Persistent Storage for Program Metadata

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

Daniel W. Williams

May 2012

APPROVAL SHEET

is submitted in partial fulfillment of the requirements

for the degree of

[illegible]

AUTHOR

[illegible]

Advisor

[illegible]

[illegible]

[illegible]

David Evans BP

Erik Altman BP

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean, School of Engineering and Applied Science

Abstract

Building a modern software system has increasingly become a complex and difficult to manage task. Typical software development systems involve dozens of tools to assist the programmer in building secure and robust programs. Many of these tools perform detailed analysis, collecting large amounts of program metadata to better understand the program and improve it. Unfortunately this metadata is often discarded immediately after the tool is finished running. By saving, organizing and making such metadata available across the software development toolchain, software developers can build new tools and improve existing tools with ease. To achieve this goal, this work presents Metaman, a system for metadata storage and retrieval. Metaman allows any tool in the toolchain to submit and query metadata about the program, avoiding duplication of analysis and saving data that was previously discarded.

An important class of tools with specific metadata needs is the set of run-time tools. Many high-level programing environments offer robust built-in run-time systems to assist the programmer with features such as security models and run-time introspection. For applications created using languages without such features, Software Dynamic Translation (SDT) systems have been introduced to allow these features to be applied to arbitrary programs. However, because these SDT-based tools operate at runtime, lengthy analysis phases can negatively effect application performance. To validate the value and utility of maintaining program metadata across the software development toolchain, the research demonstrates how ubiquitous program metadata can be used to provide the ability for SDT systems to improve the performance, security, and program understanding without the need to do costly analysis at runtime. These improvements show the benefits of ubiquitous availability of metadata for SDT systems as well as other tools in the development toolchain.

Acknowledgments

There are too many people who deserve thanks for their invaluable help throughout this process, so this is not a comprehensive list. First and foremost, I would like to thank my committee, Jack Davidson, John Knight, Dave Evans, Mary Lou Soffa, Eric Altman, and Bill Scherer.

Friends and family have been invaluable to me throughout this process: Mom, Dad, Jennifer, Rachel, Erik, Spiegel, and Ross.

Finally, I'd especially like to thank (again) Jack Davidson and members of his research group for their insights and assistance, particularly: Jason Hiser, Michele Co, Sudeep Ghosh, Wei Hu, Clark Coleman, Mark Bailey, Joy Kamunyori, and Julie Parent.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1952, Grace Hopper wrote the first automatic compiler, for the Arithmetic Language, version 0 (A-0), to make it easier to abstract algorithmic programming across the specifics of hardware [63]. Hopper describes A-0 as a system to abstract away the details of programming, moving more of the work from the programmer to the computer. A-0's tools included pre-compiled subroutines that could be "translated and placed" into the program. The symbolic subroutine resolution and lookup offloaded the work of manual table lookups used by programmers. These tools are the basis for the underlying components of modern software development, specifically linkers and assemblers. Later in 1957, John Backus and his team developed the FORTRAN compiler, whose syntax was wholly disconnected from the syntax of the underlying machine language [49]. The "*FOR*mula *TRAN*slator" allowed the programmer to write purely mathematical and logical statements, and have them translated into the specific machine language independently. These linkers and compilers created a basic set of tools allowing programmers to be more productive.

By the 1970s, the UNIX$^{\text{TM}}$operating system was developed along with the C programming language. As compilers and related software tools progressed, the properties and requirements of a compilation system have been formalized. However, the fundamental goal has remained: to abstract the details of the specific hardware while maintaining a high-level of performance, security, and reliability. These early software development tools paved the way for general-purpose programming and the field of software engineering. Hopper's original goal of moving the repetitive work of programming from the programmer to the computer has resulted in a robust set of tools for software development, and as a result, more complex and valuable software.

The robust base of UNIX^{TM}and UNIX-like tools has given rise to a rich software ecosystem. Ever-advancing hardware and software tools have allowed programmers to build applications that are more feature-rich and larger than ever before. This ecosystem is being driven by a number of important trends:

- improved language features,
- improved run-time systems,
- shorter release cycles, and
- more advanced hardware.

Improved language features have played a major role in how software is developed. Advanced features such as garbage collection, dynamic code loading [111], reflection [116], and exception handling [111] require more robust components operating at run-time. With a larger set of run-time features, many parts of compilation have been deferred until run-time. Due to byte-compiled languages like Java and C#, software development tools have increasingly been built to interact while the program is running, resulting in Just In Time (JIT) compilation. JIT compilation interprets byte code, and only “hot” code is translated into machine code and optimized.

The next important trend applies to applications written in languages without such heavyweight run-time libraries such as C, C++ and Fortran. These applications can take advantage of run-time based security and performance advancements using software dynamic translation (SDT). SDT systems provide a runtime to native binaries which allows each instruction to be examined before being executed. With this ability, SDT systems are the basis for many tools such as profilers [77, 85, 131], dynamic optimizers [10, 20, 125], and security policy enforcers [65, 71].

Because it has become easier to add new features, and security concerns have made bug patches common, the release cycle of many applications has progressively shortened [14]. The short turn-around time from one version of an application to the next forces developers to automate the development process as much as possible.

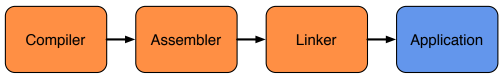


Figure 1.1: Basic Toolchain

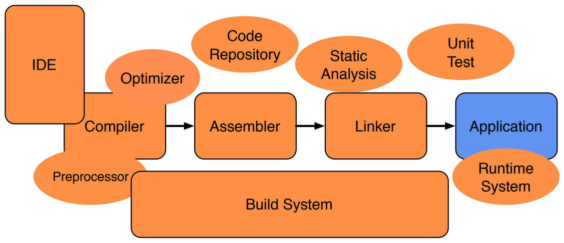


Figure 1.2: Modern Toolchain

Finally, while hardware seems to have reached a "clock-speed" plateau, Moore's law still applies. Moore's law states that the number of transistors on an integrated circuit doubles approximately every two years. To improve hardware performance, new features, rather than increased clock rate is making the hardware/software interface more complex. To support these new features, hardware relies on the software development toolchain. For years, hardware designers have leveraged Moore's Law to increase the amount of work done in a processor per clock cycle, and to increase the number of pipeline stages to allow for shorter clock cycles. This results in higher clock rates. Increasing clock rates reached the point of diminishing returns for overall system performance. Therefore, hardware designers have been forced to redesign processors to incorporate new features to achieve improved performance. Many of these features, including multicore, instruction bundling [42, 79], and register windows [68], require the support of the software development toolchain. For applications to take advantage of these hardware features, the toolchain must emit code that utilizes them.

Software developers have been able to meet the challenges of those trends due in large part to the tools that have helped them manage and understand the software they write. The de facto standard in building software continues to be based in the UNIX tradition of a series of stand-alone programs. Each of these programs takes a well-defined file as input, and outputs another file in the format expected by the next tool. Figure 1.1 shows the traditional UNIX software development toolchain. The process starts with the programmer's source code, which is lexically analyzed and parsed by the compiler, which then emits the resulting assembly code. The assembler takes the assembly file and then generates an object file that contains the actual binary instructions. Finally, the linker combines the object files with the necessary libraries to create the executable. The executable can then be run in conjunction with the run-time system.

There are many variations on this basic theme. For example, some languages require a pre-processor, others defer linking until the program is loaded into memory just before execution. The average programmer does not need to be exposed to the the specific steps necessary to translate their source program to a binary. To simplify the process, the majority of developer tools such as the GNU Compiler Collection, `gcc`, will invoke all of the programs listed above with a single command, with intermediate files created and then deleted by `gcc` itself. All of these tools are highly modular. They each work on an input and output format, and for the most part, do not interact with the other tools—instead they ensure compatibility by conforming to the output file format.

Programming language and compilation researchers continue to build small, stand-alone tools to improve individual aspects of the program. As a result, most software developers use a large number of mostly-independent tools which result in a powerful, yet disorganized toolchain. Figure 1.2 shows a more current landscape of software development. Many tools interact to build the software. However, the flow of building an application is obscured by the large set of tools required to properly build the application. Optimizers are linked in to the compiler to improve performance without altering semantics. Software engineering tools such as code repositories [29] and unit tests are included to track how the software changes over time and ensure any such change does not introduce new bugs. Static analysis tools have been added to the toolchain to ensure statically verifiable properties of the code, such as basic lock correctness and simple memory management properties [12, 56].

The set of necessary tools has become sufficiently complex that working with each tool individually has become difficult. To address this problem, integrated development environments (IDEs) have created a unified interface to the set of software development tools. Modern IDEs such as Eclipse [122] and Visual Studio [67] offer programs a wide set of tools to improve, understand, alter, document, and debug their code. Some of the tools integrated into the IDE are inherent to the particular IDE. Other functionality can be added with “plug-ins”—dynamically loaded code that runs within the IDE to provide additional functionality. Finally, some of the tools are launched by the IDE as sub-processes, such as external visualization software or static analysis tools. The IDE offers programmers a single interface to their tools, making it easier for programmers to understand and navigate large projects. While IDEs provide a unified view of the tools being used, for the most part they do not offer a unified system for handling a vitally important resource: Data about the program, also known as program metadata.

1.1 Program Metadata

While the UNIX tradition of small programs that perform a single function offers an important benefit of being able to separate concerns and swap out programs for similar ones, it also has a significant disadvantage. Because the tools themselves are independent, they often redo work already performed by a previous tool. Many tools operate by doing some analysis on the program—gathering data about the program (*metadata*) by creating an abstraction of the program to eliminate the unnecessary details for that tool. The tool then performs some action using the metadata to either improve the program or document a feature of the program. To save the information gathered by the tool, either a change must be made to the output format to store the information or an otherwise innocuous annotation must be added. Overcoming such hurdles can be difficult. Debugging information is currently the most widely used metadata for programs, and early debugging research focused more on LISP where a program interpreter was available [40]. To handle such a problem in compiled languages debugging formats such as DWARF [35] were introduced. When a compiler uses DWARF, it emits an annotation into the assembly, which the linker loads into a separate section. Any other tool subsequently altering the code, or doing analysis with the debugging information must parse and correct all the debugging information, not just the relevant information. Such a barrier to use makes adding information to the debugging specification a rare occurrence.

However, many tools collect their own metadata, and many other tools have use for metadata unavailable to them. For example, information about the control flow of the program – the control flow graph (CFG) – is a common piece of information used by many tools. The CFG is initially collected by the front-end compiler to lay out the basic blocks of the program, and later used by the optimizer to help with program analysis [3]. However it is also used by many other tools including link-time optimizers [33], security policy enforcers [1]. It is valuable to many run-time systems, though they often do not have access to it due to the code discovery problem. The fact that static analysis cannot fully determine the targets of all branches, makes it impossible to fully determine what is code in a binary. The tools that do have access to the full CFG often recalculate it themselves after it has already be calculated by the compiler. Such unnecessary duplication of work is an important hindrance to advanced tool development.

1.2 Thesis

The pressure for more features, better performance, and more reliability in new applications, combined with the need for new hardware support has greatly increased the complexity of the software development toolchain. Additional tools are continually being added to the toolchain, and existing tools are also constantly upgraded. As a result, fully understanding the changes being made by individual tools and the effect of those changes have on the final program is difficult. Many of these programs collect and maintain metadata about the program and use this metadata to improve or analyze the program. Often that metadata is used and then discarded, even though another tool might be able to use the data as well.

The thesis of this dissertation is that the comprehensive collection and organization of program metadata across the software development toolchain can improve the software development process as well as the resulting applications.

Program metadata is used by almost every tool in the software development toolchain. By collecting and organizing program metadata, new tools can be built and integrated into the toolchain more quickly and effectively, resulting in new features and more robust software becoming available. A metadata manager is a new software development tool for overcoming the problem of disparate metadata. *Metaman*, the *Meta*data *man*ager is the prototype implementation of a metadata manager useful for collecting and organizing metadata. Metaman was designed, implemented, integrated with of software tools to aide in evaluating this thesis.

1.3 Research Overview

The specific research contributions of this dissertation are the direct consequence of evaluating the thesis. This work examines both the design and structure of implementing comprehensive metadata management, as well as the tools that can be enabled by the increased availability of program metadata. There are many vital design decisions to consider to effectively handle program metadata. General data storage techniques is a wide area of research, however, program metadata has special requirements and restrictions that make it a unique area of study. Because program metadata must be kept up-to-date when changes to the program occur, the metadata system must be able to synchronize with the program. Additionally, a wide range of metadata is possible. Metaman was created to explore the design space of metadata systems.

To demonstrate the positive effect of persistent metadata on the software development process, Metaman is evaluated with software dynamic translation (SDT) systems, a tool category which historically has little access to such metadata. Because SDT systems are designed to work on raw binaries, most assume no additional information about the binary. This dissertation evaluates the inclusion of metadata in SDT systems to improve the areas of optimization and overhead reduction, as well and program understanding and security. These areas are active research avenues for SDT systems, and the areas each have numerous specific needs for program metadata.

SDT is a technique for examining and potentially modifying every binary instruction before it executes. SDT systems operate at the application level, above the OS. When the OS starts execution of a new process image (i.e., via the `exec` system call on POSIX systems), the SDT takes control of the process before any application code is run. As shown in Figure 1.3, once the SDT system has control of the process, it examines the PC value and begins translation. Translation can be as simple as a copying the current instruction without modification, complete replacement of the instruction with new instructions or a function, or replacement of sequences of instructions with

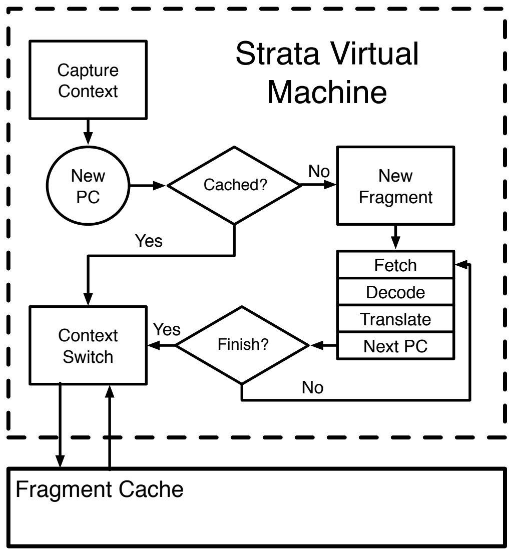


Figure 1.3: Software Dynamic Translation

new sequences. The translated instruction(s) are then placed in a code cache for the translated code, called a *fragment cache*. Translating each instruction before it is first executed allows the SDT system to modify or instrument any instruction, and therefore it is a powerful and flexible tool for improving programs in changing environments.

SDT systems are a large and vibrant research area (discussed in more detail in Chapter 2). However, for the purpose of evaluating metadata management and the value of program metadata, this work focuses on the following areas: overhead reduction, optimization, security and program understanding. The goal of overhead reduction is to reduce or eliminate SDT overhead as much as possible to allow SDT systems to run at near-native speeds. Similarly, the goal of optimization within SDT systems is to improve the already existing binary code and secondary instrumentation to potentially improve performance beyond native speeds.

These properties make SDT systems ideal candidates for evaluating the inclusion of program metadata. Because they operate on native binaries, there is a large need for additional information about the program, and many opportunities for improvement if such information is available. Further, because SDT systems operate at run-time, they are particularly sensitive to performance and scaling problems presented by the use of comprehensive program metadata. This thesis presents a variety of uses for run-time metadata by SDT systems, including a novel approach to indirect branch handling using high-level programming constructs, and a novel run-time overrun detection system using debugging information to refine the identification of stack variables.

This dissertation makes the following research contributions:

- A taxonomy of program metadata sources and representations.
- Structure and design of a persistent metadata storage system.
- Analysis of the implementation of a prototype of a persistent metadata storage system.
- A unified XML format for persistent metadata.
- A technique for indirect branch handling for SDT systems using metadata from virtual functions, switches, and returns.
- Implementation and evaluation of improved indirect branch handling.
- A novel algorithm for detecting buffer overruns using an SDT system and debugging metadata.
- Implementation and evaluation of the buffer overrun technique.
- Integration of novel metadata into an existing tool.

The next section describes how these contributions are presented in the subsequent chapters.

1.4 Organization

Chapter 2 presents the background of current software development practices and discussion of systems using program metadata. Compiler tools, build tools, and run-time tools are discussed along with their current uses of program metadata and other information required to run. Existing attempts to integrate metadata into systems are also examined. Holistic systems such as Jalapeño [7], Oberon [74], and various IDEs offer important pieces of the solution, and this chapter examines their strengths and weaknesses. Finally, a detailed overview of SDT systems that are used as the basis for analysis in later chapters is included.

Chapter 3 presents the design of a metadata manager, the solution to the problem of incomplete, unshared, and disorganized metadata. Program metadata includes a wide array of different information and representations. Chapter 3 analyzes the state of program metadata representations and considers the trade-offs for creating a unified representation. Using that analysis, the design of the first metadata manager prototype is also presented: Metaman. The structure of Metaman followed the primary design principles for effective metadata management: 1) flexibility, 2) scalability, and 3) ease of use.

Persistent metadata storage needs to be flexible to effectively support new metadata, new tools, and a changing development environment. Scalability is an important design concern because each tool can generate a large amount of metadata, and when other tools wish to use metadata that is already collected, search and retrieval time for that metadata is an important factor. Finally, ease of use is important to aid adoption of the system. If there is a steep learning curve or many prerequisites, system builders will not want to invest in learning the system.

Metaman uses state-of-the-art tools to organize and make metadata available to tools throughout the software development toolchain. With Metaman, system builders can create new and advanced tools quickly and easily, and such tools do not need to reproduce work already done earlier in the toolchain. Chapter 3 also discusses the details of program metadata: its sources, how it is used, and how it is represented.

Chapters 4 and 5 describe tools built using the facilities provided by Metaman. These tools show the value of Metaman in real-world situations by applying it to problems related to Software Dynamic Translation (SDT). SDT is becoming widely used as a technique for addressing concerns such as performance, adaptability, and robustness. Because SDT systems typically operate on arbitrary binaries, they often do not have access to any program metadata. Because of their usefulness and stringent performance requirements, SDT systems provide an ideal testbed for evaluating the effectiveness of program metadata storage and access techniques.

Chapter 4 focuses on run-time performance and overhead-reduction in SDT systems. SDT systems offer many potential benefits, however those benefits can only be realized if the SDT system does not slow the application down too much. The early Dynamo system focused on dynamic optimization [10], using the SDT system to improve performance of binaries beyond what was provided by static compilation techniques. Subsequent SDT systems have focused less on optimization, but still require low overhead to make their tools practical. By applying knowledge about the origins of indirect branches that occur in an application, the SDT system is able to use metadata to improve indirect branch handling by identifying the possible targets of the high-level structures (i.e., virtual function tables (VFTs) and switch tables) that caused the indirect branch.

Chapter 5 examines the value of Metaman in the security and program understanding context. SDT systems are a powerful tool for improving security and for enabling better program understanding. They are also used for a variety of security and profiling purposes. Chapter 5 illustrates how an SDT system can be augmented with debugging information to help identify a common programming error that often leads to exploitable vulnerabilities—buffer overruns. The Buffer Overrun Detection Engine (BODE), a tool for identifying buffer overruns at run-time, is presented. The chapter also highlights Metaman's usability. The novel stack layout refinement of BODE was ported to another, more comprehensive memory protection tool, MEDS [59]. MEDS operates on arbitrary binaries identifying pointers and ensuring that pointer arithmetic is valid. Using the flexibility of Metaman, MEDS uses BODE metadata to improve its stack analysis, catching illegal accesses that were previously unidentified.

Chapter 6 discusses the future directions of Metaman, and well as the potential of a fully integrated metadata system. There are many possible future uses of a modular, comprehensive metadata repository such as Metaman. The final chapter discusses using Metaman as part of a holistic security mechanism, and also discusses the possibility for using Metaman as the cornerstone of a completely new method of building software that will be more modular and more flexible that traditional file-based development. Finally, Chapter 7 summarizes and concludes the work.

Chapter 2

Related Work

This chapter examines the current uses and storage of program metadata in the software development process. Program metadata has been available in various forms since the beginning of formalized software development. It is used to improve program execution, better understand software artifacts, and aid in program debugging. This chapter describes the evolution of program metadata, its use in tools such as build systems, code repositories, just-in-time (JIT) translators, Virtual Machines (VMs), and Integrated Development Environments (IDEs).

Because the use of metadata in run-time systems provides a primary motivation and evaluation framework for this research, the chapter provides an overview of various run-time systems, such as software dynamic translation (SDT) and VM JITs along with the current metadata capabilities of these systems. Chapters 4 and 5 describe specific SDT-based tools, and related work for those tools is included in those chapters.

2.1 Current Metadata

Early software development systems focused primarily on generated code (the primary data), rather than the abstractions used to analyze and manipulate the code (the metadata). Most uses of persistent program metadata have been directed at helping the programmer understand the program and its behavior. Debugging information is a popular example of persistent metadata. Debugging information is typically stored as annotations in intermediate files or compiled into an isolated segment of an executable. This information allows the debugger to relate the behavior of the program back to the source code. As the complexity of software applications has grown, IDEs began to use metadata to help the programmer manage and understand very large software projects [24, 46]. In these systems, metadata is maintained internally to give immediate feedback about the program to the programmer.

As software development environments evolved, all of the software development processes—editing, management of external resources, debugging, revision control, testing, etc.—were integrated into a single environment. These systems make persistent metadata available as part of their whole-system design.

2.1.1 Debuggers and Debugging Information

The most common type of program metadata currently in use is debugging information. Often, during the software development process, debugging information is included in the binary. This information allows debuggers to present information to programmers in terms of the source code. For example, program addresses are mapped to source code lines. The debugger can use that mapping to insert breakpoints and step through code line-by-line. Debuggers also use the debugging information to decode program data. Raw data in memory can be associated with specific types (strings, integers, float), and properly displayed to the programmer. All of the debugging information is encoded into a debugging format which is either included as part of the binary, or contained in a separate file.

One popular debugging format is DWARF [35]. DWARF is a binary format that encodes a large amount of valuable information about a program. DWARF information is generated by many compilers including `gcc`, IBM's `XLC` [66], and SUN Studio compilers [100], typically with compilation flag `-g`. In object code, DWARF represents debugging information in a tree structure with "debugging information entries" at each node of the tree. The top of the tree splits the sub-trees into individual "compilation units" that typically correspond to individual source code files. Each compilation unit stores information on the types, variables and functions that are defined in that file, as well as a mapping of source code line numbers to binary addresses. This data allows symbolic debuggers, such as `gdb`, to perform introspection on programs and report information directly correlated with specific lines of the source program, making it easier for the programmer to relate program behavior back to the source code. For example, placing a breakpoint on a specific source line, or displaying the value of a field within a `struct` both rely on debugging information.

The other major debugging format is Microsoft's program databases (PDB) format which is generated by Microsoft Corporation's current suite of compilers. The PDB format is proprietary and undocumented, but it can be accessed through Microsoft's debugging API [94]. However, there are several unofficial sources of information about PDB files. Schreiber in his book "Undocumented Windows 2000 Secrets" gives an unofficial structure for the PDB format. Similar to the DWARF format, PDB files combine data from multiple source files. These are PDB "streams" which relate back to individual source files. The streams contain metadata such as symbol and address information used by Microsoft's debugger [106].

While debugging information is the most widely-used set of program metadata, a number of systems have tried to build more general systems that make use of program metadata. These systems are described in the next section.

2.1.2 Integrated Development Environments and Holistic Systems

As software engineering practices developed and programmers began to use a growing number of software tools, they began integrating the tools into a unified user interface. These Integrated Development Environments (IDEs) have given programmers access to their tools and organized their code to make the software development process more efficient [38].

Smalltalk-80 was one of the first languages to explicitly link the language and the associated programming environment, presenting such a unified user interface. Goldberg and Robson describe the then-novel graphical environment as containing "one or more rectangular areas called *views*." [46] (These views are what modern GUI designers call "windows.") These features are the basis of modern IDEs. IDEs have been the basis for many of the early holistic systems, which make use of metadata to improve programs. Smalltalk and other early tools such at the TEAM project aimed to link tools together to eliminate barriers to tools interacting [24]. For example, the TEAM tool enabled software tests and analysis routines to run automatically after compilation, eliminating the need for the programmer to run them explicitly. These tools paved the way for modern iterations such as Eclipse and Visual Studio. Modern IDEs such as these provide an array of features to the programmer, such as interactive diagrams of data structure relationships like class hierarchies and tools visualizing graphical user interfaces.

IDEs can be seen as the initial attempt to unify various metadata (which will be discussed in Section 2.1.2). Many of the standalone tools used in a UNIX-like environment are built in to the IDE, allowing it to share much of the metadata. However, these tools create a difficult choice for the developer of new software tools: they can either build the tool for the IDE specifically, in which case it will only be useful to users of that IDE, or they can build a standalone tool and not take advantage of the information sharing inherent to the IDE.

For example, the Montana project, which became VisualAge for C++ [88], is an IDE that provides metadata throughout the development process. The Montana project takes a holistic approach to building systems where there is a high level of metadata available [69,118]. Montana provides an end-to-end framework for adding custom plug-ins to collect and use program metadata. Montana allowed for three different types of extensions: observer extensions, incorporation extensions, and dependency graph extensions. Observer extensions enable programmers direct access to program metadata providing facilities for collecting relevant data, storing it, and presenting it to the user. Incorporation extensions allow the plug-in to insert itself directly into one of the compilation phases, adding new functionality and using the metadata already computed by previous phases. Finally, dependency extensions allow the plug-in to create new dependencies, thus ensuring custom resources and metadata are available when they are needed.

Other holistic systems focus less on the interface presented to the programmer, and more on continuous program modification and optimization. System designers wanted to utilize metadata not only for debugging purposes, but to also improve the program. Oberon was one of the first systems to completely integrate the software development toolchain and allow program metadata to be accessible throughout the software development process [74]. Oberon focused on profiling metadata, along with phase detection to choose what portions of the code to optimize, and when to reconsider optimization choices. Further, they used the information gather at run-time to improve data locality by dynamically choosing to cluster related memory accesses [73].

Similarly, Jalapeño[1] uses monitoring and recompilation, and provides an API to allow programmers to improve the JVM and create novel optimizations. [7]. Because the JVM uses expressive Java byte code as its input format, Jalapeño has access to metadata encoded in the byte code about the program during recompilation. Later work improved on this by storing profiling information offline to be used to improve decisions make in later executions [8].

It is also common for tools to collect and store metadata purely for their own use. For example, control flow integrity ensures that a program adheres to the control flow specified in the source code [1]. It achieves this effect using a combination of static analysis and dynamic checking, inserting ad-hoc metadata to identify control flow points that cannot be statically verified. Instruction set randomization (ISR) also collects link-time metadata to correctly pad basic blocks so encryption and decryption on the program text can be done correctly. Metadata for ISR is stored in the binary, then used by the software dynamic translator, Strata, when decrypting the text [65]. Dyninst provides an API for altering programs dynamically [21]. The Dyninst API provides an avenue for a large amount of metadata that can be collected by attached tools. These tools illustrate the large potential of a holistic metadata system.

