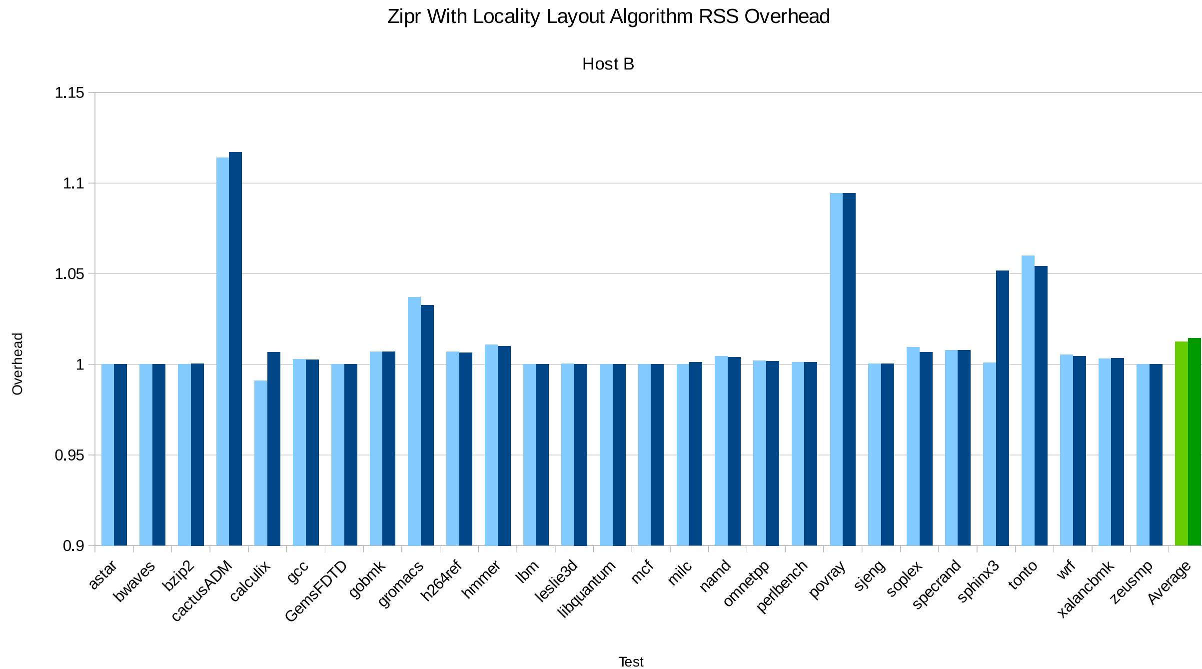

Figure A.4: Maximum RSS overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

A.4 Filesize Overhead

Figures A.9, and A.10 show the filesize overhead for statically rewritten versions of each of the programs in the SPEC benchmark suite. For Host A, the average filesize overhead of the applications of the SPEC benchmark suite was 1.086x. For Host B, the overhead was 1.092x. An overhead of less than 10% in both cases shows that the Locality Layout algorithms produce statically rewritten programs/libraries with minimal impact on filesize [216, 11].

Compared to the filesize overhead of the programs of the SPEC2006 benchmark suite rewritten using the default algorithms of the Reassembly phase, the Locality Layout algorithm has mixed results. For Host A, there is less than a 1% improvement and for Host B there is less than a 1% impairment. These results demonstrate that the Locality Layout algorithm has minimal, if any, impact on the filesize overhead of the statically rewritten program/library when compared to the default algorithms of the Reassembly phase.

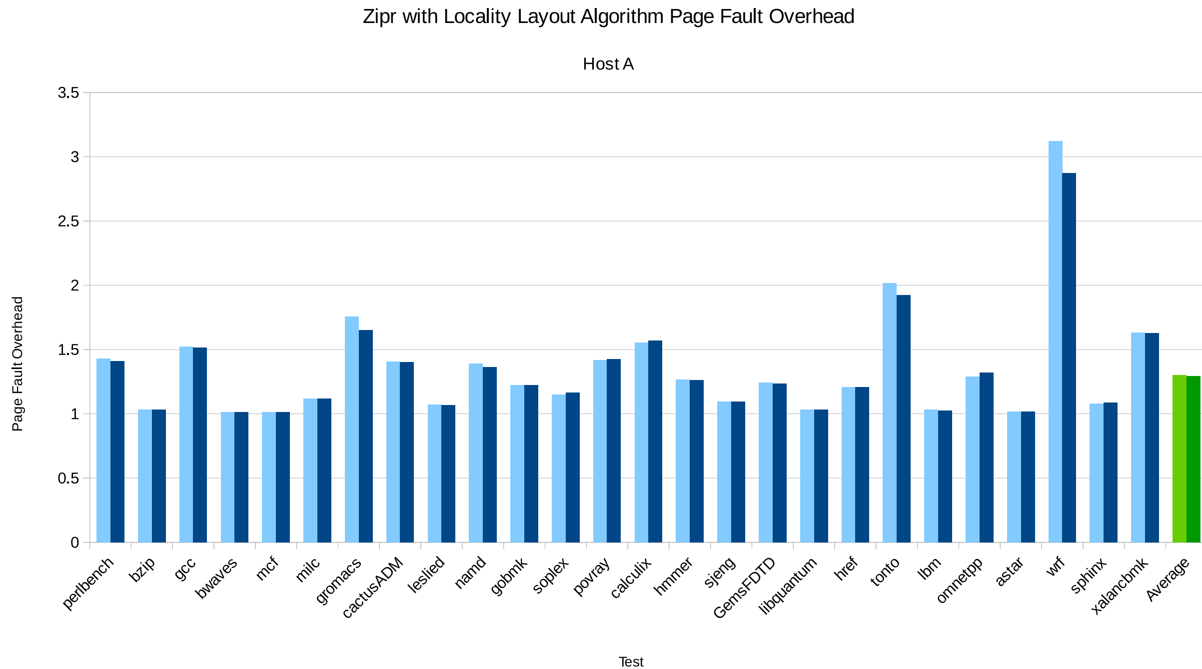

Figure A.5: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

A.5 Placement Ratios

The discussion in Section 7.3 indicates that there are conditions under which the Locality Layout algorithm may fail to place a dollop on the same page as the link that targets it and/or fails to place dollops on the same page in the rewritten program that they were on in the original program. Data gathered during the process of statically rewriting the applications of the SPEC2006 benchmark suite with the Locality Layout algorithm demonstrates that this is not just a hypothetical consideration.

Figure A.11 compares the *placement ratios* for Hosts A and B. The placement ratio is the percentage of dollops that the layout algorithm places in the preferred position. The preferred position depends on the layout algorithm. In the case of the Locality Layout algorithm, a dollop is successfully placed when it is placed on the same page as the link that targets it or it is placed on the same page in the rewritten program that it was on in the original program. The placement ratios are nearly the same for Hosts A and B. Since dollop placement in the layout algorithm depends exclusively on the nature of the original program, this is expected since both hosts generate the original applications with identical versions of the compiler. The placement ratios for Host A and B are 61.38% and 61.07%, respectively.

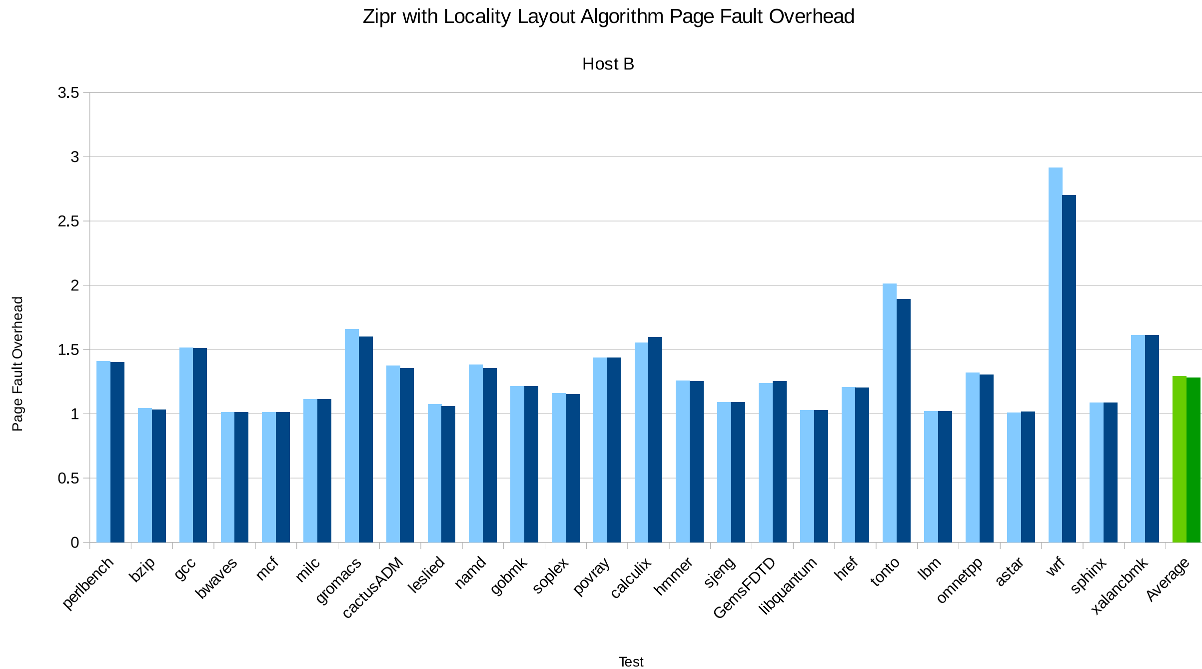

Figure A.6: Minor page fault overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green.

A.6 CGC

The more than 140 CBs in the CGC dataset provide another way to assess whether the Locality Layout algorithm achieves its stated goal of lowering the memory overhead of statically rewritten programs and, therefore, improves the overall performance of those programs. See Section 5.6 for a complete discussion of the CGC dataset and the terms used in the description of the evaluation.

Figure A.12 shows the availability scores for each of the RCBs generated by Zipr using the Locality Layout algorithm instead of the default algorithms of the Reassembly phase.[1] The average availability score was 87.09%. This compares well with the average availability score for the RCBs generated by Zipr using the default algorithms of the Reassembly phase (83.42%) – a 4.4% increase.

Table A.1 shows the distribution of dominators for the RCBs in the CGC dataset generated using the Locality Layout algorithm. Overall there are 54 RCBs with a perfect availability score compared with only 42 RCBs with a perfect availability score when statically rewritten using the default algorithms of the Reassembly phase. The distribution shows a notable decrease in the number of RCBs that have a lower availability score

[1]The Null Transformation was the only transformation applied to the CBs.

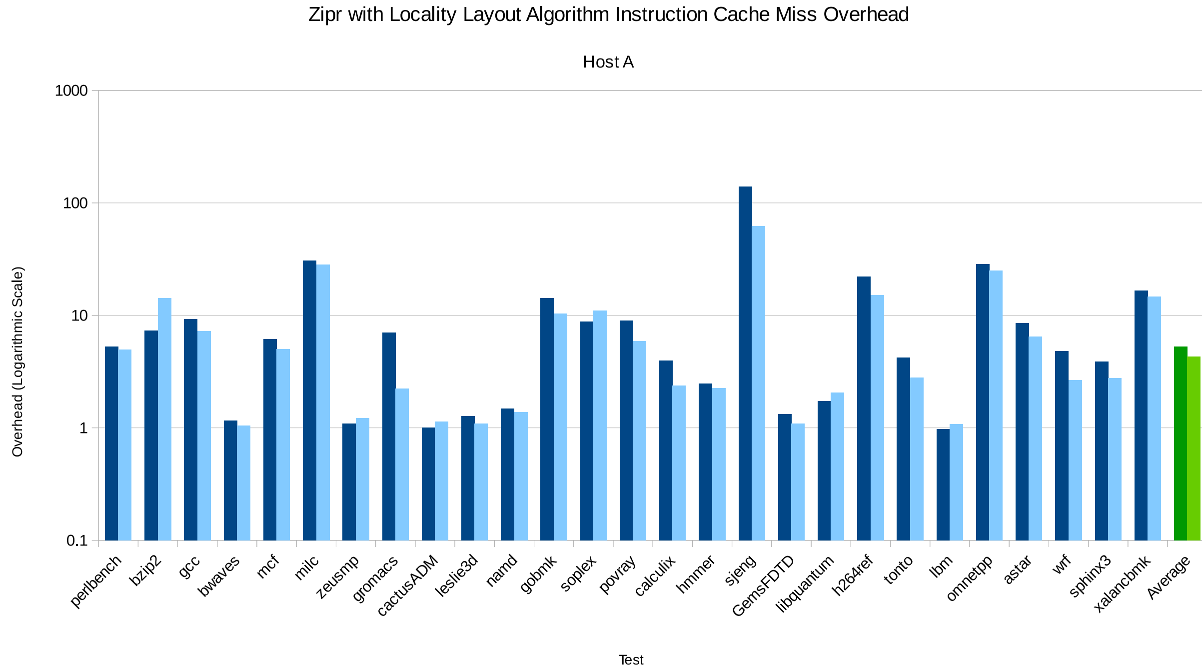

Figure A.7: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm without profiles. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green.

Category	Default	Locality
None	42	54
Performance	16	12
Filesize	1	2
Memory	83	74
Functionality	1	1

Table A.1: Distribution of dominators for the CBs in the CGC dataset when reassembled using the Locality Layout algorithm.

because of memory overhead. Of the RCBs that have less-than-perfect availability, more than 83% are most negatively impacted by their memory overhead. In absolute terms, there is an 10.84% improvement in the number of RCBs with less-than-perfect availability score whose penalty is dominated by memory overhead when compared with the RCBs generated by the default algorithms of the Reassembly phase.

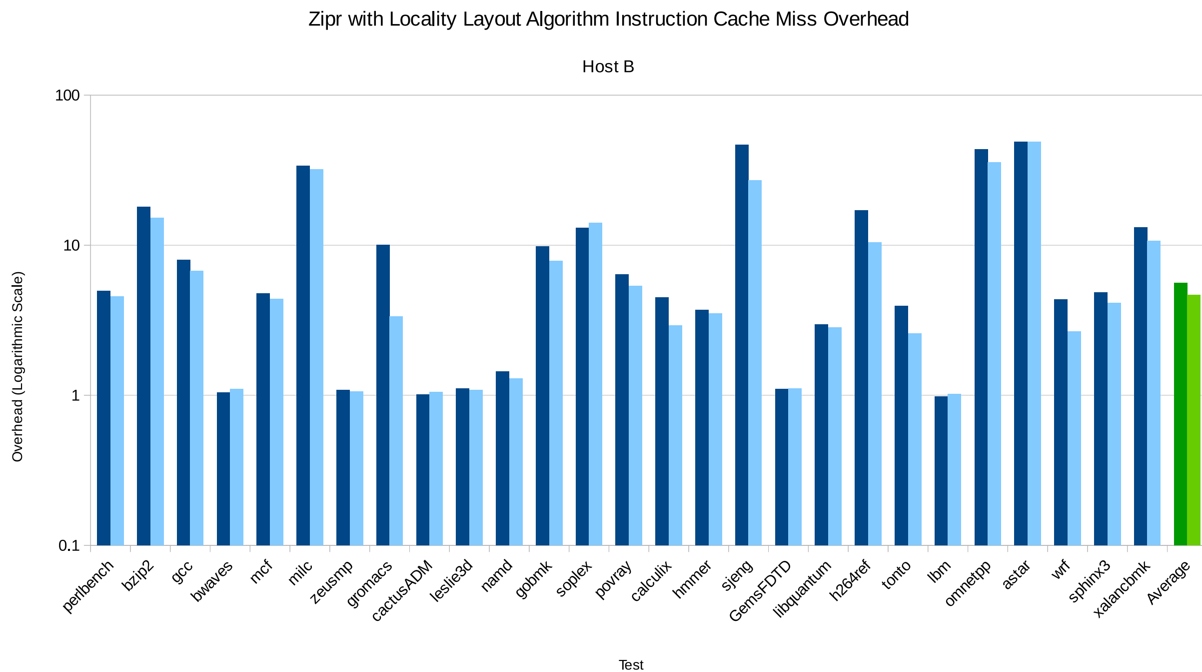

Figure A.8: Level 1 Instruction Cache miss overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm without profiles. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. For comparison, the results for the applications reassembled using the deafult algorithms of the Reassembly phase are shown on the left, in light blue and light green.