Some projects have proposed frameworks which can handle multiple types of metadata, with varying degrees of generality. The LENS project collects compiler transformations with the goal of better program understanding. By observing which optimizations directly contribute to the generated code, developers can better understand how effective the optimizations are [90].

Similarly, Xu *et al.* embed metadata with the goal of improving dynamic binary translation [130]. They used IA-32 EL, a dynamic binary translation system which converts IA32 (x86) code into IA-64 (Itanium). Because x86 has a very small number of general purpose registers relative to Itanium, it was advantageous to re-do register allocation, because the Itanium required fewer register spills. Generally moving memory accesses into registers is unsafe due to the possibility of aliasing. However, with explicit annotations indicating which memory accesses were register spills

[1]now JikesRVM

and not aliases, these memory accesses could be safely moved into registers. Xu *et al.* collected the original register allocation information and stored it in an ELF section, where the dynamic binary translator could access it later.

All of these projects gather information about the executable, and store them along with the binary. These projects make a strong case for the need an organized system of program metadata, allowing tool builders to register, update and query metadata throughout the development process. The next Chapter introduces Metaman, a *meta*data *man*ager, developed to improve software development by making software metadata ubiquitous and easily available.

Metadata is also quite valuable to the reverse engineering community. Holt *et al.* presented GLX, a format for representing graph information. For example, GLX is used to represent abstract syntax trees (intermediate-level information) and module dependence relationships (high-level information). GLX combines previously created formats such as typed graphs (TGraph) [36] and PROGRES [107] into a typed XML-based language. The older formats represented specific methods for encoding graph-based relations between code objects such as basic blocks (CFGs) and functions (call graphs). These data languages are used by a variety of tools including parsers and reverse engineering tool kits. Such formats show a proof-of-concept of how disparate formats can be made to inter-operate [62].

2.2 Run-time systems

A run-time system is the portion of the software development environment that provides functionality while the program is running. This functionality ranges from simple, such as the C runtime which only provides minimal environment setup and memory management, to complex systems that provide a range of services and just-in-time (JIT) compilation. Virtual Machines and JIT translators are common run-time systems that translate or interpret byte-encoded programs into the native hardware instructions, and provide system services such as reflection and memory management. Because typical bytecode languages are abstract and stack-based, much of the metadata is encoded in the bytecode itself.

An important tool that has less access to program metadata is software dynamic translation (SDT) systems. SDT system were initially built to run on arbitrary binaries with no metadata [10]. SDT systems provide a rich basis to explore the usefulness of persistent program metadata. This section examines VMs, JITs, and SDT systems in detail.

2.2.1 Virtual Machines and Just-in-Time Systems

An important advancement for software development is the creation of advanced virtual machines that abstract hardware specifics and allow for the creation of managed languages. Managed languages protect the user from common errors by providing services and features such as memory management in form of garbage collection and run-time type checking [111]. Further, these languages, such as Java, C#, and variants of LISP offer large amounts of introspection, which afford programmers more flexibility to examine and alter the program based on the environment it is running it. Managed languages also typically disallow direct manipulation of memory addresses (pointers), making it easier to ensure the memory safety of applications.

Memory management and memory safety offer important advantages to software developers. However, the cost of these features is that many of the original VMs had unacceptably poor performance compared to similar code targeting native hardware. To address these performance problems, VM writers started including JIT compilers as part of the system. JIT-based systems such as Jalapeño (now JikesRVM) will do a simple compile of the program without optimization and then profile to determine the active code segments. Once a region of code is determined to be hot, the JIT will recompile that region, this time with more aggressive optimizations [7].

2.2.2 Software Dynamic Translation

One of the the first major works in SDT systems was Dynamo, a dynamic optimizer built for PA-RISC at HP Labs by Bala *et al.* [10]. Dynamo's goal was to improve application performance through dynamic optimization—the optimization of a running program. They observed that modern executables often come from third party sources, and as a result the performance of a system is at the mercy of how well those third parties optimized their code. Therefore, the Dynamo system was designed to work on arbitrary binaries, programs with little or no associated program metadata. When Dynamo is applied to raw binaries, it is able to perform state-of-the-art optimizations at run-time [10, 34]. They achieve a great deal of success, improving performance, even on optimized binaries.

Subsequent SDT systems include Pin [85], DynamoRIO [19, 20], HDTrans [119], and Strata [109, 110]. These systems support a variety of platforms and architectures, including x86, MIPS, SPARC, Itanium, and ARM. However, most of these systems have focused on x86, with each of them at least supporting the x86 architecture. The goals of these systems are also diverse, with some focused on instrumentation, others focused on security, and still others focused on performance. All these systems kept the original design goal from the Dynamo project, working on arbitrary binary applications, without the need for source code, object code, or any other metadata. However, the newer SDT systems added another important goal — maintaining control of the application. Because Dynamo was specifically designed to improve run-time performance it would “bail out” if the optimizations were not improving overall run-time performance. Many of the tools built with the later systems require all instructions to be monitored, requiring them to maintain control of the application for the entire run of the program.

Performance of SDT systems is an important concern. To amortize the cost of translating the instructions, dynamically created basic blocks in the fragment cache (or code cache) are reused, and branches that target already-translated code are linked within the fragment cache in a process called fragment linking, reducing the SDT overhead significantly [110]. Further, there is a significant design space to be examined to determine the proper ending condition for the fragment building process. For example, the SDT system can stop every time it reaches any control flow instruction, or it can translate the not taken path, leaving a trampoline to return to the SDT if the branch is taken. To be even more aggressive, it can queue the taken path and translate that fragment as well, before returning to application code. Such aggressive translation schemes allow for fewer context switches into the SDT VM, as the cost of possible translating some code that will never be executed. Hiser *et al.* have determined that generally more aggressive translations schemes improve performance over conservative schemes [58]. Memory usage is another important performance metric. SDT systems can duplicate code either by forming super-blocks which include some already translated code, or by retranslating code that has multiple entry points. Guha *et al.* have examined these trade-offs when using SDT systems in embedded environments [48].

Code cache management is another significant design choice affecting performance. Generally, the code cache is not large enough to hold the entire application code base, especially in cases of code duplication due to dynamic basic block formation and trace creation. As a result, the SDT system must choose how to flush the code cache, either evicting the cache en mas, or using a more fined-grained policy of evicting individual blocks, or groups of blocks. Hazelwood *et al.* showed that an intermediate policy dividing the code cache into "cache units" provides the best performance in terms of (hardware) cache miss rate and link/unlink overhead [51, 52].

Another important performance concern is trace selection. Trace selection is the process of connecting individual dynamic basic blocks into a superblock, a code region with a single entry point and multiple exits. Choosing when to form traces and exactly which basic blocks to combine into a trace is an important performance trade-off. Duesterwald *et al.* found that lightweight profiling to identify hot regions worked well. When a hot region was identified, a trace could be formed with "next executing tail" (NET) – that is, the next time that start of the trace is executed, the system identifies the path taken, and creates a trace with those basic blocks. [34]. Hiniker *et al.* improved on the NET approach by introducing the "Last-Executed Iteration" (LEI) algorithm. LEI provides improved handling of nested loops by adding a small history buffer to better identify trace-ending conditions. [57].

A important source of overhead in SDT systems is the handling of indirect branches [60]. Handling indirect branches is challenging for SDT systems because the target of the branch can change every time the indirect branch instruction is executed. Therefore to ensure the target has been translated by the SDT system, the system must alter the indirect branch to identify the corresponding fragment cache address. Typically this lookup is done with an optimized inline hashtable lookup. Hiser *et al.* have shown that the careful choosing of the parameters of the indirect branch translation scheme can have a significant effect on performance [60].

SDT systems are used for a variety of tools such as dynamic optimization [10], simulators [91, 129], dynamic patching, security policy enforcement [71]. Most of these tools fall broadly into the categories of optimization, security and program understanding. Chapter 4 deals with optimization and overhead reduction, while Chapter 5 deals with program understanding and profiling, in the interest of continuity, a more detailed discussion of related work in these areas is deferred to those chapters.

Chapter 3

Comprehensive Metadata Management

Program metadata—literally data about data—is an important part of almost all software development tools. Most software development tools collect program metadata in isolation, and a significant amount of valuable information is lost from phase to phase. Driven by a desire to maintain modularity, most tools just transiently keep metadata. Saving program metadata and making it available to other tool developers offers many potential benefits such as avoiding duplication of work and promoting development of new tools.

This chapter discusses program metadata. It includes a taxonomy of program metadata in use today. Because of the variety of types of metadata shown in the taxonomy, there are many challenges to effectively storing program metadata and making it available. There is a nearly-unlimited set of data that is potentially relevant. Further, the subset of metadata that is useful can change from application to application and from run to run. This research investigates solutions to the challenge of storing and reusing program metadata. The solution is a new tool for software development which performs comprehensive metadata management. A metadata manager provides the ability to store the wide variety of program metadata, maintain that metadata, and allow efficient access to the metadata by a range of tools. This chapter also describes the structure of the first prototype metadata manager, Metaman, including the design decisions and trade-offs made for the prototype implementation. Finally, this chapter details how software tools need to be altered to integrate with a metadata manager, and how it has been successfully accomplished with Metaman.

3.1 Metadata

Metadata is ubiquitous in modern computing systems. Metadata offers information about how, where, and when a piece of data came into being. For traditional data, metadata provides context about the data, typically to benefit the people using the data. In the context of software development, metadata is any data that describes the structure of program or informs the system about the programmer's intent beyond the strictly necessary program instructions. Program metadata has a variety of uses and consumers. Some metadata is used to improve the programmer's understanding of how their software behaves, such as debugging metadata used by symbolic debuggers to give programs a source-level view of the program while it is executing. Other tools analyze and automatically improve software, such as link-time optimizers. Still other tools operate at run-time, such as security policy enforcers. These tools use metadata to help infer programmer's intent, to better identify when a program is behaving erroneously.

Metadata can vary from simple information like the program's control flow, to very complex data about design and deployment. In the following section, we provide a taxonomy of program metadata.

3.2 Metadata Taxonomy

Table 3.1 provides examples of program metadata, which programs collect the metadata, and which programs can use the metadata. It is important to note the variety of tools that provide and consume metadata. This listing is not meant to be comprehensive; rather it is meant to thoroughly show the different types of metadata that can be stored and queried by Metaman. The encoding of this metadata is described by the Metaman XML schema, included in Appendix A.

The different types of metadata are grouped into four general categories based on the source of the metadata. The categories are compilation, configuration, deployment, and run-time.

In general, compilation metadata comes from the preprocessor, compiler, or related static analyzers and it is linked to the final state of the binary or byte-encoded application. Compilation metadata is usually invalidated by changes to the application or recompilation. Configuration metadata is more generally related to the build process and the build environment. It is used to diagnose and recreate program faults, and to track difference between program versions. Similarly, deployment metadata tracks information about the system where the program is run, and its related environment. And finally, run-time information is data about a specific execution of an application.

Table 3.1: Taxonomy of metadata.

	Metadata	Source	Use	Format
		Compilation		
1	Abstract Syntax Tree	Compiler	Genetic Programming	Tree
2	Alias Info	Compiler, Static Analyzer	Formal Verification, Security Tools	Associative Array
3	Control Flow	Compiler	Control Flow Integrity	Graph
4	Debugging Info	Compiler	Debugger	Tree
5	Register Allocations	Compiler	Register Reallocation, Runtime System	Associative Array
6	Switch Table	Compiler	SDT Code Layout	Associative Array
7	Symbol Table	Linker	Debugger	Associative Array
8	VFT Table	Compiler, Assembler	SDT Code Layout	Associative Array
		Configuration		
9	Development System Data	Build & Configuration Tools	Reproducibility	List
10	Formal Verification	Verification Tools	Run-Time Verification	Associative Array
11	Optimizations Performed	Optimizer	Deoptimization, Debugging	List
12	SCM Revision Info	Version Control	Debugging Correlation	Scalar
13	System & Library Calls	Assembler, Build System	Security Policy Enforcement	Associative Array
14	Test Cases & Coverage	Testing Suite	Fault Isolation	List
		Deployment		
15	Code Signature	Build System	Run-Time Verification	Scalar
16	Dynamic Libraries	Linker	Policy Enforcement	List
17	Security Policy	Security Tools	Run-Time Verification	List
18	Target System Data	Deployment Tools	Fault Detection & Reproducibility	List
		Runtime		
19	Instrumentation	Runtime System	Basic Block Layout	Associative Array
20	Memory Management	Memory Analysis Tools, Garbage Collection	Performance Tools, Real-Time Tools	List
21	Profiling Data	Run-Time System / Profiler	Feedback Directed Optimization	Associative Array
22	Program Fault Data	OS, Runtime System	Debugging Tools, Automated Repair	Scalar
23	Run-Time Arguments	OS, Runtime System	Fault Detection	List

3.2.1 Compilation Metadata

Compilation metadata is information generated at compile-time, typically related to the transformation of the source language into a lower-level format. Included in this category is metadata about the structure of the program and programmer intent insofar as it can be inferred statically.

1. **Abstract syntax tree** An abstract syntax tree is an intermediate representation of the program, which contains most of the data found in the original program in an already parsed format [3]. It is useful for source-to-source transformations [26], relating source code to binary representation, and automatically modifying programs [126].

2. **Alias information** Pointer aliasing is information about what memory address can be accessed by a given memory reference. It is valuable for determining which memory references can be altered or eliminated [3]. Alias information can be used by optimizers as well as run-time systems that need to alter the instruction stream.

3. **Control flow** Identifying how the program executes and how it is divided into basic blocks allows for advanced analysis by both static and dynamic tools. Statically, control flow graphs are used as the basis for data flow analysis which enables a wide array of optimization [3]. Dynamically tools such as control flow integrity use the static control flow graph to help enforce the control flow of the program at run-time [1].

4. **Debugging info** As described in Chapter 2, debugging information is used to map source information to binary data. Debugging info is used by debuggers along with other analysis tools.

5. **Register allocations** Optimizing compilers allocate automatic variables into registers to improve performance. The specific information is valuable to run-time systems and other analysis tools which might want to later reclaim allocated registers or further allocate [130].

6. **Switch table** Compilers often emit code that computes the target of a switch via table entries. Indirect branches are used to jump to the correct code. The table location and size is necessary

to statically identify the original control flow graph for code with switch tables. SDT systems can use the switch layout to improve indirect branch translation. More details on metadata-assisted switch translation are discussed in Section 4.3.

7. Symbol table The symbol table is used during the linking process to resolve dependencies across object files. Often the symbol table which maps source symbols to addresses is retained in the binary, and can be used by debuggers to give access to break points by symbolic names even when full debugging information is unavailable.

8. VFT table Virtual function tables are used to allow polymorphic functions to target different methods depending on the concrete type. The size and targets of the table can be used by SDT systems to improve VFT translation. More details on VFT translation are in Section 4.4.

3.2.2 Configuration Metadata

Configuration metadata includes information about the methods and state of the tools used to create the program. Metadata such as compiler optimizations and build flags are important for recreating a program in the case of a problem, and also for information about exactly which code contributed to a resulting binary.

9. Development system data This metadata includes information about the OS, architecture, compiler, and system tools where the software was configured and built. Development system data is useful for identifying dependencies and for identifying discrepancies between the development system and deployment system.

10. Formal verification The process of formal verification ensures that a given implementation of a program meets its specification. Proofs of correctness can be attached to programs. Such metadata can be used by run-time systems to verify correctness before executing the code [96].

11. Optimization Typical optimizers make numerous passes over the program, and perform discrete optimization on the program. The list of optimizations (combined with instrumentation) allows the programmer to better understand what optimizations improve execution [90]. Optimization lists also allow de-optimizers to effectively remove an optimization if it is incorrectly applied.

12. Software configuration management revision number Source code is typically versioned and configured to track changes and variations. The revision number corresponds to a specific set of source files. The SCM number allows developers to better identify variations in programs, and bugs.

13. System/Library calls Library calls are calls made by application code into third party libraries. Both the static and dynamic sets of calls are valuable metadata. The static calls are useful for building security policies, for example, to help avoid arc-injection attacks [113].

14. Test cases and test coverage Program testing in an important part of software development. The set of tests determine code coverage and identify software failures before the program is distributed.

3.2.3 Deployment Metadata

Deployment metadata included information about the target system, and the environment in which the program is going to be run. Some programs have many possible targets, and a given target can change over multiple invocations of the program. This metadata is valuable to help understand how the program is going to be executed.

15. Code signature Code signing is one way of ensuring that code is only run from known or trusted sources. Typically public key infrastructure (PKI) is used, and the certificates and signatures are necessary metadata to ensure the chain of trust for signed code objects.

16. Dynamic libraries Dynamically loaded code is used as the basis for software "plug-ins." Identifying parts of the program that load code and the types of operations the code is meant to perform is important metadata for software security and performance analysis.

17. Security policy Many systems use a security policy to decide what actions are allowable on that system, and to map programmer intent to specific operations. The specification of allowable actions is important for reproducing errors and verifying security.

18. Target system data Similar to development system data, target system data includes the OS, architecture and other statistics on the target system. This data is valuable for field verification of software, to ensure that the target system has compatible features with the development system.

3.2.4 Run-time Metadata

Run-time metadata is collected during the execution of the program and contains information relating to that specific run. Such metadata in valuable in that it exposes dynamic information that is often impossible to get through static analysis.

19. Instrumentation Data about function execution, statement execution frequency and other instrumentation data [77, 131] is useful both at run-time for phase detection and for subsequent builds using feedback-directed optimizations.

20. Memory management Memory management is an important part of any large scale system. For non-garbage collected languages, details of memory management such as the specific implementations of `malloc` and `free`, and any randomization that occurs such as stack or library randomization, is valuable for debugging and memory visualization tools. For systems with garbage collection, the behavior and implementation of the garbage collector is important for performance tuning and identifying memory bottlenecks.

21. Profiling data An increasing number of systems include lightweight profiling as it becomes more ubiquitous and less invasive. Profiling data is often used immediately for run-time optimizations, but is also valuable for development, as input to feedback-directed optimizations.

22. Program fault data When programs fail, typically there is significant diagnostic information available. A complete record of memory state (a "core dump") is often stored to secondary storage. Some operating systems also gather other system state information into a bug report to aid developers. Such metadata is valuable to debuggers and automated tools for fault localization [126].

23. Run-time arguments For each invocation of the program, there is a specific set of arguments given as input to the program. Such data, combined with the target system data is necessary to reproduce program behavior that occurs in the field.

The variety and scope of the program metadata presented here demonstrates the need for a a system of comprehensive metadata management. The next section presents the design of the metadata manager tool.

3.3 Metadata Management

Metadata managers must be carefully designed to handle the wide variety of metadata described in Section 3.2. The variety and potential volume of metadata is an important design consideration. It must be able to handle a potentially very large amount of data. It also must respond quickly to queries about the data. It must provide functionality to keep data synchronized. Further, it must accept data from a variety of sources. Figure 3.1 shows the metadata manager design to meet these requirements. The metadata manager is attached to the build system to leverage the build system's knowledge about inter-file dependencies. By knowing which files are used as input to create other files, the metadata manager can track changes and invalidate stale metadata. Figure 3.1 also shows the relationship between the metadata manager and the software tools. The tools query and submit data to the metadata manager, a process described in detail Sections 3.4 and 3.4.2. The raw data is stored in a database, and metadata manager handles conversions if a tool is unable to read the manager's native format. Once the executable is built, the metadata manager can interact with the program through the software dynamic translation layer or similar advanced run-time system[1].

[1]This work focuses on compiled languages such as C and C++, however most of the design and structure would be the same for byte-code based languages like Java or C#.

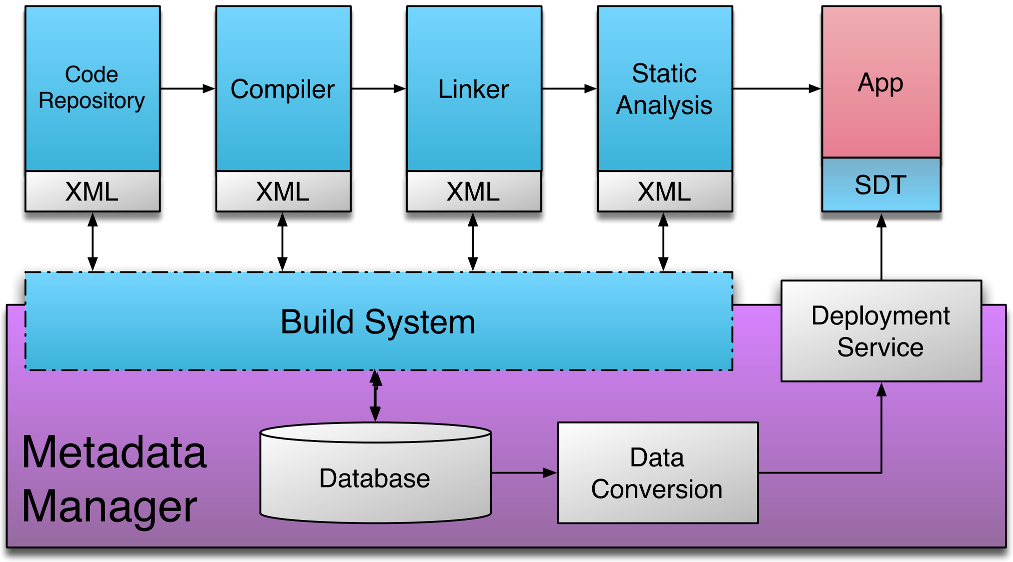


Figure 3.1: Basic structural design for metadata managers, including build and analysis tool interaction.

3.3.1 Metaman Implementation

Metaman was created as a prototype metadata manager to test these design points and gather practical experience collecting metadata. An overview of Metaman is shown in Figure 3.2. Metaman adheres to the original design principles, although it is targeted at a specific development platform (UNIX/C), and therefore it operates in conjunction with GCC and related build tool (GNU LD and GNU AS). When dynamic translation is used, Metaman integrates with Strata, one of the SDT systems discussed in Section 2.2.2.

Metaman's implementation is largely consistent with the design laid out in Figure 3.1. The prototype does makes some compromises to make the development practical. One simplification is that Metaman currently only supports development and execution on the same system. Chapter 6 discusses implementation details of how to extend metadata for different software distribution models.

The prototype of Metaman is built with the SCons build tool. [75]. SCons is a flexible build system that supports a variety of languages and systems. It is based on the Python language, and

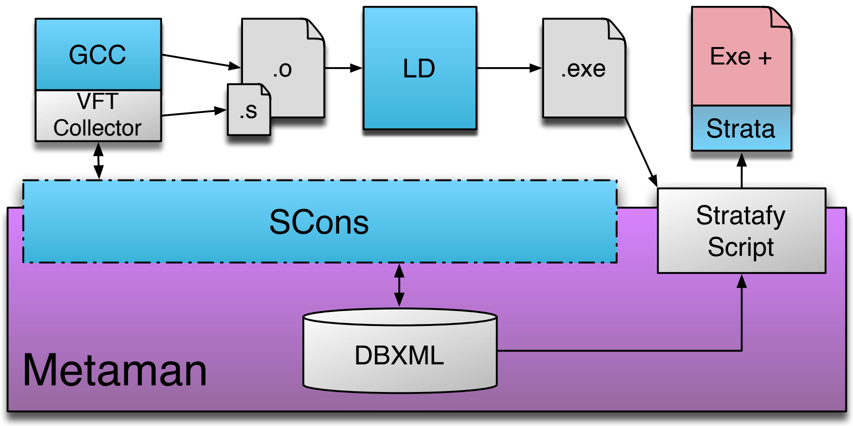


Figure 3.2: Metaman Implementation

therefore it provides an easy method of extension. SCons' modular "Tools" allow Metaman to transparently wrap its behavior by replacing default tools with augmented ones. For the prototype, Metaman replaced the `CFile`, `Object`, and `Program` tools with new versions of the tool that collected metadata and sent it to the DB XML database.

SCons' flexibility makes it a good choice for the Metaman prototype. However, to be widely adopted, the system must eventually be ported to other, more ubiquitous build systems. To support additional build systems such as Ant and Make, Metaman would require utilizing similar indirection as SCons in those tools, or altering the build tools themselves. In the case of Make, the input to the build tool is a Makefile that is often generated itself as part of the `autotools` build process. Because most build systems built with `autotools` have been developed with portability in mind (i.e., using `$CC` to abstract away the particulars of the compiler used), altering individual build systems should possible. With highly customized build systems, such as the build system of `glibc`, manual intervention will still probably be required if the usual hooks are not available.

As a program is built, the build system integrated with Metaman invokes the existing tools to compile, link and assemble the program. However, as the tools generate their output file, Metaman also collects a separate metadata file, converting that file to XML, and storing it in the XML database.

The XML database is used as the backing store of the metadata that is collected. The Metaman prototype implementation uses the Berkeley DBXML database [86], which allows updates with single or multiple XML files (e.g., one per object file)[2]. Metaman updates with multiple XML files, which then are indexed. Metaman can then perform XPath queries over the entire dataset. Finally, because it may not be practical to enable integrate an XML-based system with all tools, particularly run-time tools, the Metaman prototype used a data conversion system to convert XML data into a more compact binary format, discussed in Section 5.2.1.

The following section discusses the compile-time, run-time, and XML components in more detail.

3.4 Build-time Tools

Tools used at build-time include the preprocessor, compiler, linker and static analyzers. These tools are often run in an environment where the system-builder has complete control (i.e., a development system) and where programmers are willing to tolerate long build times in exchange for more detailed analysis. Therefore, the focus for Metaman is not on size or run-time efficiency, rather the primary goal is making it easy for tools to integrate with Metaman, and to increase the amount of information available.

The tools used at build time often perform a highly detailed analysis of the code. While the metadata is usually not included in the final application binary, many tools make their analysis available in "human readable" output to assist power users in understanding what is happening to their code.

[2]However, most XML databases support the features necessary for Metaman

3.4.1 Case Study: The GNU Assembler

An example of human readable annotations is support for `.listing` output by the GNU assembler. Typically, the assembler takes a compiler-generated assembly file as input, and produces a machine-code object file, usually Executable Linking Format (ELF) on a Linux system. For Metaman, the assembly process is very important because it represents the final mapping between source-level information and the specific bytes that will be executed on hardware. The top of Figure 3.3 shows a snippet of a typical assembly listing, from the GNU Assembler, invoked with the `-al` options, which turns on the listing output, and directs the assembler to include the assembly output as part of the listing. The code snippet shows two x86 instructions, a move of a constant onto the stack, and a call to the `puts()` function.