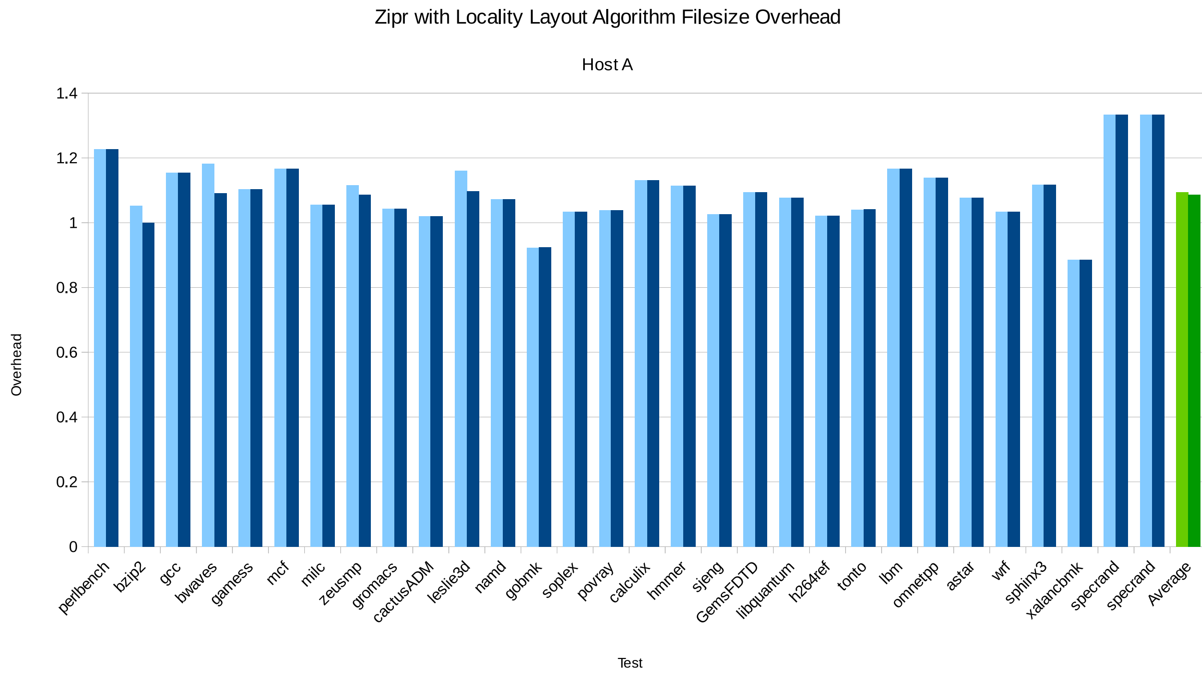

Figure A.9: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host A. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

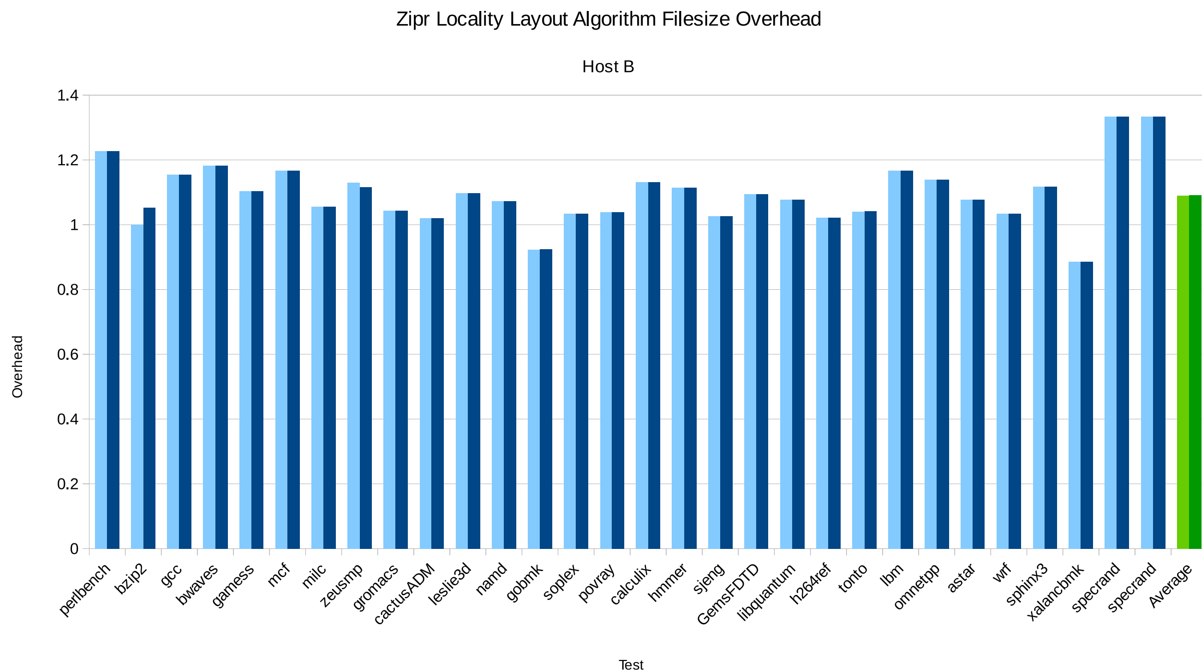

Figure A.10: Filesize overhead of the applications in the SPEC2006 benchmark suite when reassembled using the Locality Layout algorithm. These results are for Host B. The results for the applications reassembled using the Locality Layout algorithm are shown on the right, in dark blue and dark green. The results for the applications reassembled using the default algorithms of the Reassembly phase are shown on the left, in light blue and light green.

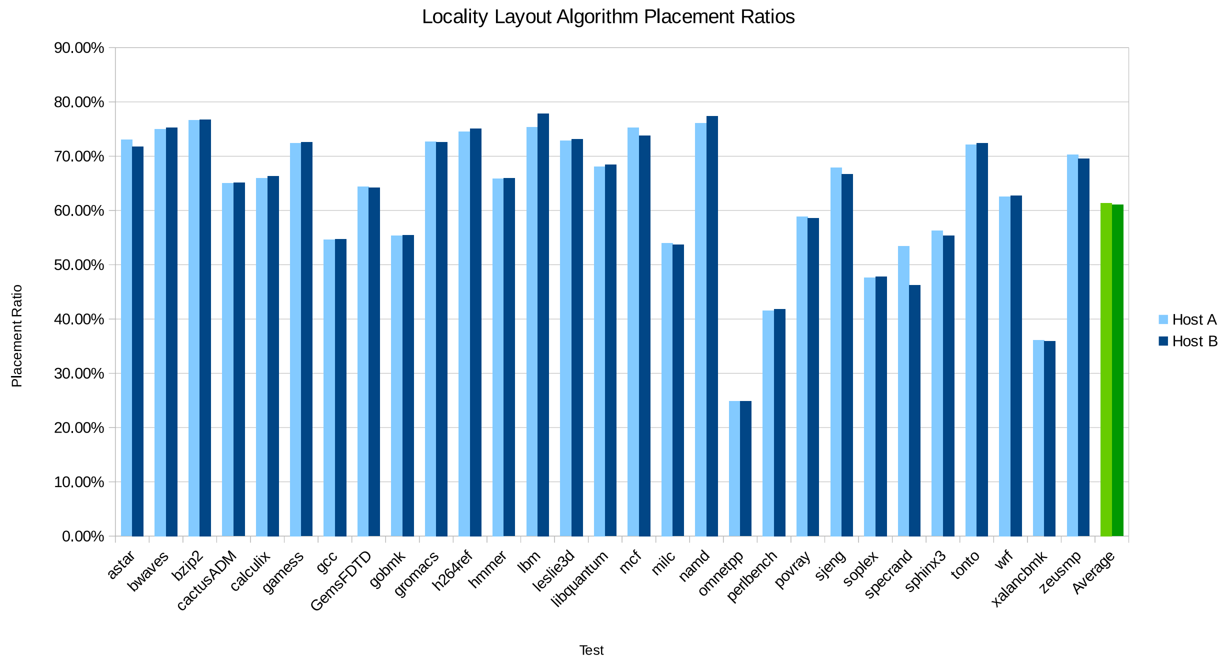

Figure A.11: The percentage of dollops that could be placed either on the same page as their link or the same page that they were on in the original program/library. Host A's placement ratios are shown in light blue and light green. Host B's placement ratios are shown in dark blue and dark green.

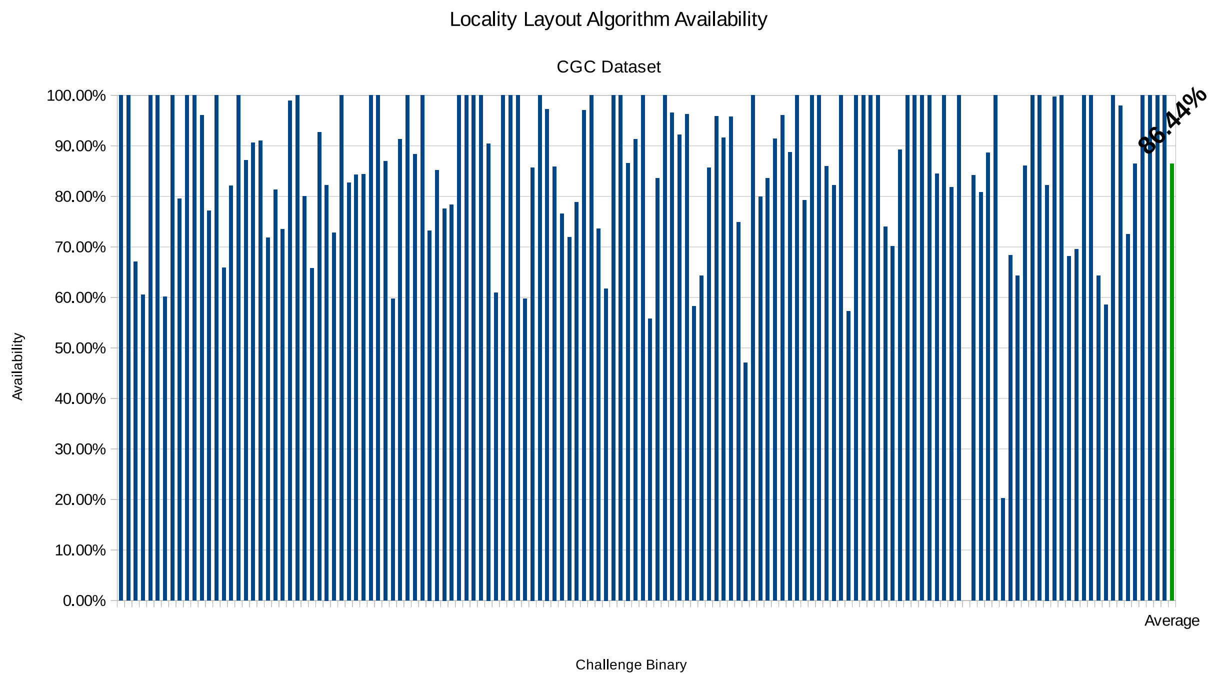

Figure A.12: Availability scores for the RCBs in the CGC dataset when reassembled using the Locality Layout algorithm.

Bibliography

[1] A Portrait of Smartphone Ownership — Pew Research Center, Retrieved 2016-05-09 from http://www.pewinternet.org/2015/04/01/chapter-one-a-portrait-of-smartphone-ownership/.

[2] Instrumentation Options - Using the GNU Compiler Collection (GCC), Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[3] Intel® Quark SoC X1000 SeriesGet Started, Retrieved 2016-05-10 from http://www.intel.com/content/www/us/en/embedded/products/quark/x1000/overview.html.

[4] Microsoft BlueHat Prize Contest Official Rules, Retrieved 2012-02-11 from http://web.archive.org/web/20120211224214/http://www.microsoft.com:80/security/bluehatprize/rules.aspx.

[5] OpenBenchmarking.org - Server Motherboard Test Suite Collection, Retrieved 2016-05-10 from https://openbenchmarking.org/suite/pts/server.

[6] Overview of x64 Calling Conventions, Retrieved 2016-05-30 from https://msdn.microsoft.com/en-us/library/ms235286.aspx.

[7] Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing, Open-Source Benchmarking, Retrieved 2016-05-10 from http://www.phoronix-test-suite.com/.

[8] Title 14: Aeronautics and Space PART 33 - AIRWORTHINESS STANDARDS: AIRCRAFT ENGINES Subpart B - Design and Construction; General, Retrieved from http://www.ecfr.gov/cgi-bin/text-idx?SID=07205d4ce53ae3f5d709d030d7c6103b&mc=true&node=se14.1.33_128&rgn=div8.

[9] Guidance for the Industry, FDA and Compliance on Off-The-Shelf Software Uses in Medical Devices. Tech. rep., U.S. Department Of Health And Human Services Administration Food and Drug Administration Center for Devices and Radiological Health Office of Device Evaluation, 1999.

[10] Intel® Xeon® Processor 5600 Series. Tech. rep., Intel Corporation, 2010.

[11] DARPA Cyber Grand Challenge: CQE Scoring Document. Tech. rep., Defense Advanced Research Projects Agency, 2014.

[12] DARPA Cyber Grand Challenge: CQE Scoring Document. Tech. rep., DARPA Information Innovation Office, 2014.

[13] Intel® Xeon® Processor E5-1600/E5-2600/E5-4600 v2 Product Families. Tech. rep., Intel Corporation, 2014.

[14] Net Losses: Estimating the Global Cost of Cybercrime. Tech. rep., Center for Strategic and International Studies; Mcafee Corporation, 2014.

[15] No Title, 2016, Retrieved from https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[16] Transparent Hugepage Support, 2016, Retrieved from https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[17] Your tool works better than mine? Prove it, 2016, Retrieved 2017-12-08 from https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/.

[18] Dynamically writing polls, 2017, Retrieved from https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/understanding-poll-generators.md.

[19] Optimization Options - Using the GNU Compiler Collection (GCC), 2017, Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[20] Safety Communications - Firmware Update to Address Cybersecurity Vulnerabilities Identified in Abbott's (formerly St. Jude Medical's) Implantable Cardiac Pacemakers: FDA Safety Communication. Tech. rep., Food and Drug Administration, 2017.

[21] Submitting a Challenge Binary, 2017, Retrieved from https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/submitting-a-cb.md.

[22] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, New York, USA, 2005), ACM Press, p. 340.

[23] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, USA, 1986.

[24] Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., and Barua, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, New York, USA, Apr. 2013), ACM Press, p. 295.

[25] Anand, K., Smithson, M., Koth, A., Elwazeer, K., and Barua, R. Decompilation to Compiler High IR in a binary rewriter. Tech. rep., University of Maryland, 2010.

[26] Andreesen, M. Why Software Is Eating the World, Aug. 2011, Retrieved from http://www.wsj.com/articles/SB10001424053111903480904576512250915629460.

[27] Apache JMeter. Apache JMeter, 2016, Retrieved 2016-09-28 from http://jmeter.apache.org/.

[28] Arnold, G. W. Laying the Foundation for the Electric Grid's Next 100 Years, 2011, Retrieved from http://www.nist.gov/smartgrid/upload/Arnold-ETSI04052011.pdf.

[29] Backus, J. The history of Fortran I, II, and III. *History of programming languages I* (1978), 25–74.

[30] Balakrishnan, G., and Reps, T. WYSINWYX. *ACM Transactions on Programming Languages and Systems 32*, 6 (Aug. 2010), 1–84.

[31] Ball, T., Mataga, P., and Sagiv, M. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, New York, USA, 1998), POPL '98, ACM, pp. 134–148.

[32] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. On the (im)possibility of obfuscating programs. *Journal of the ACM 59*, 2 (May 2012), 6:1—-6:48.

[33] Barr, M. Bookout v. Toyota: 2005 Camry L4 Software Analysis. Tech. rep., Barr Group, 2013.

[34] Barr, T. W., Cox, A. L., Rixner, S., Barr, T. W., Cox, A. L., and Rixner, S. Translation caching. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, New York, USA, 2010), Vol. 38, ACM Press, p. 48.

[35] Bendersky, E. Position Independent Code (PIC) in shared libraries on x64, 2011, Retrieved 2017-09-01 from http://eli.thegreenplace.net/2011/11/11/position-independent-code-pic-in-shared-libraries-on-x64/.

[36] Bendersky, E. Assembler relaxation, 2013, Retrieved 2016-09-08 from http://eli.thegreenplace.net/2013/01/03/assembler-relaxation.

[37] Bernat, A. R., and Miller, B. P. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (New York, New York, USA, 2011), ACM Press, p. 9.

[38] Bhagat, I., Gibert, E., Sánchez, J., González, A., Bhagat, I., Gibert, E., Sánchez, J., and González, A. Global productiveness propagation: A code optimization technique to speculatively prune useless narrow computation. *ACM SIGPLAN Notices 46*, 5 (Apr. 2011), 161.

[39] Bhatkar, S., DuVarney, D. C., and Sekar, R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium* (2003), USENIX Association, pp. 8–8.

[40] Bigelow, D., Hobson, T., Rudd, R., Streilein, W., and Okhravi, H. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, New York, USA, Oct. 2015), ACM Press, pp. 268–279.

[41] Bishop, P., Bloomfield, R., and Froome, P. Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications. Tech. rep., Adelard.

[42] Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., and Boneh, D. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (May 2014), IEEE, pp. 227–242.

[43] Blazakis, and Dionysus. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (2010), USENIX Association, pp. 1–9.

[44] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel*, 2nd ed. ed. O'Reilly, Sebastopol, California, USA, 2003.

[45] Buse, R. P. L., and Weimer, W. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 144–154.