The purpose of the listing output is to help the programmer understand the relationship between the assembly and the resulting object file. It presents a compact representation of useful metadata: the first number (34 on the first line) is the the assembly line number. After that is the offset into the current object-file section, i.e. the .text section (`0011`). Then, the hexadecimal representation of the values being emitted (`C7042400`), and the text of the assembly line (`movl $.LC0, (%esp)`). While this information is potentially useful to a variety of tools, the output is specifically designed for reading on a fixed-width screen. Notice that the first line of the listing only contains 4 bytes of output, and the second line also refers to assembly line 34, and contains an additional 3 bytes of output. Such a split occurs because the listing of the hexadecimal output wraps the data on line 34 into two logical lines to avoid making the line too long and therefore unreadable in an 80-character terminal. While an 80-character line limit may be appropriate for a terminal window, any tool trying to programmatically use this data will require an advanced parsing scheme.

One option for collecting this data for metadata is to process the listing file, either using regular expressions or a parser generated from a CFG. Scanning for the required data is the quickest way to add data to Metaman. However, because the output is formatted for humans, creating a full parse tree is difficult, and regular expressions do not always collect the desired data in all cases. Because XML allows additional tags and attributes to be added without requiring changes to the tools consuming the XML data, it is often possible to use regular-expressions to collect the data.

Assembly Listing

```
  34 0011 C7042400              movl    $.LC0, (%esp)
  34      000000
  35 0018 E8FCFFFF              call    puts
  35      FF
```

Assembly listing XML

```
<assembly-line number="34" offset="11">
  <hex>C7042400000000</hex>
  <text syntax="att">
        movl    $.LC0, (%esp)
  </text>
</assembly-line>
<assembly-line number="35" offset="18">
  <hex>E8FCFFFFFF</hex>
  <text syntax="att">
        call    puts
  </text>
</assembly-line>
```

Figure 3.3: Assembly listing to XML conversion.

For output that is less structured, more powerful techniques such as parsing may be required.

Another option for collecting the data is to alter the assembler. In the case of the GNU assembler, this technique was straightforward and effective. The assembler has the listing functionality modularized into a single file: `listing.c`. The listing source file contained all the logic for outputting the listing data collected by the assembler. The process to retro-fit the assembler to output XML instead of text output resulting in a relatively small (150 line) patch to `listing.c`.

The second half of Figure 3.3 shows the resulting XML listing after the patch is applied. The XML version of the listing maps the logical lines of the listing to an assembly-line tag. It contains sub-tags for the hexadecimal representation as well as the assembly text. This metadata allows Metaman to gather a complete mapping of object files used to create a program, including symbolic information and static data. Further, invoking the assembler with the `-alh` flags also includes the high-level language source. That data also maps directly to an XML tag, which allows Metaman to associate source statements to particular bytes in the executable.

3.4.2 Metadata Storage Conversion, and Run-time Tools

Operating in conjunction with the application, run-time tools provide a wide range of important tasks. Because they operate in conjunction with the program, they have different requirements from their compile-time counter parts. Run-time operation is typically much more resource constrained because it is competing directly with the program for the system's resources.

Metaman uses XML to communicate with software tools, and while it offers many advantages, it also presents numerous challenges. This section examines the use of XML and how Metaman deals with the challenges of large XML files at run-time, and integrating those files with tools not designed to work with XML.

Our XML format follows the DWARF technique of breaking up the XML tree into discrete units. Doing so both easily accommodates current programming models where modules are separated into source files, and it is flexible enough to adapt different schemes that might break a program into more fine-grained units (i.e., class- or function-level units).

3.4.3 Case Study: DWARF

Current software tools used to create programs can and do produce binaries with metadata, typically debugging output. Metaman uses an XML format that is inspired by the DWARF debugging format [35], as discussed in Chapter 2. We choose not to simply extend the DWARF specification for a number of reasons. First, the tools for parsing and querying DWARF information are highly limited compared to those available to XML. Secondly, the DWARF format has a highly focused purpose—specifically aiding in program debugging—so the information collected in the DWARF format is primarily useful for program understanding, not general-purpose use.

While the DWARF format was not sufficiently general for our purposes, it does offer a good starting point for general metadata collection. We implemented a tool, `dwarfxml`, that converts DWARF information into our XML format. DWARF breaks up file information into separate compilation units, typically corresponding to individual object files. This partitioning helps the debugger correctly parse and identify shadowed variable names and other file-local data. Metaman's XML format mirrors this structure by enclosing information into a compilation-unit tag, which can comprise a standalone XML file, or can be bundled with as a sequence of tags in the case of an executable built from multiple source files.

Further, the DWARF standard translates to XML in a straightforward manner because the DWARF Debugging Information Entry (DIE) is organized in a hierarchical format. The DIE's internal structure is decided by its tag; the set of tags is listed in the DWARF standard, with their names prefixed by `DW_TAG_` [35]. The top-level entry for object files and programs is the `DW_TAG_compilation_unit`, which divides the program into object which typically correspond to object files. The DIE also have associated attributes, prefixed by `DW_AT_` which are dependent on the tag. The DIE also uses file location as an identifier to cross-reference to other DIEs in the program. The Metaman XML format uses the identifier to create a unique `id` attribute to maintain the cross-referenced metadata. The `dwarfxml` program maps the DIE's tag to an XML element, and DIE attribute to that element's attributes, using the names specified by the DWARF format. Note that the XML produced by the patched `as` program does not exactly match the DWARF-inspired XML presented earlier. It is converted into valid Metaman XML by an XSLT script, discussed in the next section.

3.4.4 Manipulating Metadata

One of the primary benefits of using XML as the metadata storage format is that there are many XML-based tools for manipulating and organizing data. One such tool is Extensible Stylesheet Language Transformations (XSLT). An XSLT program potentially alters the structure and adds information to existing XML files. The Metaman schema (see Appendix A) is meant to be highly flexible, accepting metadata in multiple forms, and then providing tools to "normalize" and restructure the data to fit the needs of the current querying tool.

The nested nature of XML allows for implicit relationships between individual metadata tags which can include important meaning to the child metadata. A key example of such an implicit relationship is location information. Symbolic and address-based location data is a key part of most program metadata, informing the tool where in the program the specific metadata applies. Some tools do not keep specific address- or symbol-based locations on data, but rather just associate them in relation to a structural metadata tag such as a function or basic block. When extracting specific metadata from Metaman, Metaman applies the location information from the nearest lexically enclosing tag. Another feature which reduces the amount of redundant information and makes encoding complex metadata more practical is the cross reference. At any point in the document which allows a specific metadata tag, that tag can be replaced by a `<metadata-ref>` tag, which includes a reference attribute linking it to the actual metadata. For metadata that is passed from object to executable, this is particularly important, because much metadata (and data—see Chapter 4) can be redundantly included for multiple objects.

An XSLT program can be used to rewrite an existing XML file into a new format concisely. The generic assembly listing XML output generated by the altered version of `as` can be easily altered to conform to the Metaman XML schema. XSLT allows programmers to rename tags, alter the XML structure or interchange tags and attributes. The XML output of `as` is easily altered partially because the patch was made with the Metaman schema in mind. However, even a program such as Valgrind [112], whose XML format preceded Metaman, can be reformatted for Metaman XML in a terse 25-line XSLT script. That process is discussed in more detail in the next section.

3.4.5 Tools Already Using XML

Some tools that are typically automated have already begun using XML. Valgrind's `memcheck` (and some other Valgrind-based tools) provide a command-line option `--xml=yes` to force Valgrind's output to be XML. The Valgrind developers have not released a formal schema or DTD, however the output contains straightforward tags for errors that the Valgrind system has discovered. Included in the `<error>` tag is the stack frame, including the name of the most immediate function (listed in the `<fn>` tag). This tag allows the Valgrind XML to be integrated into Metaman XML with a simple 25-line XSLT script by matching the `<fn>` tag to the associated Metaman `<function>`. A separate `<valgrind-run>` Metaman element provides the rest of the metadata about the command arguments and other details of the run.

GCC-XML is an extension of the GCC C++ that emits structural information about the C++ source to XML. Included in the output is class and structure information, as well as function information that is readily integrated into Metaman.

As discussed in Chapter 2, the GLX graph representation format combines a number of graph-based formats. Holt *et al.* list a large set of tools using the formats that GLX combines. These tools include the parsers (Acacia and Datrix), static analyzers (CPPanel), and graph visualization tools (DaVinci and EDGE) [62]. These analysis tools can be integrated into Metaman's schema in a manner similar to how Valgrind's metadata is handled.

These tools show the value of annotated XML output for software tools, and are improving the ubiquity of XML as an exchange format for software development. However, for XML use to be widespread it must perform well on large systems, and be straightforward to set up on small system. The next section discusses performance-related issues.

3.4.6 XML-based Metadata

There are a number of performance concerns related to using XML to store metadata. Performance in terms of size (memory and on disk), data access time, and finally performance of adding novel metadata to the system. To examine Metaman's ability to handle current metadata, DWARF entries were translated into Metaman XML. The DWARF standard translates to XML in a straightforward manner because the DWARF Debugging Information Entry (DIE) is organized in a hierarchical format, appended by name/value attributes which map nicely to XML tags and attributes respectively.

A major concern for performance of run-time tools is memory usage. If the program metadata uses too much memory it may cause the operating system to page out data that would have otherwise remained cached. The maximum memory usage for other SPEC2006 benchmarks is shown in Figure 3.4. The benchmarks are shown running natively, then with Strata alone, and finally with Strata and metadata included at runtime to improve SDT indirect branch handling (described in detail in Chapter 4).

`Xalan` has the largest increase in memory usage due to metadata. This is due to its large code base and many small functions. Figure 3.5 details the memory usage from Xalan over the course of execution. The collection of metadata switch and VFT metadata (described in Chapter 4) does show a significant increase in memory usage, but also a much shorter execution time due to the

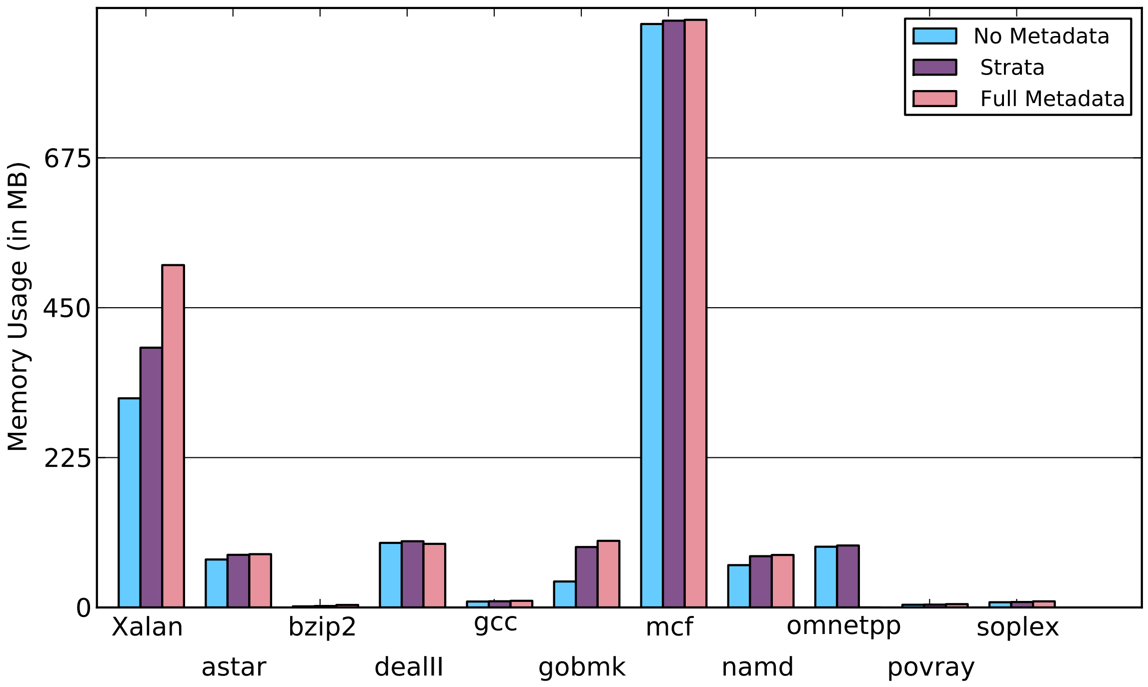


Figure 3.4: Max memory usage for benchmarks natively, with Strata and with additional metadata.

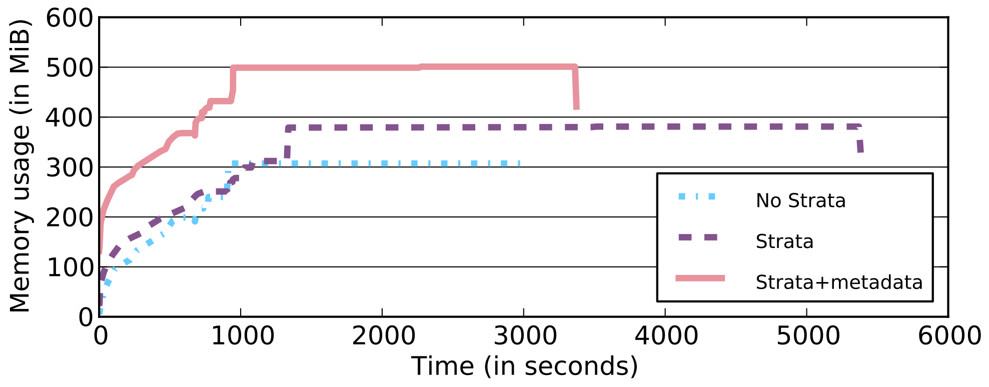


Figure 3.5: Xalan's memory usage.

performance benefits. This effect represents an important time/space tradeoff exposed by the use of program metadata.

Because XML is an ASCII-based file format, XML files are often larger than corresponding binary-encoded data. To examine the exact file size increase, we converted already-existing metadata into our XML format: debugging information. Figure 3.6 shows the increase XML and DBXML size (in MiB) per KiB of debugging information. The base XML file uses 11.6 bytes per byte of debug data (correlation coefficient $R = .99$), while the DBXML format uses 24.5 bytes per debug byte ($R = .99$). For DBXML, indices were used to improve query operations, resulting in the higher memory usage.

Because XML uses significantly more storage space than the corresponding binary DWARF representation, it is important to ask what the value is of the XML representation over the pure binary representation. One of the primary goals of Metaman is to allow for a wide range of tools to use and submit data, and further, to allow these tools to submit their own novel metadata. While this could be theoretically done by extending the DWARF specification, the tools for parsing and querying DWARF information are limited compared to those available to XML.

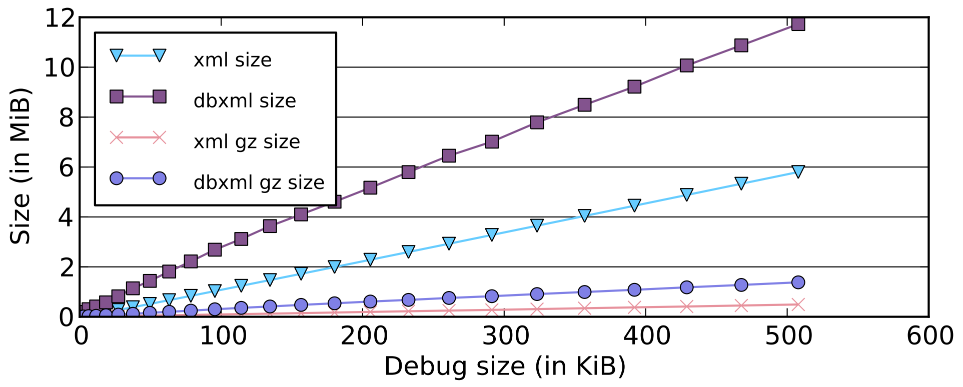


Figure 3.6: Comparison of XML size.

3.5 Design Alternatives

The ultimate goal of Metaman it to provide the easiest path for making metadata available to the widest set of tools and use cases, while meeting the primary design goals of flexibility, scalability, and ease of use. A number of design alternatives were considered to best support integrated metadata.

Initial prototypes of Metaman used a more traditional SQL database. SQL databases have been heavily researched and as a result are very robust and well polished. SQL databases make scalability a top priority, with open source and proprietary databases able to handle millions of connections per second. The trade-off with such a high level of scalability is flexibility. Traditional high-performance SQL databases require an advanced entity-relationship design, which requires a large amount of knowledge about the types of queries to be made against the data, and how those queries will be structured. Such a system might be feasible in a well-established system where everything is known about how the metadata will be used, but requiring a new database design any time any new metadata is required is highly impractical for a developing system such as Metaman. For this reason Metaman uses XML and DBXML storage.

Both raw XML and DBXML offer the flexibility required for Metaman, so another important question is when to use one or the other. Because both are based on XML, and present XML

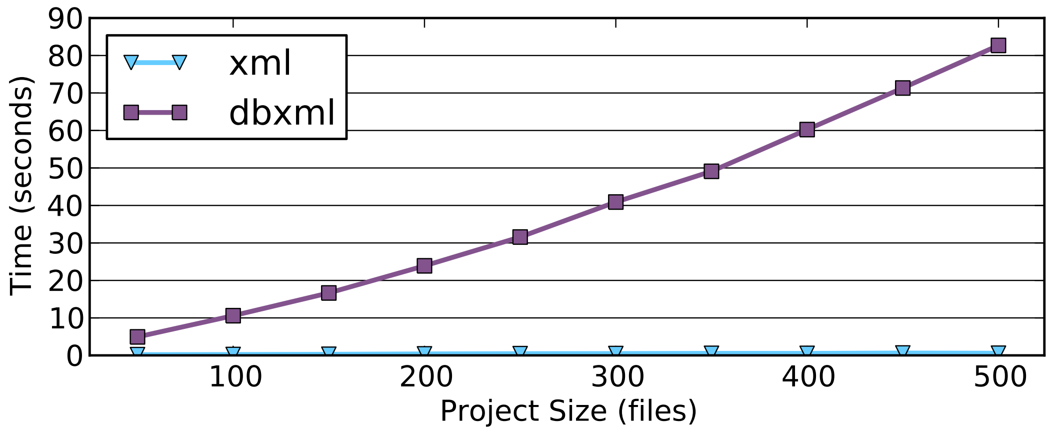


Figure 3.7: Performance of XML parsers for full-program queries.

as an application programming interface, it is feasible to select which to use on a case by case basis. DBXML requires overhead to set up the repository, so for small projects where raw XML is suffcient, it is a good choice. Further, if the tool requires processing all (or a large portion) of the metadata, rather than small specific bits of data, repeated queries to the DBXML database can cause a performance bottleneck. For example, the process of associating all the variables in a file with their types involves many disparate lookups to resolve each variable. Many of the types are defined recursively, so as a result each variable lookup results in many lookups of specific metadatum. Figure 3.7 shows the results of a naïve DBXML implementation of type gathering, with one XPath query for each specific piece of metadata needed. That implementation is compared to a pure XML implementation, where the whole XML file is parsed, and the types are resolved using an in-memory hashtable.

The X-axis marks the size of the synthetic project, in terms of number of files. As the project's file count increases, both implementations increase linearly, however, the pure XML solution remains running under a second, while the coefficient of the DBXML solution is much larger, taking over 82 seconds for a 500-file project.

In keeping with the design goal of flexibility XSLT and XPath were chosen for Metaman due to their wide use by various XML system. Other systems such as XQuery were only being introduced at the inception of this research, however as they gain support, they might become more practical in the future [124].

3.6 Related Work

3.6.1 Build Systems

An important advance in software engineering is the separation of concerns. Commonly, this manifests itself as separate files within a software project, resulting in many source files contributing to the final software product. As a result, modern programs are often composed of dozens and even hundreds of individual files that must be compiled and linked together to produce the final executable. As the size of projects and number of source files increased, it has become prohibitively difficult to manually manage building such programs. Build systems have been created to manage this task. Make is one of the first build systems. It allows users to explicitly specify dependencies and automatically identify changes in those dependent files. Encoded in the dependency is the command to recreate the file from its dependencies. If a change occurs Make recursively invokes the commands to rebuild the file that are dependent on the changed file [41]. Only building necessary files requires that the build system maintain a dependency graph of all the files of the system, a feature that is important to successfully managing metadata.

Make allows the automation of much of the build process, allowing the build tools to only rebuild the parts of the application that have changed since the last build. However, fully identifying the complete list of dependencies can be a difficult task in large systems. The `srm` system introduced a technique to address this problem. It includes language-specific analysis that detects dependencies implicitly, taking the burden off the programmer to list the dependencies explicitly [99]. Subsequent build systems such as SCons [75] and CMake [89] utilize this technique and have generalized it to support a wide variety of programming languages.

The important metadata associated with build systems – namely the dependency graph of the software system – is a vital piece of metadata. As discussed in Chapter 3, knowledge of what files affect other files is important for tracking metadata as well as the build process.

3.6.2 Code repositories

Tracking information about a program temporally has been some of the first and most common metadata about the program. Tracking changes over time is most commonly handled by a version control system. The Concurrent Versioning System (CVS) was introduced in 1990 as a such a version control system [17]. CVS is based on the Revision Control System (RCS) [123], but improved on RCS in a number of key areas to better support software development. CVS supports subdirectories, along with concurrent access (file locking). Such features allow code repositories to scale to very large projects with large code bases and many contributors.

Subsequent revision control systems have focused on "distributed" version control, allowing multiple repositories to provide better tracking for individuals in a large group working on projects potentially held on different servers or organizations [133]. Such ideas serve as the basis for newer version control systems like GIT and Mercurial [32].

Modern tools use code repositories to better understand the software engineering process. By analyzing the code changes (or "diff") from one revision to another, Buse *et al.* are able to improve the associated log messages (free form annotations describing the code changes) [22].

Chapter 4

Optimization and Overhead Reduction of SDT Systems

Users expect applications to be responsive and fast. As a result, run-time performance is an important metric for software development. As software developers discover the advantages of robust run-time systems, there is a corresponding need to make run-time systems as efficient as possible to minimize the performance impact of the run-time features.

While generating optimal code is undecidable in the general case [2], there are many algorithms which are very effective at improving code for typical workloads, both in terms of speed and code size. Naïvely generated code typically is processed by various optimization phases that analyze the code and apply various transformations with the goal of improving various run-time characteristics of the program such as execution time, code size, or energy consumption. Optimizing compilers use many types of program metadata collected in multiple analysis phases as input to their optimizations. Indeed, modern compilers typically include many analysis phases which distill the program into intermediate forms such as RTLs and SSA which represent the movement and fundamental operations of the data, typically based on the control-flow or data-flow of the program [30, 31]. Much research has gone into compile-time optimizations and intermediate representations with texts available from Aho *et al.* [3], Muchnick [95], and Bacon [9].

While there is still much ongoing research in compile-time optimization, run-time optimizations are gaining in popularity, because they offer the ability to apply optimizations that take into account the input and phases of the program—data typically unavailable at compile-time. Run-time, or dynamic optimizers, are used in a variety of contexts.

Binary optimizers allow systems to optimize already-generated binaries [10], introducing new optimizations and also optimizing legacy binaries that were distributed without optimization. Just-In-Time (JIT) compilation systems often dynamically optimize code for languages where final compilation is deferred until run-time [7, 46]. SDT systems can offer sophisticated profiling techniques that are sufficiently efficient to allow a running program to be profiled and then the collected information can be used to guide dynamic optimization of the running program.

Compilers that perform static optimizations use many types of program metadata collected in multiple analysis phases as input to their optimization algorithms. Indeed, modern compilers typically include many analysis phases which distill the program into intermediate forms, such as RTLs and SSA, which represent the movement and fundamental operations of the data, typically based on the control-flow or data-flow of the program [3].

As discussed in Chapter 2, dynamic optimizations (optimizations applied at run-time) use feedback-directed optimization (FDO) techniques to improve performance for systems with larger run-time systems, such as JITs and SDT. Michael Smith categorizes FDO improvements as "any technique that alters the realization of a program based on tendencies observed in the present run or in past runs [117]." Such a definition includes a wide spectrum of tools and techniques, ranging from very static optimizations done using information from training runs, to immediate and continuous recompilation based on run-time determination of the current hot execution path. The key observation about FDO techniques is that the feedback is simply another form of metadata. By applying metadata tools across the toolchain, it becomes less important when in development the feedback is gathered, because it can be applied equally well at all stages. If feedback is required but none is available, a profiling tool can be used to gather the information immediately. If there is profiling data available, it can be used to improve the program while it is running, and the next time it is compiled.

Both static and dynamic forms of FDO require a profiling phase to gather the requisite feedback. Software profiling is widely studied area of program understanding. Profiling done for FDO encapsulates specific knowledge that can be automatically sent back to the optimizer to improve performance. Profile information can be collected in a number of ways: binary instrumentation [78, 85], statistical monitoring [50], or hardware-based performance counters [93], or some combination of these techniques. For many of these profiling techniques, SDT systems are a natural choice as a basis of the profiling implementation. SDT systems allow instrumentation to be inserted and removed as needed; further most provide mechanisms to statistically sample a running program as well.

However, to be effective, especially for FDO occurring entirely within the run of a single program, the underlying SDT system must not introduce too much overhead. Overhead can arise from a variety of sources within an SDT system. Most SDT systems require that every instruction be examined before execution. The process of decoding and examining these instructions introduces overhead. However, most of that overhead is amortized by caching the resulting code. A significant source of overhead that cannot be easily amortized is indirect branch handling. Because indirect branches can target any address, they have be handled specially when they are executed, in case they are targeting an address not yet translated by the SDT system. This process can cause significant overhead and limit the effectiveness of SDT systems.

This chapter shows the value of program metadata in action, applying it to the real-world problem of optimizing and removing overhead from SDT systems. For all SDT-based tools, run-time performance is an important consideration. If the SDT tool slows an application down too much or consumes significant resources it will not be used, making the tool less valuable.