[46] Cabutto, A., Falcarin, P., Abrath, B., Coppens, B., and De Sutter, B. Software Protection with Code Mobility. In *Proceedings of the Second ACM Workshop on Moving Target Defense* (New York, New York, USA, 2015), ACM Press, pp. 95–103.

[47] Calder, B. G., Grunwald, D. C., Zorn, B. G., Calder, B. G. ., and Grunwald, D. C. . Quantifying Behavioral Differences Between C and C++ Programs. Tech. rep., University of Colorado, Boulder, 1994.

[48] Chao Zhang, C., Tao Wei, T., Zhaofeng Chen, Z., Lei Duan, L., Szekeres, L., McCamant, S., Song, D., and Wei Zou, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (May 2013), IEEE, pp. 559–573.

[49] Chen, X., Bos, H., and Giuffrida, C. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *Proceedings of the 2017 IEEE European Symposium on Security and Privacy* (Apr. 2017).

[50] Chen, Y., Wang, Z., Whalley, D., and Lu, L. Remix: On-demand live randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy* (New York, New York, USA, Mar. 2016), CODASPY '16, ACM Press, pp. 50–61.

[51] Clark, S., Frei, S., Blaze, M., and Smith, J. Familiarity breeds contempt. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, New York, USA, 2010), ACM Press, p. 251.

[52] Cole, M., Esposito, R., Biddle, S., and Grim, R. Top-Secret NSA Report Details Russian Hacking Effort Days Before 2016 Election, 2017, Retrieved from https://theintercept.com/2017/06/05/top-secret-nsa-report-details-russian-hacking-effort-days-before-2016-election/.

[53] Conti, M., Crane, S., Frassetto, T., Homescu, A., Koppen, G., Larsen, P., Liebchen, C., Perry, M., and Sadeghi, A.-R. Selfrando: Securing the Tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies 2016*, 4, 454–469.

[54] Cook, S. P., Angermayer, J., Lacher, A., Buttner, A., Crouse, K., and Lester, E. Dependability of software of unknown pedigree: Case studies on unmanned aircraft systems. In *Proceedings of the 34th IEEE/AIAA Digital Avionics Systems Conference* (Sep. 2015), IEEE, pp. 5E5–1–5E5–20.

[55] Cortes, E. DRE Certification Recommendation. Tech. rep., Virginia State Board of Elections, Richmond, Virginia, USA, 2017.

[56] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. Pointguard TM: Protecting pointers from buffer overflow vulnerabilities, 2003.

[57] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium* (1998), pp. 63–78.

[58] Crandall, J. R., Wu, S. F., and Chong, F. T. Minos. *ACM Transactions on Architecture and Code Optimization 3*, 4 (Dec. 2006), 359–389.

[59] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Fraz, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 2015 Network and Distributed System Security* (San Diego, California, USA, 2015), Internet Society.

[60] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.-R., Brunthaler, S., and Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (May 2015), IEEE, pp. 763–780.

[61] Crane, S. J., Franz, M., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.-R., Holz, T., and De Sutter, B. It's a TRaP. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security* (New York, New York, USA, 2015), ACM Press, pp. 243–255.

[62] Curtsinger, C., Berger, E. D., Curtsinger, C., Berger, E. D., Curtsinger, C., and Berger, E. D. STABILIZER. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, New York, USA, 2013), Vol. 41 of *ASPLOS '13*, ACM Press, p. 219.

[63] Davi, L., Liebchen, C., Sadeghi, A.-R., Snow, K. Z., and Monrose, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 2015 Network and Distributed System Security* (San Diego, California, USA, 2015), Internet Society.

[64] Davi, L. V., Dmitrienko, A., Nürnberger, S., and Sadeghi, A.-R. Gadge me if you can. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security* (New York, New York, USA, 2013), ACM Press, p. 299.

[65] Davidson, M. A. Mandated Third Party Static Analysis: Bad Public Policy, Bad Security, 2014, Retrieved 2016-09-21 from https://blogs.oracle.com/maryanndavidson/entry/mandated_third_party_static_analysis.

[66] Do, H., Elbaum, S., and Rothermel, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering 10*, 4 (Oct. 2005), 405–435.

[67] Eager, M. J. Introduction to the DWARF Debugging Format. Tech. rep., Eager Consulting, 2012.

[68] Eagle, C. *The IDA Pro Book*, 2nd ed. No Starch Press, San Francisco, California, USA, 2011.

[69] Edwards, A., Srivastava, A., and Vo, H. Vulcan: Binary transformation in a distributed environment. Tech. rep., Apr. 2001.

[70] ElWazeer, K., Anand, K., Kotha, A., Smithson, M., Barua, R., ElWazeer, K., Anand, K., Kotha, A., Smithson, M., and Barua, R. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Notices 48*, 6 (Jun. 2013), 51.

[71] Enst, M. D., Czeisler, A., Griswold, W. G., and Notkin, D. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering* (New York, New York, USA, 2000), ACM Press, pp. 449–458.

[72] Erlingsson, U., and Schneider, F. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, IEEE Comput. Soc, pp. 246–255.

[73] Eustace, A., and Srivastava, A. ATOM: A Flexible interface for Building High Performance Program Analysis Tools. Tech. rep., Digital Western Research Laboratory, 1994.

[74] FDA. Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices. Tech. rep., Center for Devices and Radiological Health Center for Biologics Evaluation and Research, 2005.

[75] FDA. MAUDE Adverse Event Report: MERGE HEALTHCARE MERGE HEMO PROGRAMMABLE DIAGNOSTIC COMPUTER, 2016, Retrieved 2016-06-28 from https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfmaude/detail.cfm?mdrfoi__id=5487204.

[76] Fleming, P. J., and Wallace, J. J. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM 29*, 3 (Mar. 1986), 218–221.

[77] Flexera. Open Source Risk - Fact or Fiction? Tech. rep., Flexera, Itasca, Illinois, USA, 2017.

[78] Fog, A. Optimizing Subroutines in Assembly Language. Tech. rep., Technical University of Denmark, 2017.

[79] Fog, A. The Microarchitecture of Intel, AMD and VIA CPUs. Tech. rep., Technical University of Denmark, 2017.

[80] Fraser, C. W., Myers, E. W., Wendt, A. L., Fraser, C. W., Myers, E. W., and Wendt, A. L. Analyzing and compressing assembly code. *ACM SIGPLAN Notices 19*, 6 (Jun. 1984), 117–121.

[81] Free Software Foundation, I. The GNU C Library, 2017, Retrieved from https://www.gnu.org/software/libc/manual/.

[82] Friedman, S., Musliner, D., and Keller, P. Chronomorphic programs: Runtime diversity prevents exploits and reconnaissance. *International Journal on Advances in Security 8*, 3-4 (2015), 120–192.

[83] Furber, S. B. *ARM System Architecture*. Addison-Wesley, Harlow, England, 1996.

[84] Gadaleta, F., Younan, Y., and Joosen, W. BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks. In *Proceedings of the Second international conference on Engineering Secure Software and Systems*. Springer-Verlag, 2010, pp. 1–17.

[85] Gisbert, H. M., and Ripoll, I. On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows. In *Proceedings of the 13th IEEE International Symposium on Network Computing and Applications* (Aug. 2014), IEEE, pp. 145–152.

[86] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st {USENIX} Security Symposium* (Bellevue, Washington, USA, 2012), USENIX, pp. 475–490.

[87] Goodspeed, T., and Francillon, A. Half-blind attacks: Mask ROM bootloaders are dangerous.

[88] Guillon, C., Rastello, F., Bidault, T., and Bouchez, F. Procedure placement using temporal-ordering information. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (New York, New York, USA, 2004), ACM Press, p. 268.

[89] Gupta, A., Habibi, J., Kirkpatrick, M. S., and Bertino, E. Marlin: Mitigating code reuse attacks using code randomization. *IEEE Transactions on Dependable and Secure Computing 12*, 3 (May 2015), 326–337.

[90] Halili, E. H. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites.* Packt Publishing, Limited, Birmingham, England, 2008.

[91] Harizopoulos, S., and Ailamaki, A. Improving instruction cache performance in OLTP. *ACM Transactions on Database Systems 31*, 3 (Sep. 2006), 887–920.

[92] Harris, L. C., and Miller, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (Dec. 2005), 63.

[93] Hasabnis, N., Sekar, R., Hasabnis, N., Sekar, R., Hasabnis, N., Sekar, R., Hasabnis, N., and Sekar, R. Lifting assembly to intermediate representation. *ACM SIGARCH Computer Architecture News 44*, 2 (2016), 311–324.

[94] Hawkins, W. H., Co, M., Hiser, J. D., Nguyen-Tuong, A., and Davidson, J. W. Zipr: Efficient static binary rewriting for security. In *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks* (2017).