4.1 Indirect Branch Handling

As discussed in Chapter 2, indirect branches are a significant concern in SDT systems. Indirect branches can target any memory location and because most SDT systems require that every instruction is translated before execution, indirect branches must be handled specially. To maintain control, the SDT must have a mechanism to examine the indirect branch immediately before it executes. The extra instructions required to satisfy this requirement can be a major source of run-time overhead in SDT systems. This section shows how indirect branches are currently handled in SDT systems, and later the chapter introduces a novel metadata-based solution to achieve near-native

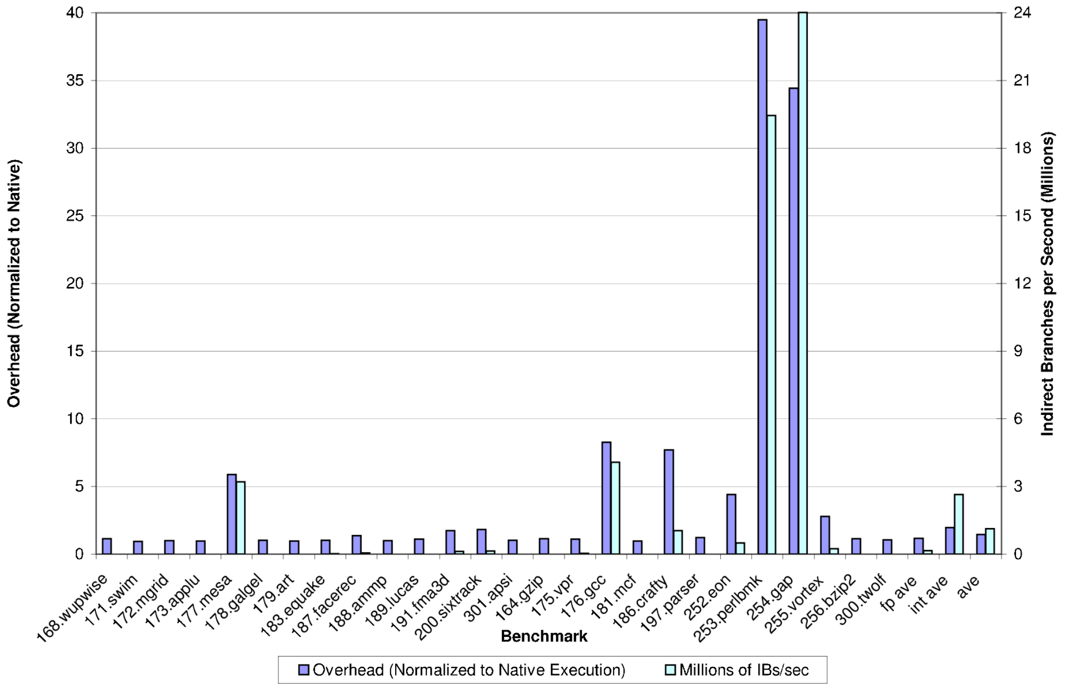


Figure 4.1: Strata SDT overhead and indirect branch rate on the SPEC 2000 benchmark.

performance for SDT indirect branch handling.

Figure 4.1 shows the correlation between SDT slowdown and the rate at which indirect branches occur. The left bar is overhead of SDT systems, normalized to native execution, graphed by the axis on the left. The right bar shows millions of indirect branches per second, with its axis on the left. The benchmarks with the highest SDT overhead, `perl`, `gap` and `gcc`, also have the highest indirect branch rate. Figure 4.2 shows the indirect branch rate grouped by cause. The top bar in the stack is the occurrence of indirect branches caused by returns, the second bar from the top is from switches, and finally indirect calls. This graph shows that the majority of indirect branches are from returns, but also that there are significant numbers of switch and indirect calls as well [61].

Traditionally, SDT systems use some translation lookup mechanism, such as the sieve, or indirect branch translation cache (IBTC) or target inlining to handle indirect branches [19, 60, 85, 119]. The sieve and IBTC are table-based approaches which cache the common translated target values. If the target is found in the table, control is transferred to the already-translated code. If the target is not in the table, control transfers back to the translation system. Because of the lookup and pos-

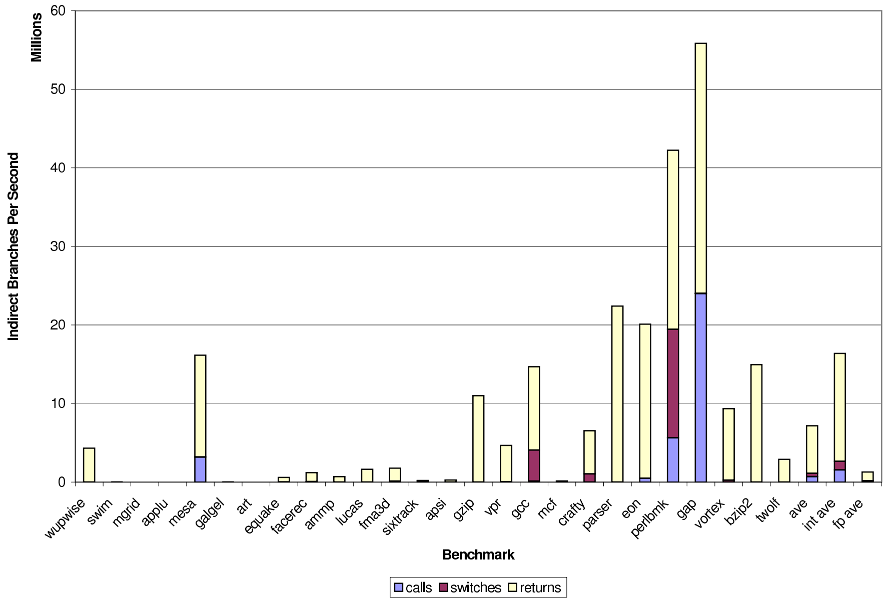


Figure 4.2: Indirect branches by type

sible jump back to the SDT system, the sieve and IBTC are general approaches that are designed to handle indirect branches that can target any program address.

The primary difference between the sieve and IBTC is the location of the hashtable "buckets." For the IBTC, the target address (the key) and the translated address (the value) are stored in data memory. For the sieve, introduced by Sridhar *et al.* [119], the check for the key is inlined directly into the code. On the x86 architecture, the sieve has a platform-specific implementation that does not require saving the `EFLAGS` register as long as the the sieve is sized correctly. This optimization is very effective because on current x86 microarchitectures, the instructions to save and restore `EFLAGS` to the stack are much slower than saving and restoring the general-purpose registers.

The high-level operation of the sieve is illustrated in Figure 4.3. The sieve stores the target application address, and then uses it to compute a hash value. It then jumps to an entry in the sieve table, which immediately jumps to either a "bucket" or back to Strata. When the translated code reaches an indirect branch, it jumps to a dispatch area, saves state, uses the branch target to calculate a hash value that is used in index into the bucket table. On a hit, it checks the branch target address and if correct, it jumps to the translated fragment and continues execution. On the x86 architecture, Strata emits the code shown in Figure 4.4 on an indirect call to the address `[eax+8]`.

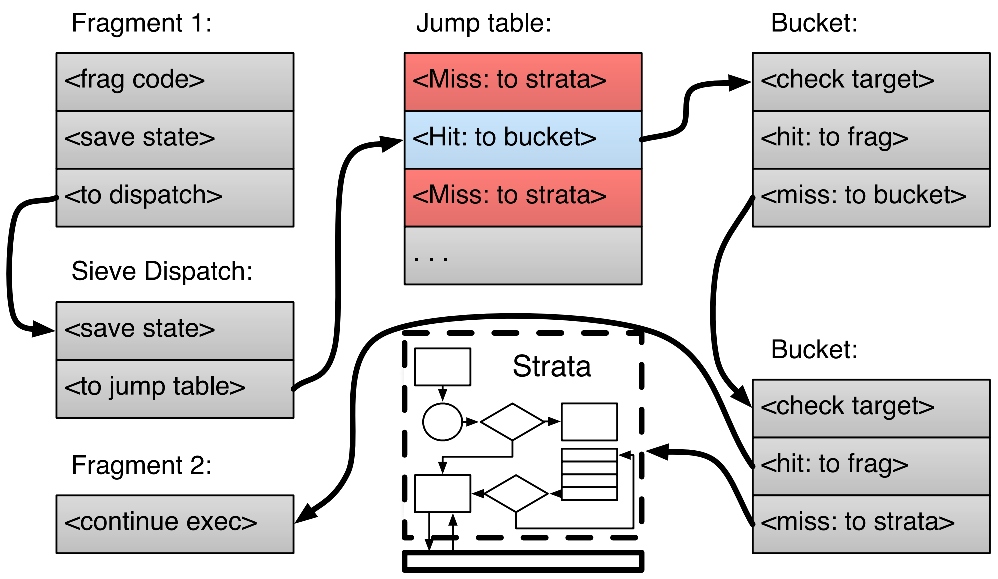


Figure 4.3: Overview of the sieve

```
0x2058: push   0x3e38      ; return address
0x205d: push   DWORD PTR [eax+8] ; tmp loc
0x2060: jmp    0xd05a      ; jmp to dispatch
        ; . . .
0xd05a: push   ecx         ; save state
0xd05b: mov    ecx,DWORD PTR [esp+4]
0xd05f: lea    ecx,[ecx*4] ; shift left
0xd066: movzx  ecx,cx      ; and 0xffff
        ; shift and add base
0xd069: lea    ecx,[ecx+ecx+0x500c]
0xd070: jmp    ecx
```

Figure 4.4: x86 disassembly for the end of a fragment jumping to a sieve dispatch for an indirect call. (Intel syntax)

```
        ; target of ecx
0x50a0: jmp    0x2100
        ; first bucket
0x2100: mov    ecx, DWORD PTR [esp+4]
        ; add the complement
0x2104: lea    ecx,[ecx+0xc250]
0x210a: jecxz  0x2111       ; check target
        ; miss - jump to next bucket
0x210c: jmp    0xed089
        ; hit case
0x2111: pop    ecx          ; restore state
0x2112: lea    esp,[esp+4]
0x2116: jmp    0x20db       ; target frag
```

Figure 4.5: x86 disassembly of a target bucket of the sieve (Intel syntax).

Here, `0x500c` is the the base address of jump table. Upon initialization this table is filled with `jmp` instructions that transfer control to a trampoline then back to Strata. The table is then updated with a jump to a bucket with the (now-translated) target. The next time that target goes through the sieve table, it jumps to the bucket, as shown in Figure 4.5.

The bucket is responsible for checking the target value against the key, and in the case of a miss it goes to the next bucket, or returns to Strata if it is the last bucket. In the case of a hit the conditional branch at `0x210a` is taken, the state is restored, and control is transferred to the target fragment [60,119]. This technique for indirect branch handling is able to handle any type of indirect branch, however, every time the indirect branch is executed, this code must run to ensure Strata has translated the target. In the case where the target is in the first bucket (which is expected for a reasonably size table), the single indirect branch executes at least 16 instructions.

4.2 Metadata Optimization

While the sieve and IBTC are powerful techniques for handling direct branches in dynamically translated code, the increase in dynamic code size is still significant, and can be a barrier to adoption of SDT solutions. Guha *et al.* have studied the memory effects of SDT systems directly [48], examining the trade-off between the memory benefits of removing code traces frequently, and the performance benefits of keeping code traces as long as is feasible. This section examines a different approach that aims to directly reproduce translated versions of the data structure used by indirect branches; these are generally not memory intensive, but offer significant performance benefits.

An important observation is that while the sieve can handle an indirect branch targeting any address, the majority of code currently in use is much more structured and restricted. For most structured programs, indirect branches are emitted as the result of one of a few programming constructs: switch tables, indirect function invocation (either through the use of function pointers or language constructs for polymorphism), and function return. These high-level constructs have a much more restricted set of targets than an assembly-language-level indirect branch. Further, with analysis of the source code—and more importantly the metadata resulting from that analysis, the SDT system can optimize the traditional indirect branch handling mechanisms to avoid the usual hashtable lookup. If the SDT system is able to determine all the possible targets for an indirect branch, it can create a translated version of the table, eliminating the need for the hashtable lookup. When the metadata is enabled, the sieve assembly shown in Figure 4.4 and 4.5 can be replaced a few-instruction assembly sequence close to the original indirect branch.

The metadata required to perform this optimization is readily available at the compilation and assembly stage. Detailed information is required about the switch, VFT, and return locations. Specifically, the following information is required:

- Switch optimization requires:
    1. the address of the switch table,
    2. the size of the switch table,
    3. the address of the indirect jump using the table, and
    4. instructions referencing the switch location.
- VFT optimization requires:
    1. the location of the VFT,
    2. the size of the VFT,
    3. the instructions assigning the VFTs to the object, and
    4. the instructions invoking the virtual function.

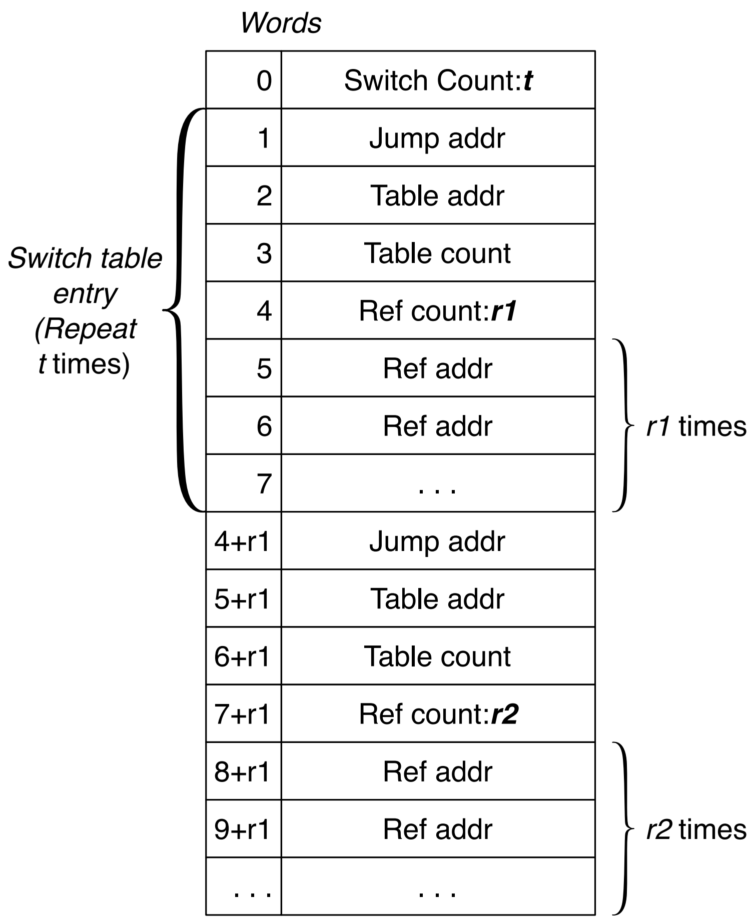
- Return optimization requires:
  1. function names and return locations, and
  2. other calls in the in the function.

An important implementation decision is exactly when to collect this metadata. For a typical optimization workflow the information needs to be gathered after optimization because the optimizer can hoist or fold instructions. However, it should be gathered before it is assembled into an object file, which can remove symbols that are unnecessary for execution, but used by the optimization. Metaman collects the data using the listing output of the assembler, as discussed in Chapter 3.

Although, it would be possible to parse the metadata in Strata using an XML parser, due to the run-time performance concerns and the need for specific analysis, the metadata is pre-parsed and encoded in a binary format. The binary format for switch information is shown in Figure 4.6. For 32-bit code, the data required for the switch is grouped into variable-length records, depending on the number of instructions referencing the switch table (*r1* for the first record in Figure 4.6). The total number of records (*t*) is encoded in the first 4-bytes of the file. The format records the number of switch tables for the module. Each switch entry is a variable-length record that contains the binary representation of the minimum amount of necessary information to fully collect the switch table in code: address of the jump instruction, the address of the switch table, and the number of other instructions referencing the switch table, followed by addresses of those instructions. That format matches the underlying C structures used by Strata, to make loading the data straightforward. Inside Strata, the data is loaded into a hashtable for efficient lookup of the entries while the program is running. The data for the VFT and returns follows a similar format. They consist of a count of the total number of entries, followed by the necessary data, including the counts of any variable sized entries, like references. The next sections discuss how this metadata is used in the specific implementations for the switch, VFT, and return optimizations.

4.3 Switch Table Translation

Switch table translation is a technique designed to reduce overhead in SDT systems. Switch table translation works by creating translated copies of switch tables into the code cache. Switch state-



Words	
0	Switch Count:***t***
1	Jump addr
2	Table addr
3	Table count
4	Ref count:***r1***
5	Ref addr
6	Ref addr
7	. . .
4+r1	Jump addr
5+r1	Table addr
6+r1	Table count
7+r1	Ref count:***r2***
8+r1	Ref addr
9+r1	Ref addr
. . .	. . .

Figure 4.6: Indirect switch info binary format

ments in C and similar languages are designed to choose one of a number of options based on a scalar value. If there are few choices, the compiler may emit a sequence of checks similar to an if/else-if construct [18, 54].

If there are a large number of choices, the compiler calculates the address of the target code block using the scalar value to index into a switch table. Then the compiler emits an indirect branch to jump to the value specified by the table. Figure 4.7(a) shows the standard layout of an indirect jump into a switch table. On x86, the target value of the jump can usually be calculated using a single instruction by jumping to a base address, `jmp_addr`, plus an offset, `eax`. The offset is calculated by the input to the switch statement, and it determines which block in the switch table will be executed.

The goal of the specialized switch translation code is to reduce the overhead from the SDT's generalized handling of the indirect branches generated by switch tables. These branches make up a significant percentage of indirect branches in SPEC2006 benchmarks like `perl`. The optimization is implemented by recreating a translated version of the switch table rather than relying on a hash table-based method such as an IBTC or sieve. Because the index into the switch table is calculated at run-time, it is not always possible to discover the targets of the switch statement without information to determine where the base of the table is located. Therefore, Metaman collects the location of each switch table, as well as instructions that reference the respective tables. At compile time, Metaman adds an additional analysis pass made on the assembly code to identify switch tables and assembly instructions that reference them. This pass inserts a symbol at each location so that the address can be identified after linking. Additionally, the symbol marking the beginning of the table is collected, along with the table size. Finally, the instructions used to calculate the base address of the table are identified and marked. When the source file is compiled this information is stored by Metaman. When the executable is linked, the linker then resolves the collected symbols into program addresses, and Metaman generates a listing for the whole program identifying all switch tables and references. This listing is then stored in an XML file which is later translated to a binary file as described in Section 4.2 (Figure 4.6).

The information gathered by the compiler is then used by Strata to create a translated switch

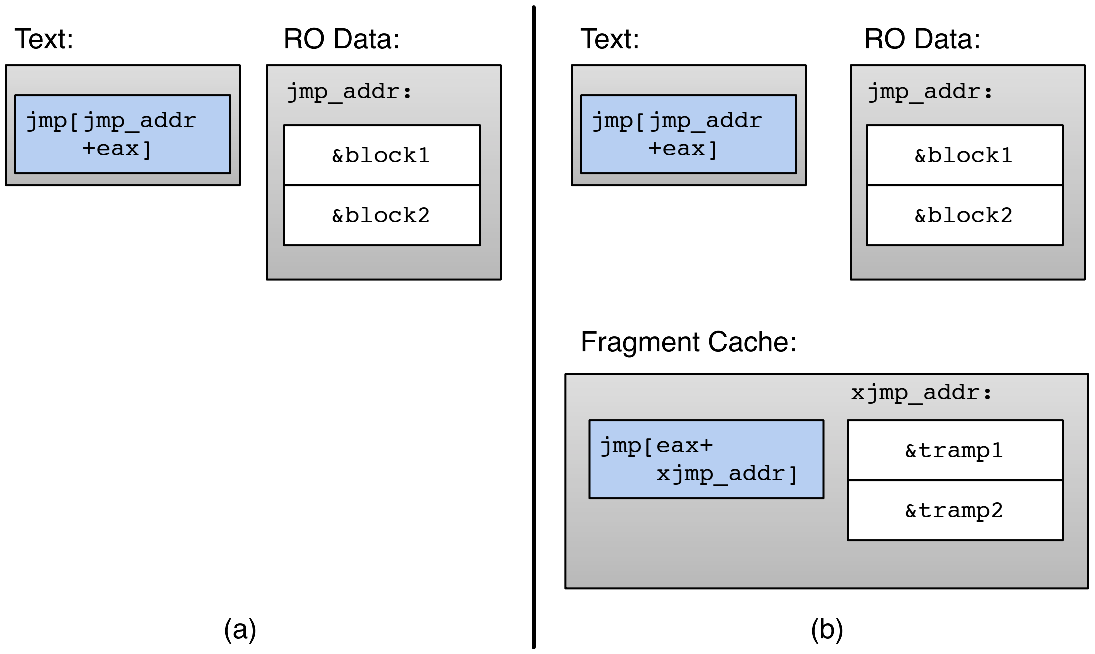


Figure 4.7: Switch Layout

```
typedef struct switch_info {
    app_iaddr_t jmp_addr; /* jump address */
    app_iaddr_t table;    /* switch table addr */
    unsigned length;      /* number of entries in table */
    /*...*/
} switch_info_t;

typedef struct switch_ref_info {
    app_iaddr_t ref_addr; /* ref addr */
    switch_info_t* switch_info;
    /*...*/
} switch_ref_info_t;
```

Figure 4.8: The C data structure for switch metadata.

table as shown in Figure 4.7(b). The jump address in the text has been gathered by Metaman and is in program memory in the `switch_info_t` data structure shown in Figure 4.8. That data structure contains a single switch entry taken from the binary file described in Figure 4.6, which was loaded into memory by Strata when it gained control of the program. With that metadata, the new switch table can be laid out lazily when Strata encounters an instruction referencing the switch, such as the jump instruction. Then the new table is created at `xjmp_addr`, with entries the contain Strata trampolines which return to Strata and allow it to translate the corresponding code block in the original switch table.

The algorithm for handling the new layout is described in Algorithm 1. When Strata encounters a switch table as it is translating the program (line 2 of `translate_insn`), it uses the `switch_ref_info_t` data structure (Figure 4.8) to identify the reference, and then it uses the `switch_info_t` structure to layout a new table, (`layout_table`), in the fragment cache using the targets of the switch. The reference is identified by looking up the value in a hashtable of references. The switch targets are iterated over, placing a trampoline corresponding to the table targets (`xjmp_addr` in Figure 4.7). The instructions that reference a switch table are recalculated (`recalculate_offset`) by using the base of the literal to refer to the newly created table. Because the addresses in the original table correspond directly to fragment addresses in the new table, the system can calculate the same offset, with the base address of the new table.

When the indirect jump for this table is encountered, it can be emitted to the code cache directly. By emitting the indirect jump directly instead of the normal indirect branch handling code, the instructions calculating the hash and comparing the tag have been eliminated, which reduces the dynamic instruction count, and also the code cache pressure.

4.4 Virtual Function Call Table Translation

Indirect call instructions provide the basis for the implementation of virtual functions in many object-oriented languages. They allow the program to invoke a function without statically knowing the address of the function. General indirect calls are very powerful constructs, used to implement

Algorithm 1 The switch table translation algorithm

```
Function translate_insn
Input: $pc$ : void * {PC to be executed.}
1: $switch\_ref$ : switch_ref_info_t
2: $switch\_ref \leftarrow$ entry_lookup($pc$)
3: if $switch\_entry \neq$ NULL then
4:     if $switch\_ref \rightarrow switch\_info \rightarrow table\_made =$ FALSE then
5:         layout_table($switch\_entry \rightarrow switch\_info$)
6:     end if
7:     $offset \leftarrow$ recalculate_offset($pc$, $switch\_entry \rightarrow switch\_info$)
8:     rewrite_instruction($pc$, $offset$)
9: end if
Function layout_table
Input: $switch\_entry$ : switch_info_t
1: $switch\_entry \rightarrow new\_table \leftarrow$ allocate($switch\_entry \rightarrow length$)
2: for i : int $\leftarrow$ 0 to $switch\_entry \rightarrow length$ do
3:     deref($switch\_entry \rightarrow new\_table + i$) $\leftarrow$ make_trampoline($switch\_entry \rightarrow old\_table + i$)
4: end for
Function recalculate_offset
Input: $pc$ : void *, $switch\_entry$ : structure
1: $immed \leftarrow$ get_immediate($pc$)
2: $new\_offset \leftarrow immed - switch\_entry \rightarrow table$
3: return $new\_offset$
```

```
typedef struct vft_info_st {
    app_iaddr_t vft;        /* VFT location */
    unsigned size;          /* table size */
    fcache_iaddr_t fvft;    /* new location */
    app_iaddr_t* refs       /* code reference to the VFT */
    unsigned ref_count      /* number of references */
    /*...*/
} vft_info_t;
```

Figure 4.9: VFT Info structure

calls through function pointers, as well as virtual function invocations in object-oriented languages. Statically identifying the target of an indirect call is undecidable; however, the targets of virtual function calls are much more restricted. Virtual function invocation is used to implement polymorphism, where the actual function being invoked can vary depending on the concrete type [83]. The Annotated C++ Reference Manuel (ARM) provides a general description of the behavior and "plausible implementation" of the virtual function table [37].

The layout of the metadata for the VFT is shown in Figure 4.9. Similar to the data layout for switches, the VFT layout requires the location of the VFT (`vft` in Figure 4.9), the size of the table (`size`), and the references to the table in source.

To show how this metadata is used, Figure 4.10(a) illustrates the typical text and heap layout of an object being constructed, similar to the one presented by ARM. Selection of the actual function to invoke is done through a virtual function table (VFT). In this example there are two functions that can be virtually invoked: `func1` and `func2`. These objects are placed in the VFT, which is statically allocated for each concrete type. When the constructor is run on a newly allocated object, an implicit field (`vft`) is stored, which points to the VFT specific to the concrete type of the object. When a virtual function is invoked, the compiler emits code to look up the function address in the VFT, and invokes the function through an indirect call.

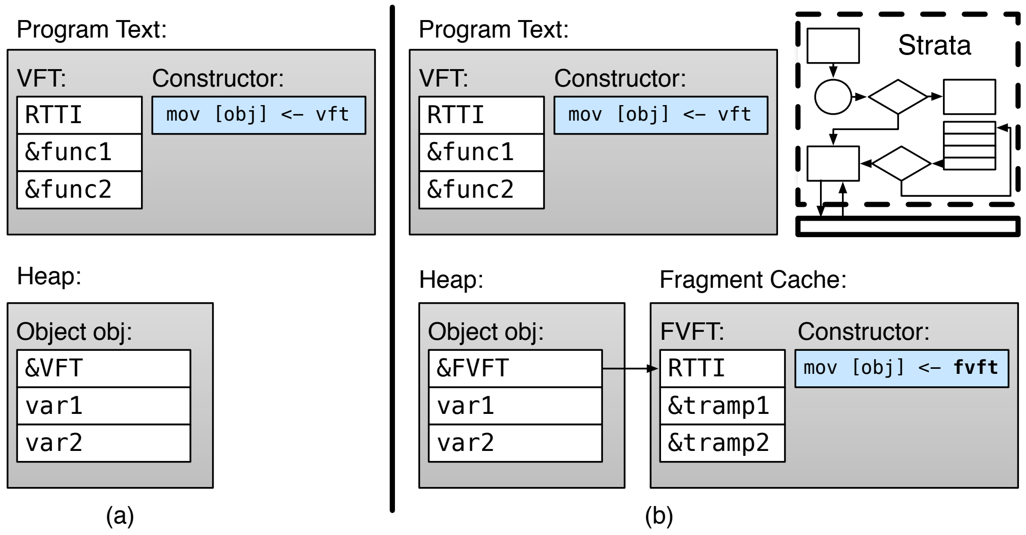


Figure 4.10: (a) Normal VFT initialization. (b) VFT initialization with Strata translation

The targets of the virtual function call are limited to those in the VFT. Using this fact, Strata can translate the VFT entirely, and remap references to the VFT to point to the translated VFT. To implement this optimization, Strata checks the PC to identify the instructions that load the VFT into the new object. To perform a full translation of the virtual function table, Strata requires data similar to that of the switch table: table size, table location, and instructions referencing the table. This metadata is collected and laid out in the same binary format as the switch table (Figure 4.6).