[95] Hawkins, W. H., Hiser, J. D., and Davidson, J. W. Dynamic canary randomization for improved software security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference* (New York, New York, USA, 2016), ACM Press, pp. 1–7.

[96] Hiser, J., Nguyen-Tuong, A., Hawkins, W. H., McGill, M., Co, M., and Davidson, J. Zipr++: Exceptional binary rewriting. In *Proceedings of the 2nd Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA, 2017).

[97] Hiser, J. D., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. {ILR}: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 571–585.

[98] Hiser, J. D., Nguyen-Tuong, A., Co, M., Rodes, B., Hall, M., Coleman, C. L., Knight, J. C., and Davidson, J. W. A gramework for creating binary rewriting tools (Short Paper). In *Proceedings of the 10th European Dependable Computing Conference* (Washington, DC, USA, 2014), EDCC '14, IEEE Computer Society, pp. 142–145.

[99] Hobson, T., Okhravi, H., Bigelow, D., Rudd, R., and Streilein, W. On the challenges of effective movement. In *Proceedings of the 1st ACM Workshop on Moving Target Defense* (New York, New York, USA, 2014), ACM Press, pp. 41–50.

[100] Holzmann, G. J. The Power of 10: Rules for Developing Safety-Critical Code. Tech. rep., NASA/JPL Laboratory for Reliable Software, 2006.

[101] Homescu, A., Brunthaler, S., Larsen, P., Franz, M., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. librando: Transparent Code Randomization for Just-in-Time Compilers. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (New York, New York, USA, 2013), ACM Press, pp. 993–1004.

[102] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., and Franz, M. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization* (Feb. 2013), IEEE, pp. 1–11.

[103] Horspool, R. N., and Marovac, N. An approach to the problem of detranslation of computer programs. *The Computer Journal 23*, 3 (Aug. 1980), 223–229.

[104] Hwang, Y.-S., Lin, T.-Y., and Chang, R.-G. DisIRer. *ACM Transactions on Architecture and Code Optimization 7*, 4 (Dec. 2010), 1–36.

[105] IARPA. Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) - Program Description, Retrieved 2016-09-21 from https://www.iarpa.gov/index.php/research-programs/stonesoup.

[106] IARPA. Securely Taking on New Executable Software of Unknown Provenance (STONESOUP), 2009, Retrieved 2016-09-21 from https://www.iarpa.gov/index.php/research-programs/stonesoup/baa.

[107] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2016.

[108] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*. 2017.

[109] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3*. 2017.

[110] Jajodia, S., Ghosh, A. K., Swarup, V., Wang, C., and Wang, X. S., Eds. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, Vol. 54 of *Advances in Information Security*. Springer New York, New York, New York, USA, 2011.

[111] Jantz, M. R., Robinson, F. J., and Kulkarni, P. A. Impact of intrinsic profiling limitations on effectiveness of adaptive optimizations. *ACM Transactions on Architecture and Code Optimization 13*, 4 (Dec. 2016), 44:1—-44:26.

[112] Joao, J., Mutlu, O., Kim, H., and Patt, Y. N. Dynamic predication of indirect jumps. *IEEE Computer Architecture Letters 7*, 1 (Jan. 2008), 1–4.

[113] Johnson, S. C. A portable compiler. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, New York, USA, Jan. 1978), ACM Press, pp. 97–104.

[114] Jun Xu, Kalbarczyk, Z., and Iyer, R. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems* (2003), IEEE Comput. Soc, pp. 260–269.

[115] Kaplan, D., Kedmi, S., Hay, R., and Dayan, A. Attacking the Linux PRNG on Android: Weaknesses in seeding of entropic pools and low boot-time entropy. In *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT 14)* (San Diego, California, USA, 2014), USENIX Association.

[116] Kaprica Security, I. KPRCA-00044 (FUN), 2017, Retrieved from https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges/KPRCA_00044.

[117] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communication Security* (New York, New York, USA, 2003), ACM Press, p. 272.

[118] Kil, C., Jun, J., Bookholt, C., Xu, J., and Ning, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (Dec. 2006), IEEE, pp. 339–348.

[119] Kistler, T., and Franz, M. Continuous program aptimization: A case study. *ACM Transactions on Programming Languages and Systems 25*, 4 (Jul. 2003), 500–548.

[120] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in Cryptology*, M. Wiener, Ed., 1 ed. Springer-Verlag Berlin Heidelberg, 1999, pp. 388–397.

[121] Koopman, P. A Case Study of Toyota Unintended Acceleration and Software Safety. Tech. rep., 2014.

[122] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium* (2004), USENIX Association, pp. 18–18.

[123] Lalande, J.-F., Heydemann, K., and Berthomé, P. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *Proceedings of the19th European Symposium on Research in Computer Security* (2014), Springer-Verlag New York, Inc., pp. 200–218.

[124] Larus, J. R. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (New York, New York, USA, 1999), PLDI '99, ACM, pp. 259–269.

[125] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California, USA, 2004), IEEE Computer Society, p. 325.

[126] Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snavely, A. PEBIL: Efficient static binary instrumentation for Linux. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software* (Mar. 2010), pp. 175–183.

[127] Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snavely, A. PEBIL: Efficient static binary instrumentation for Linux. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software* (Mar. 2010), IEEE, pp. 175–183.

[128] Lauri, N. Study argues against maintaining legacy systems, Jul. 2013, Retrieved 2016-05-09 from https://fcw.com/Articles/2013/07/23/meritalk-IT-survey.aspx.

[129] Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W. From Zygote to Morula: Fortifying weakened ASLR on Android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (May 2014), IEEE, pp. 424–439.

[130] Leverett, B. W., and Szymanski, T. G. Chaining span-dependent jump instructions. *ACM Transactions on Programming Languages and Systems 2*, 3 (Jul. 1980), 274–289.

[131] Loftus, P. Hacking Is a Risk for Pacemakers. So Is the Fix, 2017, Retrieved from https://www.wsj.com/articles/hacking-is-a-risk-for-pacemakers-so-is-the-fix-1508491802.

[132] Lu, K., Nürnberger, S., Backes, M., and Lee, W. How to make ASLR win the clone wars: Runtime re-randomization. In *Proceedings of the 2016 Network and Distributed System Security Symposium* (San Diego, California, USA, Feb. 2016).

[133] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, New York, USA, 2005), PLDI '05, ACM, pp. 190–200.

[134] Lutz, R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the 1993 IEEE International Symposium on Requirements Engineering*, IEEE Comput. Soc. Press, pp. 126–133.

[135] Mahoney, M. S. Release.0, The Beginning, 1998, Retrieved from http://www.princeton.edu/{~}hos/Mahoney/expotape.htm.

[136] Majlesi-Kupaei, A., Kim, D., Anand, K., ElWazeer, K., and Barua, R. RL-Bin, Robust Low-overhead Binary Rewriter. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation - FEAST '17* (New York, New York, USA, 2017), ACM Press, pp. 17–22.

[137] Marchand, S. Making Chrome on Windows faster with PGO, 2016, Retrieved from https://blog.chromium.org/2016/10/making-chrome-on-windows-faster-with-pgo.html.

[138] Marco-Gisbert, H., and Ripoll, I. Preventing brute force attacks against stack canary protection on networking servers. In *Proceedings of the12th IEEE International Symposium on Network Computing and Applications* (Aug. 2013), IEEE, pp. 243–250.

[139] Marco-Gisbert, H., and Ripoll, I. On the effectiveness of {NX}, {SSP}, {RenewSSP}, and {ASLR} against stack buffer overflows. *Proceedings of the IEEE 12th International Symposium on Network Computing and Applications* (2014), 145–152.

[140] Marshall, A. NTSB Says Tesla Bears Some Blame for Deadly Autopilot Crash — WIRED, 2017, Retrieved 2017-12-10 from https://www.wired.com/story/tesla-ntsb-autopilot-crash-death/.

[141] Martin, R. The Installed Base of Smart Meters Will Surpass 1 Billion by 2022, 2013, Retrieved from http://www.navigantresearch.com/newsroom/the-installed-base-of-smart-meters-will-surpass-1-billion-by-2022.

[142] Mashtizadeh, A. J., Bittau, A., Boneh, D., and Mazières, D. CCFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security* (New York, New York, USA, 2015), ACM Press, pp. 941–951.

[143] Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. System V Application Binary Interface: AMD64 Architecture Processor Supplement. Tech. rep., 2013.

[144] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), USENIX Association, p. 15.

[145] Melski, D. Building an Autonomous Cyber Battle System: Our Experience in DARPA's Cyber Grand Challenge, 2016, Retrieved from https://www.youtube.com/watch?v=xfgGZq86iWk.

[146] Melski, D., and Davidson, J. W. Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. Tech. rep., GrammaTech, Ithaca, New York, USA, 2016.

[147] Mendlson, A., Pinter, S. S., and Shtokhamer, R. Compile time instruction cache optimizations. *ACM SIGARCH Computer Architecture News 22*, 1 (Mar. 1994), 44–51.

[148] Meng, X., and Miller, B. P. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, New York, USA, 2016), ACM Press, pp. 24–35.

[149] Merrit, R. IBM tells story behind Chevy Volt design. *EE Times* (May 2011).

[150] Merten, M. C., Trick, A. R., Nystrom, E. M., Barnes, R. D., Hmu, W.-m. W., Merten, M. C., Trick, A. R., Nystrom, E. M., Barnes, R. D., and Hmu, W.-m. W. A hardware mechanism for dynamic extraction and relayout of program hot spots. *ACM SIGARCH Computer Architecture News 28*, 2 (May 2000), 59–70.

[151] Miller, D. C., and Valsek, C. Remote Exploitation of Unaltered Passenger Vehicle. Tech. rep., 2015.

[152] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., and Franz, M. Opaqe Control-Flow Integrity. In *Proceedings of the 2015 Network and Distributed System Security* (San Diego, California, USA, 2015), Internet Society.

[153] MORAN, R. Bala Cynwyd man sentenced to prison for hacking water utilities, Jun. 2017, Retrieved from http://www.philly.com/philly/news/breaking/man-sentenced-to-prison-for-hacking-water-utilities-20170615.html.

[154] MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Notices 44*, 3 (Feb. 2009), 265.

[155] NECULA, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, New York, USA, 1997), POPL '97, ACM, pp. 106–119.

[156] NECULA, G. C., AND LEE, P. Safe, Untrusted Agents Using Proof-Carrying Code. Springer, Berlin, Heidelberg, 1998, pp. 61–91.

[157] NETCRAFT. July 2016 Web Server Survey, 2016, Retrieved from https://news.netcraft.com/archives/2016/07/19/july-2016-web-server-survey.html.

[158] NETCRAFT. July 2017 Web Server Survey, 2017, Retrieved 2017-09-05 from https://news.netcraft.com/archives/2017/07/20/july-2017-web-server-survey.html.

[159] NIST. Guidelines for Smart Gride Cybersecurity. Tech. rep., National Institutes of Standards and Technology, Gaithersburg, Maryland, USA, 2014.

[160] NORDRUM, A. Autonomous Security Bots Seek and Destroy Software Bugs in DARPA Cyber Grand Challenge - IEEE Spectrum, 2016, Retrieved 2016-09-08 from http://spectrum.ieee.org/tech-talk/telecom/security/autonomous-supercomputers-seek-and-destroy-software-bugs-in-darpa-cyber-grand-challenge.

[161] ORACLE CORPORATION. Linker and Libraries Guide. Oracle Corporation, 2010, ch. 3.

[162] ORACLE CORPORATION. The HotSpot Group, 2016, Retrieved 2016-11-19 from http://openjdk.java.net/groups/hotspot/.

[163] OSBORNE, T. Incorrectly Installed Engine Software Caused A400M Crash, Airbus Official Says, May 2015, Retrieved from http://aviationweek.com/defense/incorrectly-installed-engine-software-caused-a400m-crash-airbus-official-says.

[164] O'SULLIVAN, P., ANAND, K., KOTHA, A., SMITHSON, M., BARUA, R., AND KEROMYTIS, A. D. Retrofitting Security in COTS Software with Binary Rewriting. Springer Berlin Heidelberg, 2011, pp. 154–172.

[165] OVERLY, S. Twitter increases ad transparency as Russia investigation intensifies, Oct. 2017, Retrieved from http://www.politico.com/story/2017/10/24/twitter-advertising-transparency-russia-investigation-244119.

[166] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (May 2012), IEEE, pp. 601–615.

[167] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (May 2012), IEEE, pp. 601–615.

[168] PAX TEAM. PaX ASLR Design and Implementation, 2003, Retrieved 2016-05-10 from https://pax.grsecurity.net/docs/aslr.txt.

[169] PAYER, M., BARRESI, A., AND GROSS, T. R. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2015), Springer-Verlag New York, Inc., pp. 144–164.

[170] Perrin, A., and Duggan, M. Americans' Internet Access: 2000 - 2015. Tech. rep., Pew Research Center, 2015.

[171] Petsios, T., Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. DynaGuard: Armoring canary-based protections against brute-force attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, New York, USA, Dec. 2015), ACSAC 2015, ACM Press, pp. 351–360.

[172] Pettis, K., Hansen, R. C., Pettis, K., and Hansen, R. C. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, New York, USA, 1990), Vol. 25, ACM Press, pp. 16–27.

[173] Pitchford, M. Making Sense of Software of Unknown Pedigree. *Dr. Dobb's Journal* (2009).

[174] Prasad, M., and Chiueh, T.-c. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference* (2003), pp. 211–224.

[175] Qi, Z., Long, F., Achour, S., and Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (New York, New York, USA, 2015), ISSTA 2015, ACM, pp. 24–36.

[176] Qiu, J., Su, X., and Ma, P. Identifying functions in binary code with reverse extended control flow graphs. *Journal of Software: Evolution and Process 27*, 10 (Oct. 2015), 793–820.

[177] Ramsey, N., Dias, J., Ramsey, N., and Dias, J. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, New York, USA, 2011), Vol. 46, ACM Press, p. 575.

[178] Reeves, J., Ramaswamy, A., Locasto, M., Bratus, S., and Smith, S. Intrusion detection for resource-constrained embedded control systems in the power grid. *International Journal of Critical Infrastructure Protection 5*, 2 (Jul. 2012), 74–83.

[179] Richards, M. The portability of the BCPL compiler. *Software: Practice and Experience 1*, 2 (Apr. 1971), 135–146.

[180] Ritchie, D. M. Advice From Doug Mcilroy, Retrieved 2017-01-10 from https://www.bell-labs.com/usr/dmr/www/mdmpipe.html.

[181] Rodes, B. Stack Layout Transformation: Towards Diversity for Securing Binary Programs. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, New Jersey, USA, 2012), ICSE '12, pp. 1543–1546.

[182] Rogers, J. CROMU-00009: RAM-based filesystem, Retrieved 2017-12-08 from https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges/CROMU_00009.

[183] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and Chen, B. Instrumentation and optimization of Win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop* (Seattle, Washington, USA, 1997), USENIX Association, pp. 1–1.

[184] Rosen, S. ALTAC, FORTRAN, and compatibility. In *Proceedings of the 16th ACM National Meeting* (New York, New York, USA, Jan. 1961), ACM Press, pp. 22.201–22.204.

[185] Sahai, A. Obfuscation, 2015, Retrieved from https://www.youtube.com/watch?v=7LkTUOvfeno.

[186] Salus, P. H. *A Quarter Century of UNIX*. ACM Press/Addison-Wesley Publishing Co., New York, New York, USA, 1994.

[187] Salwan, J. ROPgadget - Gadgets finder and auto-roper, 2011, Retrieved from http://shell-storm.org/project/ROPgadget/.

[188] Sandborn, P. A., and Prabhakar, V. J. The forecasting and impact of the loss of critical human dkills necessary for supporting legacy systems. *IEEE Transactions on Engineering Management 62*, 3 (Aug. 2015), 361–371.

[189] Santa Cruz Operation. SYSTEM V APPLICATION BINARY INTERFACE. Tech. rep., Santa Cruz Operation, Inc., Santa Cruz, California, USA, 1996.

[190] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (May 2015), IEEE, pp. 745–762.

[191] Schwarz, B., Debray, S., and Andrews, G. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering* (2002), IEEE Comput. Soc, pp. 45–54.

[192] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (2003), IEEE Computer Society, pp. 36–47.

[193] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007).

[194] Shacham, H., and Hovav. The geometry of innocent flesh on the bone. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, New York, USA, 2007), ACM Press, p. 552.

[195] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, New York, USA, Oct. 2004), ACM Press, p. 298.

[196] Shanley, T., and MindShare, I. *Pentium Pro and Pentium II System Architecture*, 2nd ed ed. Addison-Wesley, Reading, Massachusetts, USA, 1998.

[197] Shen, B.-Y., Chen, J.-Y., Hsu, W.-C., and Yang, W. LLBT: An LLVM-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, New York, USA, 2012), ACM Press, p. 51.

[198] Shuaib, K., Trabelsi, Z., Abed-Hafez, M., Gaouda, A., and Alahmad, M. Resiliency of smart power meters to common security attacks. *Procedia Computer Science 52* (2015), 145–152.