Figure 4.10(b) shows the layout of the translated table under Strata, and Algorithm 2 shows the algorithm of identifying and laying out the translated VFT table. When the instruction to load the VFT is encountered, Strata creates a new VFT, `fvft`, which contains pointers to trampolines that will jump back to Strata and translate the target function. As Strata copies the VFT to the translated VFT it checks the ABI being used to identify entries that are not code addresses and copy them directly. In the case of GCC's implementation, the first entry of the VFT is reserved for run-time type information (RTTI). Strata then translates the `mov` instruction to load the newly created `fvft` instead of the original. When the indirect call is encountered, it is unnecessary to emit a sieve, but instead a simple push and jump combination can replace the call shown in Figure 4.11.

The VFT metadata collected by Metaman, described in Section 3.1 allows Strata to implement

```
0x2058: push ret_addr           ; return addr
0x205d: jmp DWORD PTR [eax+8] ; call becomes a jump
```

Figure 4.11: The new indirect calling sequence.

Algorithm 2 The virtual function table translation algorithm

Function translate_insn

Input: pc : void * {PC to be executed.}

```
1: $vft\_entry \leftarrow$ entry_lookup($pc$)
2: if $vft\_entry \neq$ NULL then
3:     if $vft\_entry \rightarrow table\_made =$ FALSE then
4:         layout_vft_table($switch\_entry$)
5:     end if
6:     $offset \leftarrow$ recalculate_offset($pc$, $switch\_entry$)
7:     rewrite_instruction($pc$, $offset$)
8: end if
```

Function layout_vft_table

Input: vft_entry : structure

```
1: $vft\_entry \rightarrow fvft =$ allocate($vft\_entry \rightarrow table\_size$)
2: for i : int = 0 to $vft\_entry \rightarrow table\_size$ do
3:     if abi_non_address($i$) then
4:         $vft\_entry \rightarrow fvft + i = vft\_entry \rightarrow vft + i$
5:     else
6:         deref($vft\_entry \rightarrow fvft + i$) = make_trampoline($vft\_entry \rightarrow vft + i$)
7:     end if
8: end for
```

Function recalculate_offset

Input: pc : void *, vft_entry : structure

```
1: $immed \leftarrow$ get_immediate($pc$)
2: $new\_offset = immed - vft\_entry \rightarrow vft$
3: return $new\_offset$
```

this performance enhancement. For programs that make heavy use of virtual functions, such as the highly object-oriented `Xalan`, reducing the instruction count from at least 16 to 2 results in a significant performance gain, discussed in detail in Section 4.6.

4.5 Metadata Insured Return Layout

As shown in Figure 4.2, the most common type of indirect branch to occur in most programs is a return instruction. Return instructions are dynamically paired with call instructions which store the current address on the stack and jump to the call location. When the return instruction is reached, control is transferred back to the instruction immediately following the call. Modern object-oriented programs strongly emphasize small functions to improve abstraction, and therefore efficient return handling is vitally important.

By default, Strata converts a call to a pair of instructions, the first pushing the application address, and the second jumping to the target function. Then the return instruction can be treated as a standard indirect branch, handled by the sieve. However, that technique results in at least 16 additional dynamic instructions, which can be a large percentage of small, modular functions, and therefore result in a noticeable performance penalty. Fortunately, the majority of uses of calls and returns come from static function calls, which makes call/return behavior very regular compared to the other indirect branch instructions.

A simple improvement to the general sieve approach for returns is to have the SDT use the stack directly for returns. Using such an approach, Strata translates a direct call by emitting code that pushes a fragment cache address rather than application address. The instruction is required to be unaltered by the Application Binary Interface (ABI), and as long as that condition holds the return can be emitted to the fragment cache directly. When the return is executed, the fragment cache address is at the top of the stack, and Strata maintains control. This reduces the dynamic instruction count for executing returns from at least 16 down to 1, making the overhead for return values very small [60].

For the return optimization to work, the ABI must be strictly followed. Specifically, the return address must not be altered when it is on the stack. However, that restriction is not enforced in hardware, so assembly language programmers sometimes modify return addresses to effect optimizations (e.g., custom sibling-call optimizations). Furthermore, there are some libraries, such as C++ exception handling mechanisms, that read return addresses for various purposes, including "walking the stack" to identify the function frames present on the stack. It should be noted that these special cases are rare, and large workloads, including all of SPEC2000 can be run without any violation [60].

However, in the case of custom-coded assembly or code of unknown origin which could be malicious, the corner cases can allow the program to differ when executed under the control of the SDT system, a dangerous violation of transparency. For example, if returns are allowed to be executed directly in the SDT system, a malicious or simply clever programmer could use the return address to change control flow. A malicious programmer might wish to alter control flow to execute injected code, or to execute an unintended library function. A clever programmer might simply want to eliminate overhead by manually performing a tail-call optimization. In both cases, if the SDT system emits and uses return instructions directly, the system will lose control and start directly executing application (or malicious) code. For SDT systems designed for performance, such as Dynamo, such an outcome is not ideal, but may be acceptable if it help provide a performance benefit on average [10]. However, for SDT-based security systems, such an outcome is unacceptable.

With the use of Metaman, the optimization can be implemented safely. With compiler information about the call-graph, the optimization is only applied to calls and returns emitted by the compiler. During compilation and linking, Metaman does analysis and determines which calls and returns can be safely optimized. In the majority of cases for compiled programs, the optimization can be used. In the cases where the calls and returns cannot be determined to be transparent, the original return address is placed on the stack, and the sieve is used for those return instructions, rather than a return instruction, which might result in altering the semantics of the program.

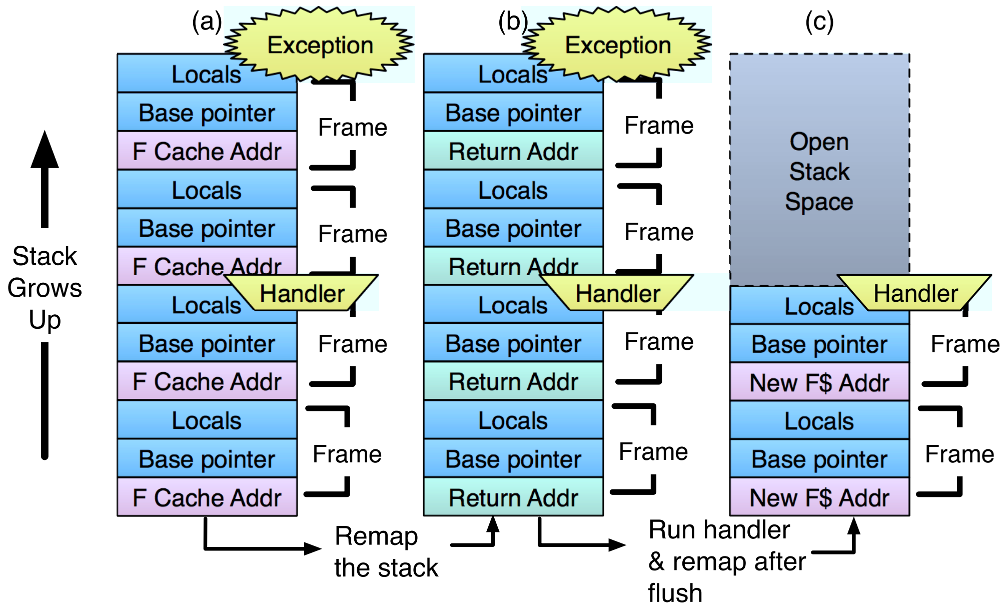


Figure 4.12: Stack view of exception handling

4.5.1 Exception Handling

An important special case is exception handling. Most object-oriented languages provide exception handling to allow the program to throw exceptions at arbitrary program points and have execution resume at a catch statement. When an exception is thrown, the run-time system must find the frame on the call stack that handles the exception. To handle exceptions in C++, the exception handler walks the stack, identifies return addresses, and uses those addresses to index into an exception table. If an exception handler is found, stack cleanup code is executed and control is transferred to the exception handler. If the exception table lookup fails, a default handler is invoked.

As discussed, when the return handling optimization is used, fragment cache addresses are placed on the stack instead of return addresses. Figure 4.12(a) shows the stack at the time of the exception. To identify when an exception occurs, Strata monitors calls to the function that walks the stack (`_Unwind_RaiseException`). When the application calls this function, Strata saves program state and then calls a wrapper function, `_strata_Unwind_RaiseException`. This function uses the `libunwind` library to identify the translated return addresses on the stack and replace them with the actual return addresses, as seen in Figure 4.12(b). The code cache is then flushed, and Strata begins translating the actual exception handling code, with the optimization temporarily disabled. The cache must be flushed so that while the exception handling code is run, no returns from already translated library code are executed. Strata then identifies the end of the exception handling code by identifying the code sequence emitted by `__builtin_eh_return` and checking the stack location to ensure that it is the end of the error handler, and not cleanup code. Once the end of the error handling is reached Strata flushes the cache again to remove any non-fast return library code. Finally, it re-enables fast returns and recreates the fragments pointed to by return addresses below the current stack location, as shown in Figure 4.12(c). Once the stack is correctly remapped, execution continues at the exception handler, under Strata's control.

4.5.2 Optimization Handling

Another case that potentially requires special handling is compiler optimizations that manipulate the stack. At optimization level `-O2`, gcc 3.3.5 implements tail-call recursion optimizations which changes recursive calls into iteration, as sibling optimizations which remove the stack frame before making a jump to the target function. In the case of tail-call optimizations, the recursive call is completely translated to iteration and the stack is not affected. For sibling optimizations, the stack frame for the caller is destroyed before transferring control to the callee, when invoking the callee is the last operation performed by the caller. The caller's original return address is left on the stack, and then control is transferred to the callee by a jump instruction instead of a call instruction, effectively turning the caller's return address into the callee's. With the return optimization implemented, the return address will be a translated address to the desired target. Since the stack manipulation does not actually alter the address, the optimization is not affected in the case where both the caller and callee are safe functions.

Metaman computes the static call graph in order to identify sibling optimizations into code not controlled by Metaman. In that case, the calling function is not included in the list of functions that are eligible for the optimization.

	Switch	VFT	Switch & VFT
`switch`	0.753268	1.901186	0.726263
`cpp_mb`	1.768101	1.322641	1.258121
Average	1.260684	1.611914	0.992192

Table 4.1: Indirect and VFT microbenchmark performance data normalized to baseline strata with a sieve. Slowdown multiplier relative to native speeds. Values less than 1 indicate speedup.

4.5.3 Other stack manipulations

Along with the special cases described above, there are a number of other x86-specific implementation constructs that can manipulate the stack, and therefore should be considered when using the return optimizations. Position independent code uses a thunk to obtain the PC to calculate the absolute position of relative offsets. Similarly, the GCC builtin `__builtin_return_address` is a programmatic interface for retrieving a function's return address from GCC. In both of these cases, an easily identifiable (though compiler- and version-specific) code sequence is emitted. Strata identifies these sequences and alters them to insert the correct return address in these cases. Other constructs such as threads and signals have the potential to effect the stack, but on x86 they do not directly manipulate return addresses, and therefore do not need to be specially handled.

4.6 Performance

The indirect branch optimizations discussed in Sections 4.1-4.5 have been implemented in the Metaman prototype. Strata is used as the software dynamic translation system, with the sieve as the baseline indirect branch handling mechanism. The experiments described in this section used the GNU software toolchain and the Metaman prototype described in Chapter 3. The experiments were run on a 2.8 GHz dual Pentium 4 system running Debian Linux, with 1GB of RAM. Application code was compiled with `gcc` version 3.3.5, using the `-O2` optimization flag.

The SDT branch handling optimizations were tested on the C and C++ SPEC2006 benchmark suite [55], as well as microbenchmarks designed to expose the overhead of virtual function calls and switch tables. The VFT microbenchmark, `cpp_mb`, consists of a simple class hierarchy which exposes a virtual function. The virtual function is invoked, through the abstract class in a tight loop.

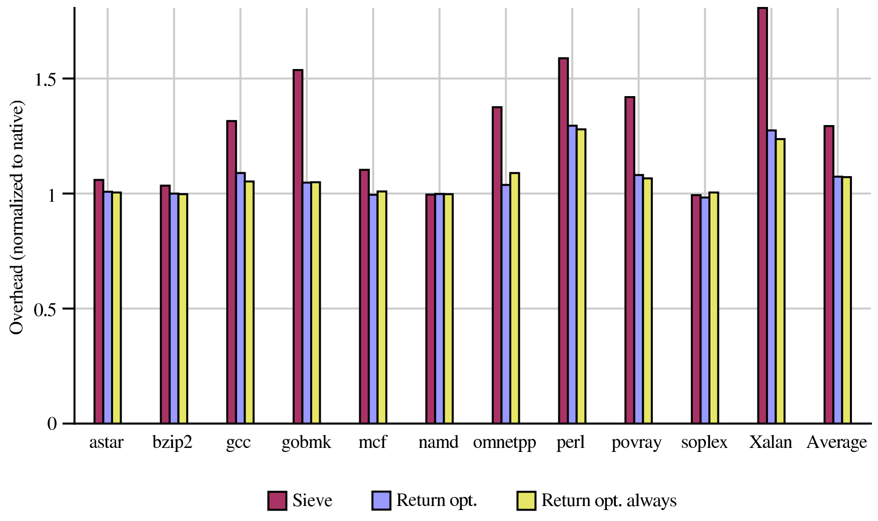


Figure 4.13: SPEC2006 performance for optimized return mechanism

The VFT handling optimization reduced overhead of that benchmark from 1.76x to 1.25x. The switch microbenchmark, similarly, consisted of a loop over a switch statement, large enough to be converted into a switch table. Using the switch table optimization on the switch microbenchmark, overhead was reduced from 1.9x to .7x. Data is shown in Table 4.1.

Figure 4.13 shows the performance of the return optimization on the C and C++ SPEC2006 benchmarks. The graph compares performance of the SDT system, normalized to native execution speed (i.e., no SDT system). Thus, the bars below 1 indicate a speed up, while bars above 1 indicate a slowdown. The first bar, labeled “Sieve,” shows the performance of returns using the sieve mechanism. The second bar shows the performance of the return optimization using Metaman metadata. Finally, for comparison, the last bar shows the performance of the “always on” return optimization. The always-on optimization performs the return optimization regardless of whether the return can be determined to be safe by Metaman. Notice that the return optimization performs competitively compared to the potentially unsafe always-on version. In fact the return optimization outperforms the always-on version on `omnetpp`, `mcf`, and `soplex`. This is due to advantageous caching effects resulting from differing code layout.

Figure 4.14 shows SPEC run-time overhead when the switch and VFT optimizations are applied. The first bar shows the baseline performance of Strata running without the return optimiza-

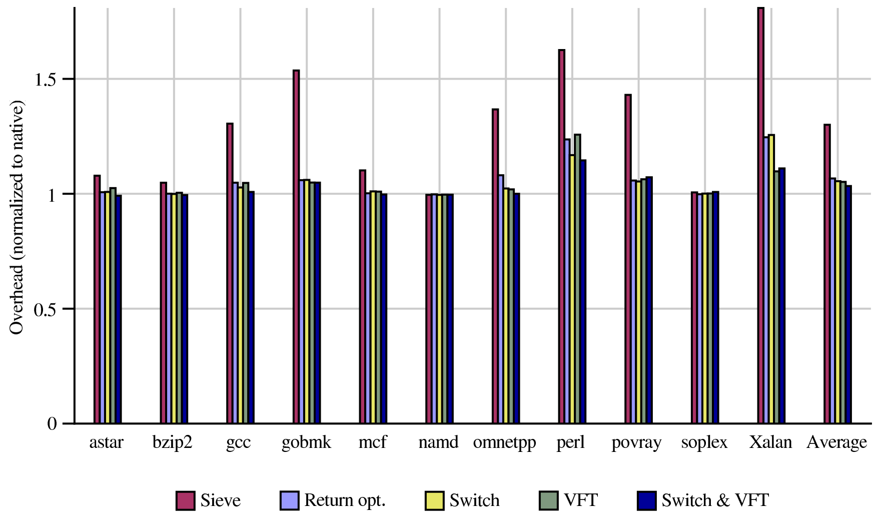


Figure 4.14: SPEC2006 performance for virtual function tables

tion, the second shows Strata running with the return optimizations. The third and fourth bars include the return optimization and then show the switch and VFT optimizations, respectively. The final bar shows all the optimizations, allowing Strata to perform within 3% of native performance. For the switch and VFT optimizations, specific benchmarks show differing improvements based on how frequently the type of indirect branch occurs. For the VFT optimizations, `Xalan` shows a 15% improvement, due to heavy use of the virtual interface used by XML. Perl also improved, due to large amount of switch invocations.

Figure 4.15 shows the fragment cache memory usage, as opposed to general memory usage examined in chapter 3. A number of parameters can affect code cache usage, notably how entry points to fragments are identified. The version of Strata used for these experiments was not specifically optimized for fragment cache usage, therefore the effect can vary from benchmark to benchmark. The return optimization offered a significant size decrease for `DealII`, due to the suppression of partial function inlining. However, those gains were removed when the VFT optimization were turned on.

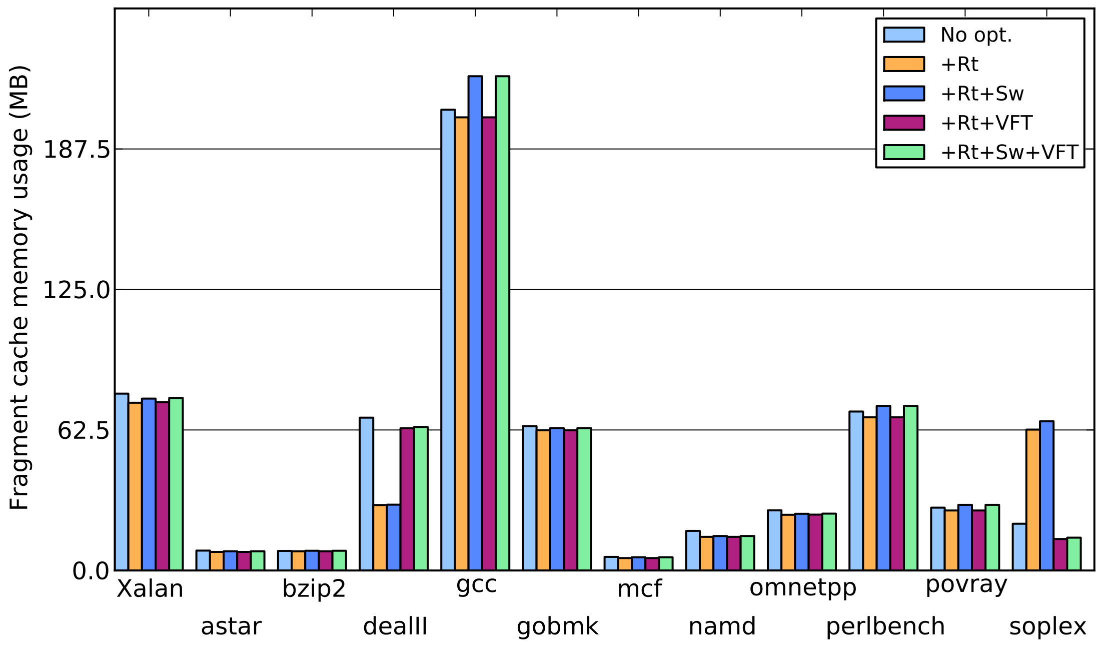


Figure 4.15: Fragment cache memory usage with the return optimization (Rt), switch optimization (Sw) and virtual function table optimization (VFT).

4.7 Related Work

Many tools have built custom solutions to the problem of run-time indirect branch handling. Dynamo describes their indirect branch handling technique as a "special switch table," with the most commonly taken path inlined [10]. Such a technique favors indirect branches with only a small number of targets, because those can easily be inlined with little overhead. Similarly, Pin uses a series of inlined checks, backed by an inline hashtable. If none of the checks match the target, the hashtable lookup would find other targets that had been translated [85]. HDTrans introduced the sieve technique, and the corresponding X86 optimization discussed in section 4.1 [119]. Scott *et al.* studied overhead reduction techniques in SDT, introducing the IBTC. The IBTC combined with fragment linking allowed Strata to perform well enough to be used as the basis of a large range of tools [108]. Hiser *et al.* performed a close examination indirect branch handling in SDT systems, evaluating the performance of the sieve, IBTC and other techniques across a series of design choices and microarchitectures [60].

Kim and Smith have proposed a hardware extension to ease the indirect branch translation problem [70]. Such a system offers many performance benefits for SDT systems in general, offloading a significant amount of work form software to hardware. However, Kim and Smith's proposed system has yet to be implemented in readily-available hardware.

These systems mostly focused on the run-time problem of identifying the target of indirect branches during execution. Comprehensive metadata allows the Metaman approach to utilize valuable compile-time data, and therefore it offers a novel technique for doing indirect branch translation.

Chapter 5

Program Understanding & Security

Over the past few decades there have been tremendous advances in software engineering techniques. System designers have been able to create powerful systems that abstract away the details of the underlying system. Modern computer systems can be thought of as a series of abstractions stacked on top of one another, each moving further away from the hardware and giving the programmer more freedom and flexibility. For example, the ABI builds on the ISA and adds structure so a program can interact with library functions and properly encode data structures. [4, 46, 114]. These abstractions allow large teams of programmers to collaborate on large-scale systems where an individual programmer is not necessarily aware of the workings of every component. However, when the programmer's understanding of the abstraction and the implementation of the abstraction diverge, it can be a source of program errors [47]. Program errors have a significant impact on the software industry and the general economy. In 2002, NIST estimated that software bugs cost the economy \$59.5 billion dollars [44]. These costs create an important need for programmers to properly understand and debug their code.

Another concern related to program understanding is program security. Many errors in programs allow malicious users of the program to alter its intended behavior. These exploitable errors provide a path for malicious attackers to gain control of a running program. Then, once the attackers control the program, they are often able to completely control the system. According to Computer Economics Inc., in 2006 malware cost businesses \$13.3 billion [25]. Security tools can provide assurance that undiscovered program errors do not compromise the entire program or the entire system.

Understanding complex programs is often difficult because of a disconnect between the programmer's intent and the actual behavior of the program. High-level languages abstract away machine details such as stack frames, parameter passing conventions and register allocation, and the resulting program can be different depending on how the compiler chooses to generate the final code [11]. Program metadata, such as debugging information, provides a method to relate the original high-level code and data structures to the resulting low-level instructions and memory locations. The mapping from high-level to low-level allows programmers, with the aid of a debugger, to identify when their high-level intent and the low-level code diverge.

Using program metadata to map intent to binary code allows the program to gain valuable insight. To achieve a better understanding of program behavior, systems builders have developed an array of program understanding tools utilizing program metadata. Debuggers, profilers and analysis tools allow programmers to examine the state of the program as it is running, and confirm or deny their expectations of the program's internal state. The use of these tools provide the programmer with an avenue to gain insight into program behavior and pinpoint when and where their model of an abstraction breaks down.

Debugging information that enables high-level debugging and profiling, along with the profile and debug data itself are valuable sources metadata for programmers. As discussed in Chapter 3, symbolic debugging is enabled by debug metadata supplied by the compiler. Similarly profile data is combined with symbols and line number information to allow profilers to give a highly accurate view of what statements are potentially causing performance problems. Further, the profile data collected can be cycled back into new versions of the program, using feedback-directed optimizations. Internal program invariants such as memory management or locking rules observed while debugging can be brought back into the program as additional annotations [39].

Security tools also rely on program metadata. Many heuristic-based security tools infer programmer intent and codify those inferences as metadata. For example, systems attempting to identify malicious access to a system develop profiles of typical behavior and increase scrutiny on behaviors that diverge from the profile [82]. Other tools rely on information known at compile-time to make it impossible to reach states known to be erroneous. Control flow integrity uses such a technique by statically verifying as much of the control flow as possible, and adding dynamic checks to indirect control flow that cannot be statically verified. [1]. In an effort to build more secure systems, programmers have turned to metadata in the form of annotations to better convey the purpose and bounds of the program [39, 104].

This chapter examines the uses of program metadata in the realms of program understanding and security. Using metadata gathered at compile time, Metaman is used to enable advanced memory protections that combine both static and dynamic information to detect buffer overruns. Additionally, the chapter explores how Metaman can be integrated into novel systems, helping to automate the genprog automatic bug fixing tool. These tools rely heavily on program metadata, and therefore show how pervasive integrated metadata can help build program understanding and security tools.

5.1 Memory protection

An important area for both program understanding and security is memory protection. Large programs written in memory-unsafe languages often have memory management schemes that can be poorly documented and that are difficult to use correctly. If a programmer violates the implicit memory management rules, the program can leak memory or contain dangerous vulnerabilities. One alternative is to use memory-safe languages such as Java and C#, whose semantics do not allow unsafe writes to memory. However, high-performance and real-time applications require features of memory-unsafe languages such as C and C++. Therefore, memory safety is an important and well-studied area of research [16,26,59,97,98,105,112]. Previous research covers a large range of memory errors, however we are focused specifically on buffer overflows because of their ubiquity in programming today. The MITRE Corporation lists the “Classic Buffer Overflow” vulnerability as the number 3 vulnerability in their Top 25 Common Weakness Enumeration (CWE), behind only cross-site scripting and SQL injection [87]. To address the problem of buffer overflows, Metaman and Strata are used as a basis to build the Buffer Overflow Detection Engine (BODE). BODE uses debugging metadata to detect overflows on the stack, heap and global memory.

Many of the current tools for identifying run-time memory errors rely on compile- or link-time hooks, and focus primarily on either heap overruns or whole-frame stack overruns. BODE is a tool that requires no compile-time information other than basic debugging info, and identifies buffer overrun errors using a combination of static debugging information and runtime instrumentation. With the debugging information, BODE is able to create a layout of the stack at per-variable granularity to identify buffer overruns in stack variables even if the overrun does not leave the stack frame. Then at run-time, it monitors program writes and identifies which variables they are accessing. If a write location sequentially accesses multiple variables, it is possibly due to programmer error, and it is flagged as a possible overrun. Concentrating on writes allows BODE to remain relatively lightweight, while still catching a large segment of errors, including potentially malicious overwrites.

There are many systems for detecting memory errors, and most fall roughly into either static or dynamic detection, though some modern systems are now using a hybrid approach, combining both static and dynamic analysis. Table 5.1 shows how BODE compares to some of the popular memory error detection systems. The first column shows what is being instrumented (reads, writes, or both) at run time. The second column shows what is being tracked in memory. BODE tracks write locations within the program text, whereas other systems track pointers or memory objects. The third column shows what static analysis is done before execution begins. Note that Annelid and Memcheck require no static analysis, while MEDS and BODE require analysis on binaries, Dataflow integrity (DFI) requires source code access and CCured requires programmer annotations of the source code [23, 26].

5.2 Buffer Overrun Detection Engine

BODE is a Metaman- and Strata-based system which leverages debugging metadata in the realm of memory protection. It uses program metadata to create a model of memory and uses that model to

	Instruments:	Tracks:	Static Analysis:
BODE	Writes	Write Locations	Debug info
MEDS [59]	Reads/Writes	Referent/Objs.	IDA Pro
Memcheck [112]	Reads/Writes	Memory (shadow)	None
Annelid [97]	Reads/Writes	Referent/Objs.	None
CCured [26]	Reads/Writes	Pointers	Manual
DFI [23]	Writes	Write Locations	Source-level

Table 5.1: Table comparing popular memory analysis system.

identify when a buffer overrun is occurring. The structure of BODE is shown in Figure 5.1, with the static components on the left and the run-time layout on the right. BODE loads the executable, Strata, and the executable's metadata from Metaman and combines them into a new executable that contains both the original executable code as well as a run-time system to perform the necessary instrumentation. In some cases it is possible to insert the instrumentation directly instead of using a run-time system such as Strata. However, using the run-time system solves the code discovery problem inherent in static binary modification [64].

BODE's operation is split into two parts: data collection and run-time analysis. The data collection occurs at compile-time while the run-time analysis occurs during execution. Metaman facilitates the transfer of the metadata, and allows BODE to effectively apply the debugging information.

5.2.1 Data Collection

To create the instrumented executable, four inputs are necessary: The executable itself, the SDT system (Strata), the instrumentation system (BODE), and the debugging metadata supplied by Metaman.

The DWARF debugging information is collected by Metaman [127]. BODE queries the data, and combines the necessary parameter and variable locations. Strata is then combined by BODE, and then it produces an executable altered to include Strata and BODE run-time monitoring. BODE uses metadata data collected from the debug information as well as run-time analysis to determine if the memory accesses of the program overrun the associated source-level data structure. Metaman collects the DWARF debugging section of the ELF executable [35]. DWARF data used by BODE is inserted into the executable by the compiler, typically with the `-g` option. The DWARF format is

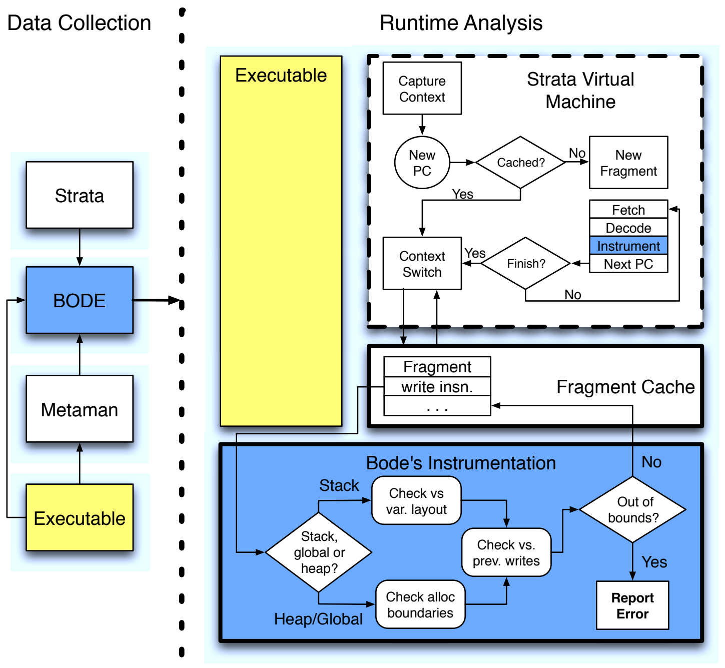


Figure 5.1: The design of BODE.

a tree-based structure, discussed in detail in Chapter 3. The DWARF information provides BODE details about each module, including location of functions, layout of automatic variables on the stack, as well as locations of global variables and associated type information of all the variables. Metaman collects the data related to functions and variables of the program, which is stored by DWARF as “Debugging Information Entries” (DIE). The DIEs specifically of interest to BODE are `DW_TAG_formal_parameter`, `DW_TAG_variable` entries for each function in the module, as well as the `DW_TAG_global` entries. The entries encode the location of the variable, which allows BODE to populate the tree mapping the address space to variables. The entries also encode a reference to their type information which allows BODE to determine the size of the variables. On the stack, the location is dependent on the stack frame, indicated by the base pointer (`ebp` on x86).

At run-time BODE utilizes Strata to identify writes to memory as well at the stack layout. The specifics of run-time data collection are detailed in the next section.

5.2.2 Run-time analysis

The run-time component of BODE is located on the right half of Figure 5.1. Strata's flexibility allows it to alter or instrument instruction streams before they execute. For BODE, Strata instruments instructions that write to memory. For performance, it only instruments writes through a register other than `EBP` and `ESP`, an optimization discussed in Section 5.2.4. At run-time when an instrumented write is encountered inside the fragment cache, the instrumentation code is run. The instrumentation code identifies the write as a write to the stack, the heap or a global. If the write is to the stack, BODE looks up the stack frame that the write references. The debugging information allows BODE to split the stack frame into its corresponding variables. In the case of globals, again the debugging information determines what variable is being accessed. The general algorithm is shown in Algorithms 3, 4, and 5.

Algorithm 3 The BODE instrumentation algorithm (`bode_instrumentation`)

```
Input: pc : void * {The PC at the point of the write}
Input: loc : void * {Effective address of the write}
1: $loctype \leftarrow$ loc_table_lookup($loc$)
2: if $loctype =$ stack then
3:     check_stack($pc$, $loc$)
4: else if $loctype =$ heap then
5:     check_heap($pc$, $loc$)
6: else if $loctype =$ global then
7:     check_global($pc$, $loc$)
8: end if
```

Algorithm 3 shows the top-level dispatch, a function called `bode_instrumentation`. The PC where the write occured and the effective address of the write are the input to the algorithm. The metadata supplied by Metaman gives the location of global data, as well as the location of memory-allocating functions such as `malloc`. The globals' data ranges and the `malloc`'d memory are held in a splay tree for quick lookup at run-time.

Once the location type of the write is determined, the appropriate monitoring function is called — either the stack, heap, or global. The algorithm for checking the stack is shown in Algorithm 4. Algorithm 5 handles the heap and globals.

In the case of writes to a stack address (Algorithm 4), first BODE uses the effective address to

Algorithm 4 Code for the check_stack

```
Input: $pc$ : void* {The program counter of the write.}
Input: $loc$ : void* {The effective address of the write.}
1: $fdata \leftarrow$ lookup($loc$) {Frame data lookup}
2: if frame not found then
3:     return
4: end if
5: for all $vrange \in fdata$ do
6:     if $loc \geq vrange.low$ and $loc < vrange.high$ then
7:         $pc\_hist \leftarrow$ lookup($write\_table$, $pc$)
8:         $cksum \leftarrow$ stacktrace()
9:         if $pc\_hist == \emptyset$ then
10:            update($write\_table$, $pc$, $loc$, $cksum$)
11:        else
12:            if $cksum \neq pc\_hist.cksum$ then
13:                update($write\_table$, $pc$, $\emptyset$, $cksum$)
14:            else if write is to a different variable then
15:                error() {Signal an overrun}
16:            else
17:                update($write\_table$, $pc$, $loc$, $cksum$)
18:            end if
19:        end if
20:    end if
21: end for
```

Algorithm 5 Code for the check_heap and check_global

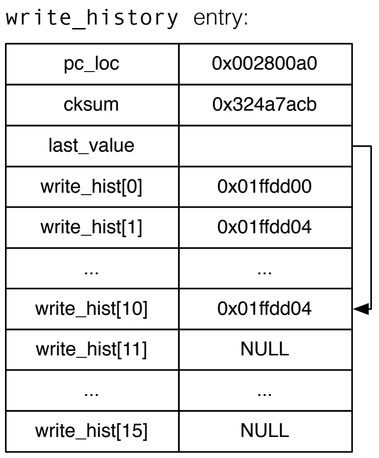
```
Input: $pc$ : void* {The program counter of the write.}
Input: $loc$ : void* {The effective address of the write.}
1: $pc\_hist \leftarrow$ lookup($write\_table$, $pc$)
2: if $pc\_hist == \emptyset$ then
3:     update($write\_table$, $pc$, $loc$)
4: else
5:     $cksum \leftarrow$ stacktrace()
6:     if $cksum \neq pc\_hist.cksum$ then
7:         update($write\_table$, $pc$, $\emptyset$)
8:     else if write is to a different variable then
9:         error() {Signal an overrun}
10:    else
11:        update($write\_table$, $pc$, $loc$, $cksum$)
12:    end if
13: end if
```

determine which stack frame the write is targeting (line 1). The frame data, *fdata*, contains a list of offsets, *vrange*. The offsets correspond to individual automatic variables on the frame of that function, as determined by the DWARF data from Section 5.2.1. If the frame cannot be found, BODE conservatively returns (line 3). Otherwise, it iterates over the ranges in the frame data to determine which variable the write accesses. Once the variable is found, the history of writes is looked up in the `write_table` hashtable. Each entry in the `write_table` contains the PC of the write, which acts as the key to the hashtable, the most recent checksum, and a list of the most recent write locations. The layout for a `write_table` entry is shown in Figure 5.2. BODE maintains the last 16 writes for monitoring and future heuristics. However, for the algorithm presented here, only the most recent write is necessary.

If the `pc` does not have any previous writes (line 9), then it is updated with a new entry based on the current write. If it is not the first write (line 11), a new checksum of the call stack is computed by a call to `stacktrace()` (line 8), and checked against the checksum in the history (line 12). If the checksums do not match, the history is reset (line 13). If the checksums do match, the variable is compared against the variable in the history (line 14), if they do not match BODE has detected a case of a potential buffer overrun, and signals and error (line 15). Otherwise, it simply records the write and continues execution.

The checksum calculation done by `stacktrace()` is an important step because typically buffer overruns occur because consecutive writes from a loop over the data. Often such a loop is abstracted into its own function, and if that function is called from different call sites, the series of writes will be different. To eliminate such false positives, BODE checks for changes in the call stack. The checks are handled by creating an XOR hash of the call stack at the point of the write. An example testing this case is discussed in Section 5.2.3.3. The checksum is computed by `libunwind`, a stand-alone repackaging of a tool to walk the stack of a program that conforms to the GNU ABI. `stacktrace()` walks the stack and includes each stack frame's return address into the checksum. Therefore, if the dynamic call stack is altered, the change is reflected in the checksum.

The algorithm for the heap and global data (shown in Algorithm 5) is very similar to the algorithm for the stack. However, because heap and global is already separated into individual variables



`write_history` entry:

pc_loc	0x002800a0
cksum	0x324a7acb
last_value	
write_hist[0]	0x01ffdd00
write_hist[1]	0x01ffdd04
...	...
write_hist[10]	0x01ffdd04
write_hist[11]	NULL
...	...
write_hist[15]	NULL

Figure 5.2: An entry of the write history. Values are from execution immediately before the buffer overrun from the example in Section 5.2.3.1

either by the compiler in the case of globals or the memory system in the case of the heap, the particular variable being accessed is already known.

BODE identifies writes at run-time, and instruments them to identify when consecutive writes at a given text address cross a variable boundary. However, it is not always possible to identify when a series of writes is intended to be consecutive. For example, if a programmer wrote a loop that manually initialized some data through a pointer, then changes the pointer to point to the next data structure on the stack, BODE would signal a false positive. A corollary to this limitation is that BODE will not correctly identify a singly-occurring out-of-bounds write, which might occur as the result of integer overflow or format string error [27]. The process of detecting overruns and their limitations are illustrated in the examples discussed in the next section.

5.2.3 Examples

This section illustrates specific examples of BODE's behavior. The test cases are designed to ensure that BODE correctly catches typical memory overruns and to show the cases where BODE fails to catch overruns. Table 5.2 shows an overview of the test cases, each with an input that causes an overrun, and another input that does not. The baseline program is described below, followed by

```c
/* Simple buffer overrun. */

int x; /* source of potential overrun */

void buffer_init( int* stp ) {
  int i;
  for ( i = 0 ; i < x ; i++ ) {
    *stp=0;
    stp++;
  }
}

int main(int argc, char* argv[]) {
  char arr2[40];
  int arr1[10];
  /* Initalize counter to user input. */
  x = atoi(argv[1]);
  buffer_init(arr1);
  return 0;
}
```

Figure 5.3: Source code of base.c.

Name	**Description**
`base.c`	Baseline example.
`adj_type.c`	One type allocated into two datastructures on the stack, then both initialized.
`call_stk.c`	One type allocated on the stack, through a different call chain.
`sngl_write.c`	A single out-of-bounds write to memory.
`malloc.c`	A heap-based overrun

Table 5.2: Table of BODE example programs.

the alterations made for each individual testcase. All the testcases are variations on the base test case, shown in Figure 5.3. The baseline and single write examples walk through cases where an overwrite occurs. The adjacent write and call stack examples walk though cases where an overwrite does not occur, even though the series of writes is identical to the baseline case where an overwrite does occur.

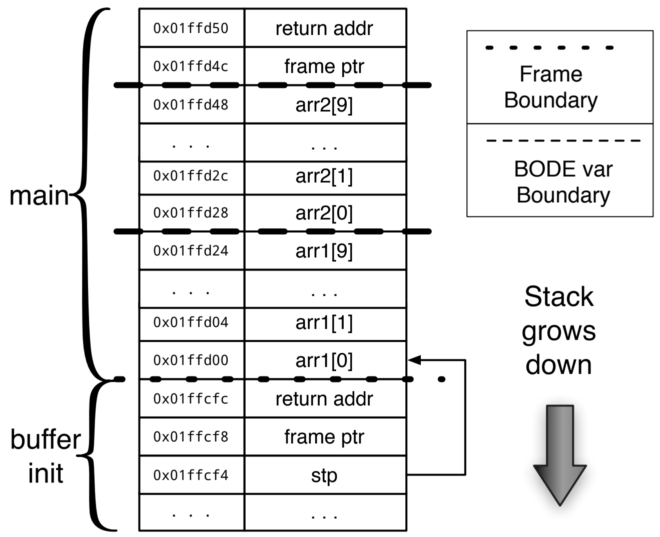


Figure 5.4: Stack layout of the example program.

5.2.3.1 Baseline

The baseline example illustrates the detection mechanism of BODE. It contains a potential stack overrun, depending on the value of a global variable, `x`. The example consists of two functions, `main()` and `buffer_init()`. The `main()` function declares two arrays on the stack, `arr1` and `arr2`. The initialization function, `buffer_init`, takes a pointer to `arr1` as a parameter.

The `buffer_init()` function contains the potential stack overrun. Figure 5.4 shows the layout of the stack as created by `gcc` when `arr1` is passed as the parameter to `buffer_init()`. The `buffer_init()` function initializes the data based on the integer `x`, which is passed in on the command line. In the case where $x > 10$, the `for` loop overwrites `arr2`, the next variable on the stack. If the value for `x` is large enough, it will overwrite the return address causing a fault.

BODE detects the overrun in this example by watching the writes in `buffer_init()`. Specifically, the overrun occurs during the initialization writes on line 8 of Figure 5.3. When targeting x86, the assignment on line 8 is translated to a `mov` instruction:

```
0x002800a0; mov    DWORD PTR [eax],0x0
```

The `mov` instruction causes the buffer overrun when it initializes past `arr1` into `arr2`. For BODE to identify the overrun it must observe the point where the write crosses over from the variables memory boundary. As discussed in Section 5.2.2, Strata is responsible for instrumenting all instructions that write to memory, including `mov` instructions that write memory, such as the one above. During translation, Strata adds a call to an instrumentation function, `bode_instrumentation`, whose semantics were given in Algorithm 3. The call is inserted into the fragment cache immediately before the `mov` instruction to identify overruns before they occur.

When the example program is executing from the fragment cache, just before the translated `mov` instruction is executed, `bode_instrumentation` is invoked. For this example, if the `mov` instruction was originally located at address `0x002800a0` and `eax` holds the address `0xbffd3000`, then the `bode_instrumentation` would be invoked with the parameters `bode_instrumentation(0x002800a0,0xbffd3000)`. Once `bode_instrumentation` is invoked, the effective address of the write (`0xbffd3000`) is identified as a stack reference (Algorithm 3, line 2), pointing to `main`'s stack frame, and therefore `check_stack` is invoked (line 3).

The `check_stack` function (Algorithm 4) looks up the stack frame information from the stack address (line 1). Each variable in the stack frame is encoded as offsets from the base of the frame. The frame address is added to the offsets to determine the range of variable (`vrange.low` and `vrange.high`, line 6). The function then iterates through the function's stack variables (Algorithm 4, for loop starting on line 5). In the case of the `main` function's stack frame, there are three variables, `arr1`, `arr2`, and the frame pointer and return address (counted as one variable because they are not source-visible). The first time the instrumentation is executed the history for the PC `0x002800a0` is empty, and therefore it simply updates the write information corresponding to the PC (Algorithm 4, line 10).

On each iteration through the `for` loop in the the example program, the instrumentation function is called each time `mov` instruction at `0x002800a0` is executed. The write table is updated on each iteration (Algorithm 4, line 17) because it contains the same dynamic call stack (line 12) and is writing to the same variable range (line 14). After the tenth iteration, the `write_history` entry

```
/* Simple buffer overrun. */

int x; /* source of potential overrun */

void buffer_init( int* stp ) {
  int i;
  for ( i = 0 ; i < x ; i++ ) {
    *stp=0;
    stp++;
  }
}

int main(int argc, char* argv[]) {
  int arr2[40];
  int arr1[10];
  /* Initalize counter to user input. */
  x = atoi(argv[1]);
  buffer_init(arr1);
  buffer_init(arr2);
  return 0;
}
```

Figure 5.5: Source code for the adjacent type example.

for address `0x002800a0` has the values of the ten writes, as shown in Figure 5.2. On the eleventh iteration, when `x` is greater than 10, the effective address passed to `bode_instrumentation` is `0xbffd3028`, which crosses into the next variable, `arr2`. The algorithm detects this condition (line 14), and signals that an overrun has occurred (line 15).

5.2.3.2 Adjacent Type

The adjacent type example slightly alters the baseline example to allocate two buffers of the same type on the stack, instead of differing types. This example is designed to show how BODE does not signal an error for normal initialization even when the series of writes is identical to the series of writes that signaled the error in the baseline example. The source code of the adjacent type example is shown in Figure 5.5. The adjacent type example differs from the basic overrun in two small but important details. First, instead of arrays of two different types of variables on the stack, there are two arrays of the same type, `int`. Second, instead of making a single call to `buffer_init()`, it is called with both arr1 and then arr2 as arguments.

When the adjacent type example is executed, as with the baseline example, the write to memory on line 8 of Figure 5.5 is instrumented so that bode_instrumentation is called immediately before the write. If the initialization size, x, is given a value greater than 10, like in the baseline example, BODE will correctly signal an overrun. However, if the value of x is initialized to 10, there is no overrun, and BODE should not signal an error. In that case the series of writes is identical to the series of write in the baseline (and shown if Figure 5.2). After the tenth iteration, the first call to buffer_init in the example returns, and then the second call to buffer_init is made, just before the first write to initialize arr2, the instrumentation function is called with the parameters bode_instrumentation(0x002800a0, 0x01ffd28). As with the baseline example, the write is identified as a stack reference (Algorithm 3, line 3), and check_stack is called. In check_stack (Algorithm 4), the frame data is obtained (line 1), which points to the destination of the memory write, the main function's stack frame. The lookup of the write_table, returns the write entry for the PC address 0x002800a0, which is holds the data of the previous writes to arr1, as shown in Figure 5.2. However, when the algorithm compares the checksum to the stored checksum (line 12), they do not match, because the return address for the current function is different. Therefore, the algorithm assumes that the write is to a new variable, and clears the write history (line 13).

This test case illustrates BODE's ability to identify overruns even if the linear sequence of writes could legitimately occur in the program. BODE observers the series of writes at an individual mov assembly instruction, in this case the mov in buffer_init. The series of writes seen at that instruction are the same in this example if x is set to 10 (i.e., no overrun) as the base example if x is set to 20. However, because BODE monitors the call stack (line 13 in Algorithm 4), BODE correctly does not indicate an error.

5.2.3.3 Call Stack

The call stack example in Figure 5.6 slightly alters the previous example, Adjacent Type. This example calls the buffer initialization through a simple wrapper function, thus changing the dynamic call stack.

```c
/* Simple buffer overrun. */

int x; /* source of potential overrun */

void buffer_init( int* stp ) {
  int i;
  for ( i = 0 ; i < x ; i++ ) {
    *stp=0;
    stp++;
  }
}

void passthrough(int* stp) {
  buffer_init(stp);
}

int main(int argc, char* argv[]) {
  int arr2[10];
  int arr1[10];
  /* Initalize counter to user input. */
  x = atoi(argv[1]);
  passthrough(arr1);
  passthrough(arr2);
  return 0;
}
```

Figure 5.6: Source code of the altered call stack example.

As with the previous two examples, if x is initialized with a value of greater than 10. An overrun occurs in the first invocation of buffer_init and BODE signals an error. However, if the value of x is 10, like the adjacent call example, the series of effective addresses instrumented by the write at address 0x02800a0 is the same as the baseline example, and shown in Figure 5.2. Once arr1 has been initialized, the example returns to the main function, and then calls passthrough, which is just a wrapper function for buffer_init. When the first write is made to arr2, BODE does not signal an overrun. The mechanism by which it identifies that the write is correct is the same as the adjacent call example: the call to stacktrace on line 8 of Algorithm 4 returns a different value than the one stored in pc_hist (line 12), causing the value to be reset (line 13). However, the reason is subtly different. The direct return value of buffer_init is the same in both cases, however, the return value higher up the call stack differs, allowing BODE to infer that no overrun has occurred.

This example again illustrates the call stack monitoring. Because the call stack differs from the write of one variable to the next, BODE correctly identifies the change and resets the series of writes that it is monitoring. The monitoring is not perfect. In the case of certain optimized recursive sequences, discussed more in Section 5.2.4, BODE can not search deeply enough down the stack and yields a false positive.

5.2.3.4 Single Write

The final example is shown in Figure 5.7. It is designed to illustrate the limits of BODE. Unlike the previous examples, it does not call the buffer_init() function, and instead makes a single write to a location in memory. The access is unsafe because the effective address of arr1[12] is in arr2's memory. However, because there is no write history, BODE does not detect that there is a memory problem. As with previous examples the bode_instrumentation function would be called, and the write_table entry for that PC will be empty, and BODE will assume that the first write from that PC is valid. BODE makes that assumption because it is designed to detect buffer overruns—code that starts in valid memory space and overruns it to adjacent memory—not general memory safety violations. In fact, when BODE uses the Eliminate ESP/EBP optimization

```
/* Single write example */

int x; /* source of potential overrun */

void buffer_init( int* stp ) {
  int i;
  for ( i = 0 ; i < x ; i++ ) {
    *stp=0;
    stp++;
  }
}

int main(int argc, char* argv[]) {
  char arr2[40];
  int arr1[10];
  /* Initalize counter to user input. */
  arr1[12] = 1;

  return 0;
}
```

Figure 5.7: Source code of the single write example.

discussed in Section 5.2.5, BODE might not instrument that memory access at all, depending on how it is optimized.

This is a false negative because there is no way to identify the pattern of writes that leads up to this write. Other source-level tools, such as CCured [26], might be able to detect this type of problem, but it requires more information, beyond the scope of the debugging metadata BODE uses.

5.2.4 Evaluation

To show the effectiveness of the BODE algorithm, it was evaluated on a number of benchmarks, including the micro-benchmarks , and real-world bugs occurring in every-day applications from the bugbench suite [84].

Table 5.3 shows the false positives and false negatives for BODE and MEDS on the sample programs. As discussed in Section 5.2.3, BODE correctly identifies the errors in all these examples except `sngl_write`.

	BODE		MEDS	
Name	F. Positive	F. Negative	F. Positive	F. Negative
`base.c`	No	No	No	**Yes**
`adj_type.c`	No	No	No	No
`call_stk.c`	No	No	No	No
`sngl_write.c`	No	**Yes**	No	**Yes**
`malloc.c`	No	No	No	No

Table 5.3: Table of BODE false positives and false negatives for the example programs.

	BODE		MEDS	
Name	F. Positive	F. Negative	F. Positive	F. Negative
bc	**Yes**	**Yes**	No	No
man	No	No	**Yes**	No
ncompress	No	No	**Yes**	No
polymorph	No	No	No	No

Table 5.4: Table of BODE false positives and negative for bugbench bugs.

Table 5.4 shows the false positives and negatives of real-world applications with real bugs taken from the bugbench application suite [84]. BODE produces no false positives or false negatives on all of the bugbench applications with the exception of `bc`. `Bc` is a arbitrary-precision calculator application that is able to handle both interactive input as well as batch processing. The bug exercised by bugbench is an overflow of the `sprintf()` function; `sprintf` is a vararg function similar to `printf`, but instead of printing the output to a file stream, `sprintf` copies the formatted string into a buffer provided as the first argument. Because `sprintf` does not do any bounds checking, the responsibility of correctly sizing the buffer falls to the programmer. The code that does the actual writes is, `vfprintf()`, a function responsible for writing out data all the `printf`-like functions. In GNU libc, this function is highly optimized and relies heavily on the use of macros. These macros result in convoluted code that results in multiple versions of the post-processed C code that makes it impossible for BODE's algorithm to identify that there is a series of writes. However, these types of optimizations are fairly rare, and usually occur only in highly optimized library code. Future versions of BODE can solve this problem by hooking-in a specialized version of commonly used library function that copy data, such as `sprintf`, `memcpy`, and `strcpy`. Replacing those functions at run-time can be easily implemented as part of BODE's instrumentation process.

Unlike the false negative, the false positive in the bc application occurs in application code. The false positive occurs in a scenario similar to the micro-benchmark `adj_type.c`. Two of the (arbitrary precision) number abstractions, `bc_num` are adjacent on the stack in the function `bc_mul`. However, the writes to these variables are in a recursive function, `_bc_rec_mul`. The recursion causes `libunwind` to incorrectly stop the stack walk prematurely, therefore returning the same value for the checksum. As a result, BODE incorrectly signals an error. One solution to this error is to avoid using `libunwind` and more actively reset the write history by instrumenting return instructions to reset the write history for the writes in a given function. However, this solution would incur a much larger performance overhead than the current lazy solution, and such false positives are fairly rare and are easily recognized by experienced programmers.