[199] Siegl, P., Buchty, R., and Berekovic, M. Data-centric computing frontiers. In *Proceedings of the 2nd International Symposium on Memory Systems* (New York, New York, USA, 2016), ACM Press, pp. 295–308.

[200] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, 8th ed. ed. Wiley, Hoboken, New Jersey, USA, 2009.

[201] Skadron, K., Ahuja, P. S., Martonosi, M., and Clark, D. W. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA, 1998), IEEE Computer Society Press, p. 321.

[202] Smith, A. U.S. Smartphone Use in 2015. Tech. rep., Pew Research Center, 2015.

[203] Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (New York, New York, USA, 2015), ESEC/FSE 2015, ACM, pp. 532–543.

[204] Smithson, M., Anand, K., Kotha, A., Elwazeer, K., Giles, N., and Barua, R. Binary Rewriting without Relocation Information. Tech. rep., University of Maryland, 2010.

[205] Smithson, M., ElWazeer, K., Anand, K., Kotha, A., and Barua, R. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Proceedings of the 20th Working Conference on Reverse Engineering* (Oct. 2013), IEEE, pp. 52–61.

[206] Solar Designer. Bugtraq: Getting around non-executable stack (and fix), 1997, Retrieved from http://seclists.org/bugtraq/1997/Aug/63.

[207] Solar Designer. lpr LIBC RETURN exploit, 1997, Retrieved from http://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[208] SPEC. SPEC CPU2006: Read Me First, 2011, Retrieved 2016-05-10 from https://www.spec.org/cpu2006/Docs/readme1st.html.

[209] Srikant, Y. N., and Shankar, P. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, 2nd ed. ed. CRC Press, Boca Raton, Florida, USA, 2008.

[210] Srinivasan, V., Reps, T., Srinivasan, V., and Reps, T. Partial evaluation of machine code. *ACM SIGPLAN Notices 50*, 10 (Oct. 2015), 860–879.

[211] Srinivasan, V., Reps, T., Srinivasan, V., and Reps, T. An improved algorithm for slicing machine code. In *Proceedings of the 2016 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, New York, USA, 2016), Vol. 51, ACM Press, pp. 378–393.

[212] Srivastava, A., Eustace, A., Srivastava, A., and Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (New York, New York, USA, 1994), Vol. 29, ACM Press, pp. 196–205.

[213] Srivastava, A., and Wall, D. W. A Practical System for Intermodule Code Optimization at Link-Time. Tech. rep., Digital Western Research Laboratory, 1992.

[214] Standard Performance Evaluation Corporation. SPEC CPU® 2006, 2017, Retrieved 2017-07-12 from https://www.spec.org/cpu2006/.

[215] Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T. The problem of programming communication with changing machines: A proposed solution. *Communications of the ACM 1*, 8 (Aug. 1958), 12–18.

[216] Szekeres, L., Payer, M., Tao Wei, and Song, D. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (May 2013), IEEE, pp. 48–62.

[217] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems: Design and Implementation*, 3rd ed ed. Pearson Prentice Hall, Upper Saddle River, New Jersey, USA, 2006.

[218] Tanner, P. Software Portability: Still an Open Issue? *StandardView 4*, 2 (Jun. 1996), 88–93.

[219] The LLVM Admin Team. The LLVM Compiler Infrastructure Project, Retrieved 2017-06-25 from http://llvm.org/.

[220] Thomson, I. It took DEF CON hackers minutes to pwn these US voting machines, Jul. 2017, Retrieved from http://www.theregister.co.uk/2017/07/29/us_voting_machines_hacking/.

[221] Tomiyama, H., and Yasuura, H. Code placement techniques for cache miss rate reduction. *ACM Transactions on Design Automation of Electronic Systems 2*, 4 (Oct. 1997), 410–429.

[222] Trail of Bits. McSema, 2014, Retrieved 2016-11-28 from https://github.com/trailofbits/mcsema.

[223] Twyford, K. Transcript of the Morning Trial Proceedings Had on the 14th Day of October, 2013 Before the Honorable Patricia G. Parrish, District Judge, 2013.

[224] Valensi, C. MADRAS: Multi-architecture binary rewriting tool technical report.

[225] Van Put, L., Chanet, D., De Bus, B., De Sutter, B., and De Bosschere, K. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology* (2005), IEEE, pp. 7–12.

[226] Vandierendonck, H., and Seznec, A. Speculative return address stack management revisited. *ACM Transactions on Architecture and Code Optimization 5*, 3 (Nov. 2008), 1–20.

[227] Veracode. State of Software Security 2017. Tech. rep., Veracode, 2017.

[228] Wagle, P., and Cowan, C. Stackguard: Simple stack smash protection for {GCC}. In *Proceedings of the GCC Developers Summit* (2003), pp. 243–255.

[229] Wahbe, R., Lucco, S., Anderson, T. E., Graham, S. L., Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (New York, New York, USA, 1993), Vol. 27, ACM Press, pp. 203–216.

[230] Walker, M. Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge, 2015, Retrieved from https://www.usenix.org/node/190798.

[231] Wang, M., Yin, H., Bhaskar, A. V., Su, P., and Feng, D. Binary Code Continent. In *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015* (New York, New York, USA, 2015), ACM Press, pp. 331–340.

[232] Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., and Vigna, G. Ramblr: Making Reassembly Great Again. In *Proceedings of the Network and Distributed System Security Symposium* (2017).

[233] Wang, S., Wang, P., and Wu, D. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium* (Washington, DC, USA, 2015), USENIX Association, pp. 627–642.

[234] Wang, S., Wang, P., and Wu, D. UROBOROS: Instrumenting stripped binaries with static reassembling. In *Proceedings of the 2016 IEEE International Conference on Software Analysis, Evolution, and Reengineering* (Mar. 2016), IEEE, pp. 236–247.

[235] Wang, W., Xu, X., and Hamlen, K. W. Object Flow Integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, New York, USA, 2017), ACM Press, pp. 1909–1924.

[236] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, New York, USA, Oct. 2012), ACM Press, p. 157.

[237] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, New York, USA, 2012), ACM Press, p. 299.

[238] Wartell, R., Zhou, Y., Hamlen, K. W., and Kantarcioglu, M. Shingled Graph Disassembly: Finding the Undecideable Path. Springer, Cham, 2014, pp. 273–285.

[239] Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., and Thuraisingham, B. Differentiating Code from Data in x86 Binaries. Springer Berlin Heidelberg, 2011, pp. 522–536.

[240] Weimer, W., and Necula, G. C. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems 30*, 2 (Mar. 2008), 8:1—-8:51.

[241] Wika, K., and Knight, J. *A Safety Kernel Architecture*. PhD thesis, University of Virginia, 1994.

[242] Wika, K. G. *Safety Kernel Enforcement of Software Safety Policies*. PhD thesis, University of Virginia, Charlottesville, Virginia, USA, 1995.

[243] Williams-King, D., Gobieski, G., Williams-King, K., Blake, J. P., Yuan, X., Colp, P., Zheng, M., Kemerlis, V. P., Yang, J., and Aiello, W. Shuffler: Fast and deployable continuous code re-randomization.

[244] Witchel, E. Programmer productivity in a world of mushy interfaces. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, New York, USA, Mar. 2016), ACM Press, pp. 591–591.

[245] Wulf, W. A., and McKee, S. A. Hitting the memory wall. *ACM SIGARCH Computer Architecture News 23*, 1 (Mar. 1995), 20–24.

[246] Xu, H., and Chapin, S. J. Improving address space randomization with a dynamic offset randomization technique. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (New York, New York, USA, 2006), ACM Press, p. 384.

[247] Yardimci, E., and Franz, M. Mostly static program partitioning of binary executables. *ACM Transactions on Programming Languages and Systems 31*, 5 (Jun. 2009), 1–46.

[248] Zabrocki, A. Scraps of notes on remote stack overflow exploitation, Nov. 2010, Retrieved from http://phrack.org/issues/67/13.html.

[249] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (May 2013), pp. 559–573.

[250] Zhang, M., Qiao, R., Hasabnis, N., Sekar, R., Zhang, M., Qiao, R., Hasabnis, N., and Sekar, R. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, New York, USA, Sep. 2014), Vol. 49 of *VEE '14*, ACM, pp. 129–140.

[251] Zhang, M., and Sekar, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, DC, USA, 2013), USENIX, pp. 337–352.

[252] Ziener, D., and Teich, J. Concepts for run-time and error-resilient control flow checking of embedded RISC CPUs. *International Journal of Autonomous and Adaptive Communications Systems 2*, 3 (2009), 256.