5.2.5 Performance Optimizations

Another important factor in the performance of BODE is run-time overhead of the instrumentation and analysis. If BODE causes too much slowdown, application developers will be less likely to invest the time to use the tool.

Initially, testing was performed with the `ncompress` benchmark to inform the early design decisions. BODE was improved by a series of optimizations that built off of each other, presented in Figure 5.8.

Eliminate ESP/EBP. The eliminate ESP and EBP optimization alters Strata to not instrument write instructions of the form `mov [esp+off] xxx`. This optimization assumes that the program was written in an ABI-conformant manner, which is an acceptable assumption since BODE also assumes ABI-conformant debugging metadata. In a typical ABI-conformant program, the `esp` and `ebp` are used to index into the stack, and therefore an access of the form `[esp+off]` is constant relative to the stack frame, and therefore cannot be used to index into an array.

Eliminate pushf/popf. Another method of improving instrumentation performance is the reduction of overhead for adding the instrumentation. Previous work has shown that eliminating the save and restore of the x86 `EFLAGS` can result in significant performance wins [60, 119]. However, because the saving and restoring of `ELAGS` represents only a small fraction of the instrumentation,

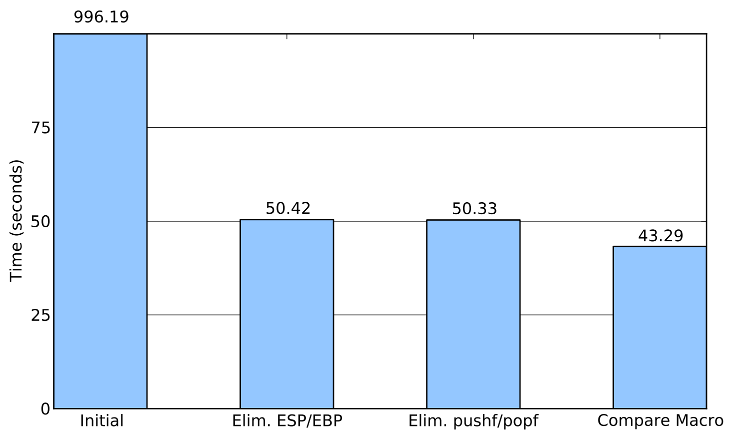


Figure 5.8: Performance of BODE on NCompress with first set of optimizations.

this optimization showed only very marginal benefits.

Compiler Optimizations. Finally, BODE leveraged the gcc compiler optimization (`-O3`), along with aggressive inlining of the underlying data structure searches. A mapping of the heap is kept in a splay tree, a binary search tree that adjusts to improve lookups of recently requested nodes [115]. The efficient lookups of recent nodes is valuable to take advantage of the temporal locality of the memory references. To reduce function call overhead BODE used an optimized comparison macro for the frequently-invoked node comparison operation.

With those basic design decisions, BODE is a fully functional system, but it still requires further tests of performance. BODE was tested for its run-time performance on the SPEC2006 benchmark suite [55]. The experiments were performed on a 1 GHz Opteron system running Linux with 2GB of RAM.

Figure 5.9 shows the results of running the SPEC2006 benchmarks under BODE. The values are normalized to native execution to show the overhead introduced by BODE. The leftmost bar indicates the performance of BODE with the optimization described above. Even with those optimizations, performance is still very slow. To improve performance further BODE uses a more in-depth optimization to reduce the number of context switches and the analysis required. The

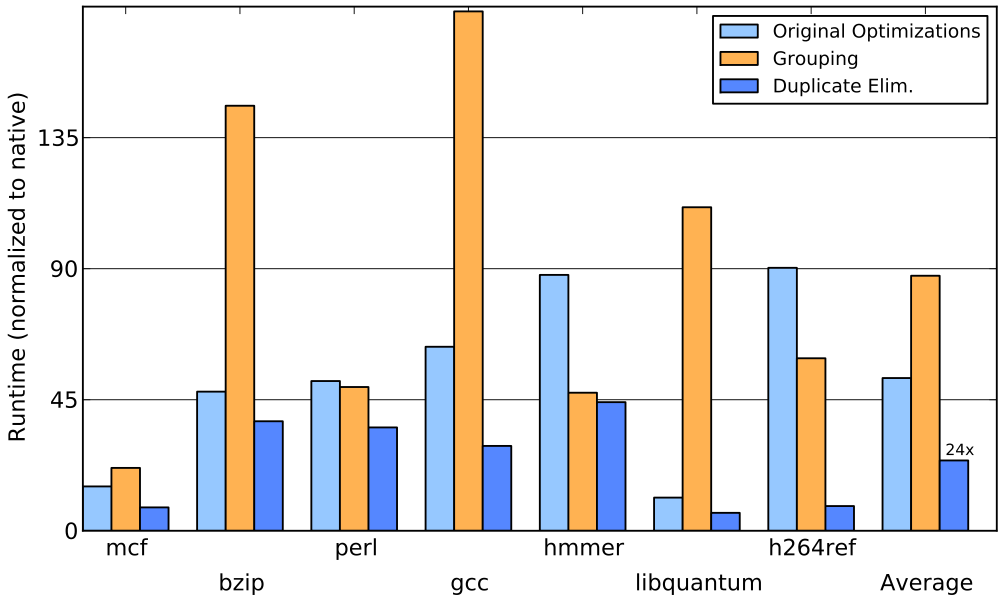


Figure 5.9: BODE performance on SPEC2006 benchmarks

analysis needed for BODE depends on three pieces of data. First, the PC of the write instruction, second, the effective address, and finally, the state of the stack and heap. By collecting the PC and effective addresses in a light-weight buffer, BODE can reduce the number of context switches it needs perform. The buffer must be emptied at the end of functions when stack information might be destroyed, and at system calls and memory function calls. However, in practice the buffer can be filled completely before needing to be emptied. The middle bar of Figure 5.9 shows the performance of combining these writes. The performance is highly volatile. The performance is better than the original implementation for some benchmarks and worse on others. While buffering reduces the number of heavyweight context switches, it increases the instruction count of instrumentation, including adding a branch. The performance of buffering is highly variable, most likely due to second-order effects including branch-prediction pollution.

While buffering does not improve performance, it does give an opportunity to reduce the amount of analysis performed. Simple profiling of BODE showed that approximately 80% of the run was spent in the BODE instrumentation algorithm. Further, there is significant duplication in the PCs and addresses being analyzed. By creating a simple per-buffer cache that stores the result of the analysis, BODE can eliminate analysis of duplicate entries. The execution time of the version that eliminates cached duplicates is shown as the final bar of Figure 5.9. Using this final optimization BODE is able to achieve average performance of 24x.

5.3 BODE/MEDS

BODE offers an important set of features to programmers who want to identify and avoid buffer overruns. However, there are other more comprehensive memory protection schemes which trade performance for a high level of accuracy and a wide range of protection against a variety of errors.

One such system is the Memory Error Detection System (MEDS). MEDS offers a holistic solution to memory protection, tracking objects and referents across the entire running of the program. MEDS operates on raw binaries with no access to source code or debugging information. As a result, some heuristics and profiling must be used to correctly identify variable and object bound-

```
1  8048190      93 FUNC GLOBAL frame_dummy FUNC_SAFE USEFP RET 80481ec
2  8048196       0 INSTR BELONGTO 8048190
3  8048196       2 INSTR DEADREGS  EFLAGS ZZ mov     ebp, esp
```

Figure 5.10: MEDS annotation.

```
<function id="0x150a" name="frame_dummy"
          address="0x8048190" end-address="0x80481ed"
          meds-flags="FUNC_SAFE|USEFP" >
  <asm-inst id="0x1512" address="0x8048196" dead-regs="EFLAGS">
    mov     ebp, esp
  </asm-inst>
</function>
```

Figure 5.11: MEDS XML.

aries [59]. As a result MEDS can benefit from some of the features of BODE. As described in Section 5.2, BODE utilizes debugging information to determine the layout variables on the stack. MEDS operates without the assumption of any additional metadata, however if the metadata is available, it could take advantage of it in a manner similar to BODE.

5.3.1 MEDS Metadata Format

MEDS uses a custom human-readable annotation file. It contains newline-delimited records of IDAPro static analysis along with profile information collected by MEDS on previous runs. The metadata is indexed off of program addresses, which allows for easy access when the program is executing. Because the annotation is address-based, with one address per line, it is well-suited for integration into Metaman format (see Chapter 3). Much of MEDS metadata fits directly into Metaman's existing metadata types. Entries such as the `FUNC` annotation map directly to the `function` Metaman tag. Other MEDS entries such as `MEMORYHOLE` annotate the state of memory in the program, which is included to help map the state of the stack to facilitate shadowing memory.

Figure 5.10 shows sample data from the MEDS annotation file. The first entry records the function `frame_dummy`, and the next two lines give the metadata on an assembly instruction in the function. Line 2 indicates the instruction is part of the function, and line 3 gives the instruction and associated killed registers. When translated to XML, shown in Figure 5.11, the function information

Input	BODE	MEDS	BODE/MEDS
No Overrun (`10`)	No Error	No Error	No Error
Interframe Overrun (`14`)	Detected	**Not Detected**	Detected
Cross-Frame Overrun (`30`)	Detected	Detected	Detected

Table 5.5: Comparison of BODE MEDS and BODE/MEDS on different input sizes of base.c.

is incorporated into the `function` tag, and the specific instruction information is in the `asm-inst` tag. The `BELONGSTO` annotation becomes implicit by becoming a child element of the `function`. The actual assembly text is included as XML character data.

5.3.2 BODE and MEDS Integration

Both BODE and MEDS use some heuristics to perform their checks. Further, they each use different metadata, with MEDS relying on IDAPro-generated metadata while BODE requires the use of debugging information. IDAPro is unable to determine the exact layout of variables on the stack. Therefore, it treats entire stack frames as a single object, so different writes to a given stack frame are considered equivalent. This technique allows MEDS to stop stack-smashing attacks that attempt to overwrite a return address, because IDAPro identifies the return address and separates it from the local variables. However, it is not able to separate one variable on the stack from an adjacent variable. BODE's use of debugging metadata, however, provides the layout of the entire stack frame.

To show a proof-of-concept on how Metaman improves the process of adding features to tools and migrating features from one tool to another, the BODE metadata was integrated into MEDS to refine the stack frame information. The BODE/MEDS integration was tested on three varying inputs of the example code shown in Figure 5.3. By initializing the value of `x` with inputs of `10`, `14`, and `30`. The results for BODE, MEDS, and BODE/MEDS for these inputs is shown in Table 5.5. With an input of 10, a buffer overrun does not occur. When the input is 14, an overrun does occur, but it is contained within the stack frame. That is, `arr1` overruns into `arr2`, but not into another stack frame. Finally, an input of 30 overruns both buffers, and into the stack frame.

As Table 5.5 shows, each system correctly identifies the non-erroneous case, and each system correctly identifies the case where the overrun crosses the frame boundary. However, in the case where the overrun only enters into another variable in the same frame, MEDS does not detect the error. BODE correctly identifies the error, and the combined MEDS/BODE system correctly detects the error as well.

5.4 Genprog

The integration between MEDS and BODE shows the flexibility of Metaman and the availability of metadata for systems that were not originally designed to interact with Metaman. However, both MEDS and BODE were projects based on the same SDT framework.

To further show the flexibility of Metaman, and the steps needed to integrate Metaman into different systems, this section describes how Genprog could be adapted to use Metaman. Genprog is a novel tool developed in collaboration with University of Virginia and University of New Mexico. It is not an SDT-based project, and was developed independently of Metaman, using their own tools and workflow.

5.4.1 Genprog Metadata

Finding and fixing errors in programs is a difficult task and constitutes a large portion of the application development life-cycle. Traditionally bugs are fixed by a combination of testing and receiving bug reports. Once the programmer has the bug report, the bug is reproduced in a test environment, the programmer localizes the bug, and then modifies the source code. Finally, a new version of the software is released. As part of the Helix Project, Forrest et al. have developed a genetic programming approach to fixing bugs. Once a test case exposing the error has been found, they use a genetic algorithm to alter the program structure until a version of the program is found that passes all the test cases (as well as the failing case) [43, 126, 128].

The genetic programming approach is a good use case for integrated metadata. It requires custom metadata at many points, including run-time and during testing. While Forrest et al.'s experimental setup used CIL to create instrumented versions of the program, a version based on SDT would be able to instrument and patch a binary, potentially without altering the original binary.

Genprog needs numerous pieces of metadata to run. It primarily operates on the program's abstract syntax tree (AST). For the Genprog tool, the AST is collected by CIL. Due to the tree structure of the AST, it can directly be converted to XML and stored within Metaman's individual `compilation-unit` tags.

Another key part of the genetic programming cycle is the test cases for the program. The test cases serve as the fitness function for the genetic algorithm – the more test cases that pass (including the original broken test case), the more fit the solution. The current implementation of Genprog uses shell scripts to run the test cases and report the number of successes, however to scale to larger systems a more formal testing system will be needed. Mozilla's Testopia, such a system, offers XML results, and also includes other important details such at run-time, which might be useful for improving the fitness function.

A related piece of metadata used by Genprog is the test case coverage – specifically the coverage of the failing test case. That data is used to ensure that changes made to the program are related to the failing test case. The coverage data is stored as a list of identifiers, mapped to the corresponding CIL statement node. As Metaman XML tags are uniquely addressable, these map to the id of the XMLized CIL statements.

Many of these pieces of metadata have been discussed earlier, however merging them and making them available for a system like Genprog is challenging. For example, the easiest way to map CIL statements into Metaman's schema is to simply take CIL's intermediate tree format and represent it as XML, adding new XML elements to Metaman as necessary. However, the ideal representation would be fully integrated into Metaman, using pre-existing standarized data formats, such as a GCC-XML representation or XML Graph representation, because then existing tools could utilize metaman with little or no changes. Here, Metaman's flexibility gives it a significant advantage. Because of Metaman's flexible XML schema, it can include a simple port of the CIL's data format, but also allow for switching to a more formal representation in the future, when developers are able to generate scripts to convert from CIL's native representation into a more well-established format.

5.5 Related Work

Many techniques have been employed to assist developers with memory errors in programs. Original work on protecting buffer overruns focused on the stack. Stackguard [28] and Libsafe [13] protect entire stack frames, but do not identify buffer overruns that occur between variables within the stack frame.

Projects such as CCured [26], Splint [39] and DFI [23] work at the source-level to move memory-unsafe languages closer to memory-safe languages. CCured achieves the goal of memory safety by using static analysis and programmer annotations to identify provably-safe pointer usage, and then also re-writes the code that is not provably safe so that it can be checked at run-time, essentially turning unknown pointers into "fat" pointers which can be followed and checked at run-time. Splint takes an approach based purely on static analysis, checking for common code that can result in unsafe run-time behavior, and also allowing the programmer to insert annotations, allowing the programmer to give checkable assertions as to how the memory is intended to be used. DFI requires no annotations, and uses reaching definitions to identify instructions that are allowed to safely write a value. Then it inserts instrumentation to enforce that write safety.

As run-time systems have advanced and reduced the amount of overhead they incur, they have become more practical for many applications, including checking for memory safety. Examples of run-time checkers include Valgrind's Memcheck [98, 112] and Annelid [97], MEDS [59], Electric Fence [101], Rational Purify [15], and Diehard [16]. These tools are based on a variety of run-time systems. Memcheck and Annelid use Valgrind as their base. Memcheck utilized Valgrind's shadow memory to identify uninitialized reads, as well as writes that occur outside any valid memory region. Annelid takes a more heavy-weight approach, following pointer usages and memory accesses to identify when a pointer is being used unsafely. MEDS also tracks pointers and memory references, however it also employs static analysis on the binary prior to execution to help it to correctly monitor the stack, and also eliminate unnecessary run-time instrumentation. Diehard takes the innovative approach of randomizing memory accesses to identify them sooner.

In contrast to the techniques described here, BODE uses a combination of debugging information and run-time analysis to protect a specific class of memory errors: overruns. This allows BODE to be faster than full-protection run-time systems like Annelid and MEDS, but still identify buffer overruns better than tools like Memcheck and Purify.

Chapter 6

Future Impact

This work has shown metadata to be a vital aspect of software development. Furthermore, this work has shown that provision of facilities for improved communication of and sharing of metadata between software development components can help software developers address some of the pressing problems such as security, reliability, efficient composability of modules, to name a few. Once general tools for manipulating program metadata become ubiquitous throughout the software development toolchain, greater opportunities to improve software development will emerge. Metaman's design and functionality serves as a first step towards such improvements.

As software systems continue to evolve and improve, it is increasingly important to have flexibility in how programs are built, optimized and maintained. Traditionally, software development and deployment have occurred in a very linear manner. A program is written, built, debugged, and tested on a development machine. Then it is deployed and run on the target machine. Modern systems have more complex interactions that blur the traditional separation between development and deployment. Dynamic linking allows most programs to defer resolving the underlying library call until the program has been invoked on the target system. Thus, all the code to be executed by a program is not necessarily available until the program is run. Further, VM-based languages like Java and C# have moved much of the optimizations phases to run-time as well, using JITs instead of compile-time optimization phases [7]. This trend continues with the widespread use of scripting languages such as Javascript and PHP, which power large-scale web-based applications. In the case of in-browser Javascript, the code is not even parsed until it reaches the target platform. However, as languages like these become more popular, they are typically JIT compiled for performance [120]. As the software development community continues to find innovative ways to distribute software and data, program metadata is going to play an even more important role in building practical systems.

An example of how metadata use can increase in the current web-centric software environment is the efforts to allow high-performance binary code to be executed within a browser sandbox. Google's Native Client (NaCl) project has the goal of allowing native executable code to run securely—without executing system calls or transferring control to unauthorized code [132]. NaCl requires compiler support to implement special handling of indirect branches and to ensure the binary is amenable to static analysis. At run-time, the binary is statically checked to ensure it meets NaCl's requirements, and then it is linked with NaCl's own libraries that ensure that control flow is not altered, and that system calls are only made through NaCl's interface. Tools such as NaCl and other web-centric tools that offer features such as dynamic patching or online plug-in installation provide valuable benefits to users, but at the same time they make security and performance even greater concerns.

The solution to manage these security and performance concerns while still allowing the benefits of web-based software delivery is to use metadata to ensure that programs have the information that need to verify that the code being downloaded is trusted and unaltered, as well as provide metadata about how to run that code efficiently. The next sections discuss how Metaman can be extended and integrated into other tools to create the next generation of software development tools.

6.1 Software as Metadata

In demonstrating that ubiquitous program metadata leads to better software tools, SDT tools and metadata were chosen because of the practical needs of SDT systems and their acute need for program metadata. However, as discussed in Chapter 2, there are many tools that create a wealth of metadata.

The source code repository is a tool discussed in the related work section, and is an important tool for the future direction of Metaman. Code repositories track all changes to code as a project progresses. The changes themselves are valuable metadata as are code branches, commit comments and other metadata stored by the repository. That information provides insight into programmer intent [22], as well as a linear history of the program. In addition, the raw source code held in the repositories can also be considered system metadata[1]. Programming styles such as Literate Programming attempt to decouple how the program is viewed and understood by developers from how the program is presented to the compiler, with the goal of making the software artifacts more geared toward developers and maintainers, rather than the compiler [76]. Ultimately, Metaman can serve a similar purpose, presenting programs in the form most useful to the tool, rather than forcing the tool to understand the native format of the code. Different tools take different abstractions of the program. It is common practice for optimizing compilers to alternate between AST [3] passes, SSA passes [30], and RTL [6] passes, as is needed by that particular phase of optimization. Similarly, if the entire program is considered metadata, each method for representing the data is just another piece of metadata. Therefore, the programmer is presented one view of the program, the compiler another, and each of these views can be completely abstracted from how the data is stored on the file system.

By taking such a view of the program, tools manipulating the program can alter the underlying representation or use an existing representation without disturbing the representation seen by the programmer. Figure 6.1 shows a possible version of dividing a software project into multiple views. The files with the "F" marker represent the smallest practical software entity, possibly a function or class. The rounded boxes represent parts of the program relevant to the current developer, similar to a "view" already available in some IDEs, though typically at the file level of granularity. The ovals represent a typical software engineering module or API – a group of functions working together in a (logically) tightly coupled way, often currently represented by a directory hierarchy. And finally, a compiler/architecture view, which is how the system is presented to the compiler, optimized for code paths and (temporally) related code—currently an individual file in most development systems.

[1]This section blurs the distinction between "data" and "metadata," but for this discussion, "metadata" is used to refer to all data about the program, including the program itself.

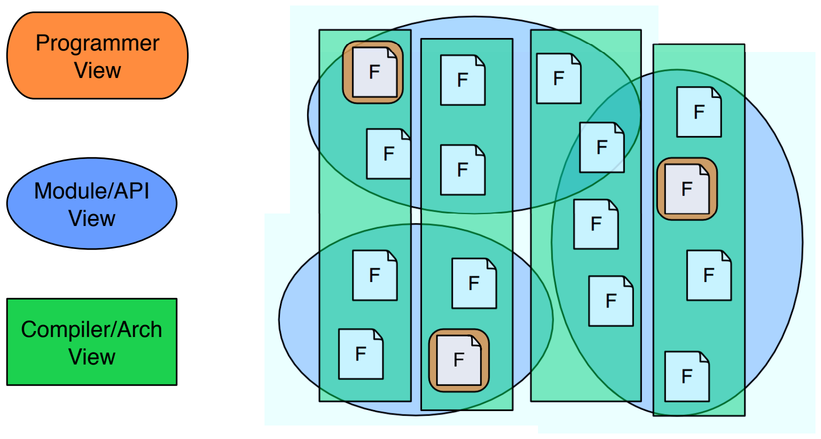


Figure 6.1: Possible layout of multiple views of a software project.

Decoupling these ways of representing the program allows for greater flexibility for tools to improve the code without intervention by the programmer. Instead of optimizing single file modules, or by investing in computationally expensive whole program (link-time) optimizations, the system could choose its own optimization modules based on usage and locality.

Choices that are intuitive to the programmer are often not optimal for the system as a whole. The prototypical example of this phenomena is register allocation. Early versions of the C programming language allowed the programmer to specify which variables they wanted allocated to a register, via the `register` keyword. However, as register allocation algorithms advanced, programmers found that the compiler could almost always choose a better set of allocated variables, and most modern implementations of C simply ignore the `register` keyword [103].

Similarly, in terms of module grouping, McFarling has shown that using feedback-directed optimizations, library code can be "packed" so that temporally local code is also local on disk and in memory [92]. This technique can reduce disk reads as well as reduce memory and I-cache pressure. These optimizations are underpinned by the fact that oftentimes an automated tool can achieve better results than programmer intuition.

Using such a system, where the "view" of the program can be easily changed to suit the tool, it would be possible for each tool to have a customized view of the program. Such customization allows programmers to modify their environment to suit their own needs. However, going too far down the path of customized metadata, can create difficulties for tool writers who would be obliged to support all the views and interfaces that a programmer might want, and would increase the learning curve for adopting a new tool. However, most tools fall into one of a relatively small set of discrete categories. A good compromise between flexibility and ease of tool creation is to encourage tools to use a few interfaces developed for that type of tool, thus making easy to create and learn, while still giving the programmer flexibility.

6.1.1 Enabling Advanced Language Features

Another part of software development that has important interactions with program metadata is language design. The most straightforward way of adding general metadata at the language level is through annotations. Language annotations have been added post-hoc to languages. For example, Splint provides a mechanism to annotate C programs [39]. Similarly, C# [53] and Python 3 [121] offer built-in annotations as part of the language. These annotations—essentially programmer supplied metadata–have been used for a variety of language features, grouped by Kirner and Puschner into three categories: platform properties, CFG reconstruction, and program semantics [72].

New language features will continue to leverage these annotation tools, however prototyping and implementing new language features typically require support across the software development toolchain. Integrated metadata management can achieve the same goals for the toolchain that annotations have for languages. For example, if a language designer wants to add a new data type such as arbitrary precision numeric, annotations can be used to add the feature without large changes to the language syntax. However, the other parts of toolchain must support the new features. Arbitrary precision numerics can be supported as a built-in library, or can be inlined. Many libraries are already available, easing the implementation, however inlined code can have significantly higher performance. With effective metadata use, the design decision can be abstracted and included in intermediate formats so that it is possible to first support library calls and then re-optimize with inline execution without doing a full recompile. The next section discusses how to fully develop a flexible system to partially recompile only as necessary.

6.1.2 Metaman and Next Generation LLVM

Generally, current programming systems fall into one of two categories, “compiled” or “interpreted.” For compiled programs most of the work is done on the development system, and the run-time system can be very small. Interpreted systems defer much of the work done in a traditional compiler, typically to a JIT. As JITs become more advanced and compiled languages add more dynamic features, the line between them is beginning to blur. Metaman allows system builders to move towards a system of applying as much optimization as possible at the time when the information is available. This approach is a powerful strategy for addressing problems related to poor heuristics. Often heuristics are necessary because too little information is available, and it is difficult or impossible to recreate the information at run-time.

Tools such as the LLVM compiler and toolchain infrastructure are beginning to decouple individual phases and representations from specific compile-time or run-time implementations [80], making them useful for both compiled and interpreted languages. Further, LLVM provides support for extensible metadata in the language specification [81]. LLVM is also highly modular; it contains components for static compilers, linkers and runtime systems. By combining LLVM’s current metadata capabilities into a system such as Metaman, and leveraging LLVM’s modular set of tools, it would be possible to create a dynamic compilation system. This system would be similar to “staged compilation,” an idea proposed by Philipose *et al.*, which breaks the unnecessarily strict divide between compile-time and run-time systems [102].

Using these techniques, systems can be built using a component-based system along with robust metadata system such as Metaman. System builders can create systems that perform optimization as early as possible, while remaining flexible enough to reconsider and undo earlier decisions when new information supersedes earlier assumptions.

Consider the following scenario for such a system:

1. Compile the application into LLVM byte-code.
2. Based on previous profiling, emit optimized x86 code.
3. Package and deploy the application – x86 code, byte-code and LLVM runtime.

4. Reevaluate x86 code based on deployment specifics (i.e., specific processor model, memory and cache size); possibly re-emit x86 from byte-code.
5. Run program.
6. Evaluate performance based on profiling data and program input.
7. Re-optimize hot code, propagate profiling information back to the build environment.

Following these steps allows the program to be seamlessly improved throughout development and deployment with minimal programmer intervention. For example, specific binaries can be compiled and optimized for commonly used architectures, but intermediate code can distributed and JIT compiled for less common architectures, saving the need to keep binaries for every type of system. Such a technique provides flexibility to programmers, however, to make such a system practical, there must be facilities to deploy and verify code on the final target system.

6.2 Deployment, Configuration and Trust

The Metaman project has shown that a comprehensive metadata service is both valuable and instrumental to the development of cutting-edge software tools. However, to make the evaluation tractable, we simplified a number of the complexities facing software engineers today. This section discusses those simplifications, and how Metaman could be extended to be a practical system for a number of environments.

For most of the Metaman experiments, the development environment and the deployment environment were assumed to be the same. This assumption allows Metaman to ignore issues of verification, and eases deployment. However, most software is distributed to new systems, potentially with different configurations and instruction sets. Software deployment is an important part of the software process and can differ widely based on the type of software involved. For client-server, web-based, and "cloud"-based software, the developers usually have direct control over the server-side environment. For commercial "desktop" applications, software is sent to the user by DVD or the Internet into wholly unknown, potentially misconfigured or hostile environments. These systems must also be patched and updated to fix vulnerabilities and provide new features. With such added challenges, effectively using metadata is even more difficult, but it has even more potential benefits.

A major hurdle to using metadata in code distributed to users is trusting the platform. Client systems can host malware, or be directly attempting to alter or subvert the program being run. One method of handling these problems is to use the runtime system as a sentinel to ensure the fidelity of the environment [45]. Then the runtime system can load, decrypt, and use metadata without it being exposed or maliciously modified. Using the secure run-time system, Metaman can create a network communication channel to the metadata server and query information as necessary.

Currently, mobile computing platforms such as Apple's iOS and Google's Android offer "App" stores that provide centralized deployment and trust mechanisms. These stores use program metadata, including information on system resource usage that allows the system to add policy-based security features. However, because the stores are controlled by companies that distribute the operating system and not the developers of individual apps, it is impossible for the third party developers to directly leverage the advantages of the distribution platform.

Using Metaman as a basis for distribution and trust offers a number of advantages. By tracking metadata as the program is developed, Metaman is able to identify the files necessary for execution on a given platform. Further, Metaman's integration with the build process make it possible to create signed binaries and track their chain of trust. As discussed in Chapter 3 Metaman can also embed metadata in binaries, allowing for more robust trust mechanisms, such as proof-carrying code [5].

Chapter 7

Conclusion

Program metadata has been used to improve programs for the entire lifespan of computing, though most often it is generated, used, and discarded. However, software development has advanced, and tools have made increasing amounts of metadata available for more of the software development process. Today, program metadata is used in nearly every software development tool. However, very few tools exist to maintain and organize program metadata. The thesis of this dissertation is that the comprehensive collection and organization of program metadata across the software development toolchain can improve the software development process as well as the resulting applications. A design for comprehensive metadata management was created to support this thesis, along with a prototype implementation, Metaman.

To illustrate the value of program metadata, and provide analysis of how to properly gather, store and manipulate the data, a taxonomy was provided to document current uses of program metadata. This taxonomy also provides a framework for new and innovative metadata. Appendix A has the taxonomy encoded as an XML schema, a flexible format for specifying XML data. The schema allows programs to generate and verify data coming into and out of Metaman, and it also helps tool developers generate new metadata using the schema as a guideline.

Chapter 2 discussed the landscape of program metadata and software tools related to comprehensive program metadata management and Metaman. Metadata related projects have paved the way comprehensive metadata management. Metaman has adopted and expanded on many of the ideas presented in Chapter 2. Early projects that attempted to collect metadata across software development include the Montana project [88], JikesRVM [7], and Oberon. These projects are important predecessors to holistic program metadata, however they differ in that they are all tied to a platform and language. Montana allowed developers to create custom metadata, however only as a plug-in that had to be written in C++. Similarly JikesRVM and Oberon allowed access to program metadata via an API. However, use of the API is limited to the platform's language (Java or Modula, respectively). Metaman differs from these tools by following the UNIX philosophy of creating small programs that do single tasks and interoperate. As discussed in Chapter 3, Metaman is designed so that it can be used with any tool written in any language. Metaman uses a programming language neutral data format, and can therefore communicate with a large range of tools, and be quickly adapted to new tools.

The LENS project was another attempt to collect large amounts of metadata [90]. LENS leverages LLVM [80] a mult-language framework. However, the LENS project focused mainly on compiler metadata and on improving understanding of the effects of optimizations on performance. Metaman provides the ability to collect compiler metadata, and perform run-time performance monitoring—the two main thrusts of LENS. Metaman's goals are more wide-spread, not just limited to understanding performance. However, integrating LENS metadata and LLVM tools into Metaman is a long-term goal of the Metaman project.

Those tools offer valuable insights into how metadata can be used and categorized. However, the overall solution to the problems created by overwhelming amounts of program metadata is a comprehensive system for managing the metadata. Metadata can be generated at all points in the software development process and also consumed anywhere as well. Comprehensive metadata management systems must interact with the rest of the software development toolchain, track metadata as it is created and used, and they must allow system builders to create new tools and use cases as the needs arise.

With those goals in mind, the Metaman prototype was created as the first comprehensive metadata management system built. It is designed to be a system for research and experimentation into comprehensive metadata management. Metaman is integrated into the software build system to leverage the dependency metadata necessary to keep the metadata from becoming stale. Metadata is deposited into Metaman's database where it can be queried or updated, either at build-time or run-time. Details of the structure and usage of Metaman were given in Chapter 3. The primary design goals of Metaman are: flexibility, scalability and ease of use. Flexibility is necessary to allow for new use cases and new metadata to be quickly integrated into the system. Scalability is important because of the large amount of data that can be generated and used. Finally, ease of use is important for adoption and productivity of system builders.

Metaman provides a powerful platform to explore the use of program metadata, and test the design points on which it is built. When used in conjunction with Strata, a software dynamic translation system, Metaman is able to provide data to all points in the software development life-cycle. As the basis for many run-time tools, Strata has the ability to examine and alter a program as it is running. Because Strata amortizes the cost of translation, it can typically execute a program with only modest run-time overhead. The ability to examine and alter programs as they run is very powerful, but its practicality is limited by the myopic view that it has while the program is executing: when SDT tools examine code, they do so from a single point in the execution, with only information about the immediate binary instructions, little to no information about control flow, data analysis or other information readily available when the program was compiled. Combining the utility of Strata with the information collected by Metaman, there is a powerful opportunity to improve programs and reduce errors.

Chapters 4 and 5 examined the practical effects of increased availability of program metadata, leveraging software dynamic translation combined with Metaman. SDT systems have created exciting opportunities to improve programs by examining their behavior while they are executing. However, some information about the program is unavailable during execution, and often it is too costly in terms of time to recreate the necessary data. Persistent metadata addresses these challenges and provides mechanisms for even greater cohesion between compile- and run-time systems.

Chapter 4 examined the utility of program metadata in the areas of optimization and overhead reduction. Improving performance, particularly when a program is run under an SDT system is a ubiquitous goal. Keeping SDT performance as close as possible to native speed enables tools that might not otherwise be practical.

Using Strata as a platform, Metaman addressed the problem of dynamically identifying all possible indirect branch targets for a given branch. The key insight into the indirect branch handling improvements is that structured programs are explicitly designed to restrict indirect control flow to one of a set number of use cases. However, indirect control flow expressed by ISA-level indirect branches has no such restriction. Indirect branches can target any machine address, while in structured programs they are typically only used for switch tables, indirect calls and returns. Encoding that metadata and making it available at run-time allow the SDT system to handle indirect branches more directly, often by emitting a single indirect branch instead of a multi-instruction sequence.

Using the metadata for switch, indirect calls, and returns gathered at compile time, Metaman and Strata produced performance within 3% of native execution on average. On some benchmarks with a high rate of indirect branches, such as Xalan, there was as much as a 10% performance improvement. These performance gains increase the applicability of SDT systems, making it possible to use them in more tools and in more scenarios where performance is critical.

Chapter 5 focused on another important use of run-time systems: security and program understanding. In these areas the flexibility of Metaman played an important role in supplying and applying metadata across the toolchain. BODE was developed using Metaman, with the goal of improving understanding of buffer overruns. The key use of program metadata in BODE is that it leverages readily available debugging metadata to provide insight to the program previously reserved to source-level analysis. BODE uses the variable layout information from the debugging data to create a map of the stack and therefore help identify when overruns occur. Using BODE, a developer can avoid a large class of dangerous vulnerabilities, which can often be used as an attack vector for malicious software.

Chapter 5 also showcases the flexibility of Metaman illustrated by the integration of BODE with the MEDS project, using Metaman. MEDS was created using its own ad hoc metadata. The integration of that metadata into Metaman shows the flexibility and value of a generalized metadata framework.

In summary, this dissertation has shown the value of program metadata across the software development toolchain. It describes a comprehensive framework for collecting, storing and accessing program metadata. The prototype system, Metaman, demonstrates the value of long-term storage of metadata. As developers create new and even more advanced tools, they will continue to rely on metadata, and leverage the techniques presented here to produce the next generation of software development tools. As a result, the utilization of program metadata will increase as developers find new and interesting ways to leverage that data to improve their programs. Comprehensive metadata management tools have the potential to strengthen and enhance all future software development processes and environments.

Appendix A

Metaman XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Metaman Schema -
    This schema describes the layout of a proper Metaman XML document.
    This serves as a basis for storing program metadata for a wide variety
    of software development tools.

    The working URI for the Metaman schema version 0.5 is:
    http://www.cs.virginia.edu/~dww4s/metaman/schema/0.5/metaman.xsd


:author: Dan Williams
:copyright: 2011, University of Virginia
-->
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="0.1">
    <!-- General containers and structures -->
    <xs:element name="metadata-file">
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
              <xs:element ref="compilation-unit"/>
              <xs:element ref="object"/>
              <xs:element ref="type-info"/>
            </xs:choice>
            <xs:attribute name="version" default="0.1"/>
            <xs:attribute name="file-type"  />
        </xs:complexType>
    </xs:element>

   <xs:element name="compilation-unit">
       <xs:complexType>
         <xs:sequence>
             <xs:element ref="metadata" />
         </xs:sequence>
           <xs:attribute name="name" />
           <xs:attribute name="id" use="required"/>
```

```
            <xs:attribute name="comp-dir" />
        </xs:complexType>
    </xs:element>

     <xs:element name="object">
         <xs:complexType>
             <xs:choice minOccurs="0" maxOccurs="unbounded">
                 <xs:group ref="all-metadata-types"/>
             </xs:choice>
             <xs:attribute name="name" />
             <xs:attribute name="id" use="required"/>
             <xs:attribute name="comp-dir" />
             <xs:attributeGroup ref="code-region"/>
         </xs:complexType>
     </xs:element>

    <xs:element name="function">
        <xs:complexType>
            <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:group ref="all-metadata-types"/>
                <xs:element ref="param"/>
                <xs:element ref="var"/>
            </xs:choice>
            <!-- By convention we consider the name attribute
                 to be the source-visible name (i.e., demangled), and the
                 symbol to be the linker-visible symbol table entry. -->
            <xs:attribute name="name" use="optional"/>
            <xs:attribute name="id" use="required"/>
            <xs:attribute name="frame-base" use="optional"/>
            <xs:attributeGroup ref="code-region"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="asm-inst">
        <xs:complexType>
            <xs:attribute name="id" />
            <xs:attribute name="dead-regs" />
            <xs:attribute name="address" />
            <xs:attribute name="data-ref" use="optional"/>
        </xs:complexType>
    </xs:element>

     <xs:element name="param">
         <xs:complexType>
             <xs:attribute name="name"/>
             <xs:attribute name="id"/>
             <xs:attribute name="loc"/>
             <xs:attribute name="type-ref"/>
             <xs:attribute name="abstract-origin"/>
             <xs:attribute name="frame-base"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="var">
         <xs:complexType>
             <xs:attribute name="name"/>
             <xs:attribute name="id"/>
             <xs:attribute name="loc"/>
             <xs:attribute name="type-ref"/>
             <xs:attribute name="abstract-origin"/>
             <xs:attribute name="frame-base"/>
         </xs:complexType>
     </xs:element>


     <xs:element name="metadata">
         <xs:complexType>
             <xs:group ref="all-metadata-types"/>
             <xs:attribute name="name" use="optional"/>
         </xs:complexType>
     </xs:element>

     <!-- The location encodes a symbolic or address. Often it is
          inlined into particular metadata element (see the
          "code-point" attributeGroup).
     -->
     <xs:element name="location">
         <xs:complexType>
             <xs:attributeGroup ref="code-point"/>
         </xs:complexType>
     </xs:element>


     <!-- Specific Metadata types -->
     <xs:element name="vft-table">
         <xs:complexType>
         <xs:sequence>
             <xs:element ref="location"/>
             <!-- list of sym/addrs that reference the VFT -->
             <xs:element name="refs" />
         </xs:sequence>
         <xs:attribute name="size" use="required" />
         </xs:complexType>
     </xs:element>

     <xs:element name="switch-table">
         <xs:complexType>
             <xs:sequence>
                 <xs:element ref="location"/>
                 <!-- list of sym/addrs that reference the VFT -->
                 <xs:element name="refs" />
             </xs:sequence>
             <xs:attribute name="size" use="required" />
         </xs:complexType>
     </xs:element>

     <xs:element name="cfg">
         <!-- Control flow. We are forced to convert a graph into a tree,
         so we want to give users reasonable options for storing the CFG-->
         <xs:complexType>
             <xs:sequence minOccurs="1" maxOccurs="unbounded">
                 <xs:element name="basic-block"/>
             </xs:sequence>
             <xs:attribute name="id" use="required"/>
             <xs:attributeGroup ref="code-point"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="basic-block">
         <xs:complexType>
             <xs:attribute name="id" use="required"/>
             <xs:attribute name="target-id" use="optional"/>
             <xs:attribute name="target-address" use="optional"/>
             <xs:attributeGroup ref="code-region"/>

         </xs:complexType>
     </xs:element>

     <xs:element name="rcs-info">
         <xs:complexType>
             <xs:attribute name="id" use="required"/>
             <xs:attribute name="rcs-id" use="optional"/>
             <!-- Tool name: cvs, svn, git, hg -->
             <xs:attribute name="rcs-tool" use="optional" />
             <xs:attribute name="rcs-repository" use="optional"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="dead-regs">
         <xs:complexType>
             <xs:attribute  name="reg-state" use="required"/>
             <xs:attributeGroup ref="code-point"/>
         </xs:complexType>
     </xs:element>

     <!-- Run-time metadata -->
     <xs:element name="run-info">
         <xs:complexType>
             <xs:sequence>
                 <xs:element ref="system"/>
                 <xs:element ref="arguments"/>
                 <xs:element ref="input"/>
                 <xs:element ref="logs"/>
             </xs:sequence>
         </xs:complexType>
     </xs:element>

     <xs:element name="system">
         <xs:complexType>
             <xs:attribute name="name"/>
             <xs:attribute name="os" />
             <xs:attribute name="arch" use="optional"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="arguments">
         <xs:complexType>
             <xs:attribute name="values"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="input">
         <xs:complexType>
             <xs:attribute name="filename" use="optional"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="logs">
         <xs:complexType>
             <xs:attribute name="filename" use="optional"/>
             <!-- stdout, stderr-->
             <xs:attribute name="type" use="optional"/>
         </xs:complexType>
     </xs:element>

     <!-- Profiling, coverage & frequency -->
     <xs:element name="coverage">
         <xs:complexType>
             <xs:attribute name="run-ref"/>
             <xs:attributeGroup ref="code-region"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="frequency">
         <xs:complexType>
             <xs:attribute name="run-ref"/>
             <xs:attributeGroup ref="code-region"/>
         </xs:complexType>
     </xs:element>

     <xs:element name="metadata-ref">
         <xs:complexType>
             <xs:attribute name="id" use="required"/>
             <xs:attribute name="ref-id" use="required"/>
         </xs:complexType>
     </xs:element>

     <!-- Debugging-related metadata -->
     <xs:element name="type-info">
         <xs:complexType>
             <xs:sequence maxOccurs="unbounded">
                 <xs:element ref="type-def"/>
             </xs:sequence>
         </xs:complexType>
     </xs:element>

     <xs:element name="type-def">
         <xs:complexType>
             <xs:attribute name="id"/>
             <xs:attribute name="type"/>
             <xs:attribute name="base-type"/>
             <xs:attribute name="size"/>
             <xs:attribute name="name"/>
             <xs:attribute name="count"/>
         </xs:complexType>
     </xs:element>

     <!-- Attribute helpers -->
     <xs:attributeGroup name="code-point">
         <xs:attribute name="address" use="optional"/>
         <xs:attribute name="symbol" use="optional"/>
         </xs:attributeGroup>


         <xs:attributeGroup name="code-region">
     <xs:attribute name="address" use="optional"/>
     <xs:attribute name="end-address" use="optional"/>
     <xs:attribute name="symbol" use="optional"/>
     <xs:attribute name="end-symbol" use="optional"/>
         </xs:attributeGroup>

     <xs:group name="all-metadata-types">
         <xs:choice>
             <xs:element ref="metadata"/>
             <xs:element ref="function"/>
             <xs:element ref="location"/>
             <xs:element ref="vft-table" />
             <xs:element ref="switch-table" />
             <xs:element ref="rcs-info"/>
             <xs:element ref="cfg"/>
             <xs:element ref="frequency"/>
             <xs:element ref="coverage"/>
             <xs:element ref="run-info"/>
             <xs:element ref="basic-block"/>
             <xs:element ref="type-info"/>
         </xs:choice>
     </xs:group>

 </xs:schema>
```

Appendix B

Glossary

Definitions are collected here for easy reference. In general, the accepted definitions for terms are used, although some terms are used in a more restricted sense than their usual interpretation.

ABI. The application binary interface is the interface for organizing data and making function calls. Stack layout and register usage are specified as well as other details necessary for modules to interact.

AST. The abstract syntax tree is a tree representation of a program, with much of the unnecessary details of the orginal language syntax removed [3].

BODE. The Buffer Overrun Detection Engine is a tool for identifying buffer overruns at run-time. BODE is described in detail in 5.2.

Fragment. A fragment or code fragment, is a small unit of code, translated by an SDT. It may consist of a single or multiple dynamic basic blocks.

IBTC. Indirect Branch Translation Cache. The data hashtable for mapping indirect branch targets to fragment cache targets. Compare *sieve*.

MEDS. The Memory Error Detections System is a object/referent tracking runtime memory analysis tool built by Hiser et al. [59].

Metadata. Data about data. Information about the program that can help improve the program structure or run-time behavior.

Metaman. Metaman is the Metadata Manager designed and implemented to show the effectiveness of collecting and storing program metadata.

Runtime, Run-time In this document the phrase "runtime" has two meanings, to avoid confusing they are spelled differently and defined here: **Runtime** [n]. Executable code that offers basic services for the programming environment (e.g., the C runtime). **Run-time** (1) [n.]. The period of time while the program is executing, distinct from compile-time. (2) [adj.] Having the property of operating at run-time (e.g., a run-time system).

Schema. An XML-based validation document for XML documents. An XML schema determines what elements and tags can or must appear in a validated XML document.

Sieve. A software-based hashtable for quickly looking up the translated target address of already-translated indirect branch instructions.

Software Dynamic Translation (SDT). Software Dynamic Translation is a system-building tool which examines and possibly alters each instruction before it is executed. Examples of such systems are Strata [109], Pin, and DynamoRIO.

Strata. Strata is a low-overhead, research oriented software dynamic translation system developed at the University of Virginia and University of Pittsburgh.

Transparency. In reference to SDT systems, transparency is the property that the SDT does not alter the defined behavior of the program. A transparent SDT system will allow the program to execute and have the same observable side-effects as if there were no SDT system running.

x86. Intel 32-bit Architecture (IA-32). X86 assembly syntax comes in two styles, Intel and AT&T. All examples in this dissertation are Intel syntax, except those explicitly related to the GNU assembler, which uses AT&T syntax.

XML. Extensible Markup Language is a data description language based on nested tags.

Bibliography

[1] Martìn Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[2] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. A formal approach to code optimization. In *Proceedings of a Symposium on Compiler optimization*, pages 86–100, New York, NY, USA, 1970. ACM.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] American Telephone and Telegraph Company. *System V application binary interface: SPARC processor supplement*. 1990.

[5] A. W. Appel. Foundational proof-carrying code. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 247–256. IEEE, 2001.

[6] Andrew Appel and Jack W. Davidson. The Zephyr Compiler Infrastructure.

[7] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[8] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th annual ACM SIGPLAN con-*

ference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pages 297–311, New York, NY, USA, 2005. ACM.

[9] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[11] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.*, 32, August 2010.

[12] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[13] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 21, Berkeley, CA, USA, 2000. USENIX Association.

[14] Richard Baskerville and Jan Pries-Heje. Short cycle time systems development. *Information Systems Journal*, 14(3):237–264, 2004.

[15] Goran Begic and Allan Pratt. An introduction to runtime analysis with rational PurifyPlus. `http://www.ibm.com/developerworks/rational/library/mar07/begic_pratt/index.html`, March 2007.

[16] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, New York, NY, USA, 2006. ACM.

[17] Brian Berliner. CVS II: Parallelizing software development, 1990.

[18] Robert L. Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985.

[19] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, August 2004.

[20] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[21] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, November 2000.

[22] Raymond P. L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.

[23] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[24] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. TEAM: a support environment for testing, evaluation, and analysis. *SIGSOFT Softw. Eng. Notes*, 13:153–162, November 1988.

[25] Computer Economics, Inc. 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. `http://www.computereconomics.com/article.cfm?id=1224`, 2007.

[26] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, volume 38, pages 232–244, New York, NY, USA, May 2003. ACM.

[27] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[28] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, page 5, Berkeley, CA, USA, 1998. USENIX Association.

[29] Anthony Cox, Charles Clarke, and Susan Sim. A model independent source code repository. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[30] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[31] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 2(2):191–202, April 1980.

[32] B. de Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Cooperative and Human Aspects on Software Engineering, 2009. CHASE '09. ICSE Workshop on*, pages 36–39. IEEE, May 2009.

[33] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of ARM binaries. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 211–220, New York, NY, USA, 2004. ACM.

[34] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. In *ASPLOS-IX: Proceedings of the ninth International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 202–211, New York, NY, USA, 2000. ACM Press.

[35] DWARF Debugging Information Format Workgroup. DWARF debugging information format version 3. `http://dwarf.freestandards.org`, 2005.

[36] Jürgen Ebert and Angelika Franzke. A declarative approach to graph based modeling. In Ernst Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 38–50. Springer Berlin / Heidelberg, 1995.

[37] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.

[38] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments. part i: tool specification. *ACM Trans. Softw. Eng. Methodol.*, 1:135–167, April 1992.

[39] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[40] Thomas G. Evans and D. Lucille Darley. On-line debugging techniques: a survey. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, AFIPS '66 (Fall), pages 37–50, New York, NY, USA, 1966. ACM.

[41] Stuart I. Feldman. Make a program for maintaining computer programs. *Softw: Pract. Exper.*, 9(4):255–265, 1979.

[42] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, volume 19 of *SIGPLAN '84*, pages 37–47, New York, NY, USA, June 1984. ACM.

[43] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, pages 947–954, New York, NY, USA, 2009. ACM.

[44] M. P. Gallaher and B. M. Kropp. Economic impacts of inadequate infrastructure for software testing. May 2002.

[45] Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson. A secure and robust approach to software tamper resistance information hiding. volume 6387 of *Lecture Notes in Computer Science*, chapter 3, pages 33–47. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010.

[46] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[47] Mary J. Granger and Roger A. Pick. The impact of computer-aided software engineering on student performance. *SIGCSE Bull.*, 23:62–72, March 1991.

[48] Apala Guha, Kim Hazelwood, and Mary Soffa. Balancing memory and performance through selective flushing of software code caches. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 1–10, New York, NY, USA, 2010. ACM.

[49] L. H. Haines. Serial compilation and the 1401 FORTRAN compiler. *IBM Systems Journal*, 4(1):73–80, 1965.

[50] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 156–164, New York, NY, USA, 2004. ACM.

[51] Kim Hazelwood and James E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:89+, 2004.

[52] Kim Hazelwood and Michael D. Smith. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society.

[53] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[54] John L. Hennessy and Noah Mendelsohn. Compilation of the pascal case statement. *Software: Practice and Experience*, 12(9):879–882, 1982.

[55] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[56] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, volume 37 of *POPL '02*, pages 58–70, New York, NY, USA, January 2002. ACM.

[57] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 11 pp.–154, December 2005.

[58] Jason Hiser, Daniel Williams, Adrian Filipi, Bruce Childers, and Jack Davidson. Evaluating fragment construction policies for SDT systems. In *VEE'06: Second International Conference on Virtual Execution Environments*, pages 122–131, New York, NY, USA, 2006. ACM Press.

[59] Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. MEDS: The memory error detection system. In *ESSoS '09: Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, pages 164–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[60] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[61] Jason D. Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Trans. Archit. Code Optim.*, 8(2), June 2011.

[62] R. C. Holt, A. Winter, and A. Schurr. GXL: toward a standard exchange format. pages 162–171.

[63] Grace M. Hopper. The education of a computer. In *ACM '52: Proceedings of the 1952 ACM national meeting (Pittsburgh)*, pages 243–249, New York, NY, USA, 1952. ACM.

[64] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *Computer Journal*, pages 223–229, August 1980.

[65] Wei Hu, Jason Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh N. Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE'06: Second International Conference on Virtual Execution Environments*, pages 2–12, New York, NY, USA, 2006. ACM Press.

[66] Ibm. Use -qdbgfmt=dwarf to enable DWARF debugging format on AIX v7.1. *IBM Support*, October 2010.

[67] Brian Johnson, Marc Young, and Craig Skibo. *Inside Microsoft Visual Studio .NET*. Microsoft Press, Redmond, WA, USA, 2002.

[68] G. Jung, D. G. Meyer, and V. Milutinovic. Flexible register window structure for multitasking. pages 110–116.

[69] Michael Karasick. The architecture of montana: an open and extensible programming environment with an incremental c++ compiler. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 131–142, New York, NY, USA, 1998. ACM Press.

[70] Ho S. Kim and James E. Smith. Hardware support for control transfers in code caches. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 253+, Washington, DC, USA, 2003. IEEE Computer Society.

[71] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[72] Raimund Kirner and Peter Puschner. Classification of code annotations and discussion of Compiler-Support for Worst-Case execution time analysis. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2007.

[73] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22:490–505, May 2000.

[74] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, 2003.

[75] Steven Knight. SCons design and implementation. In *Tenth International Python Conference*, 2002.

[76] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, January 1984.

[77] Naveen Kumar, Bruce R. Childers, and Mary L. Soffa. Low overhead program monitoring and profiling. In *PASTE '05: Program Analysis for Software Tools and Engineering*, 2005.

[78] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary L. Soffa. Instrumentation in software dynamic translators for self-managed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 90–94, New York, NY, USA, 2004. ACM Press.

[79] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not.*, 23:318–328, June 1988.

[80] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, volume 0, pages 75–86, Los Alamitos, CA, USA, March 2004. IEEE.

[81] Chris Lattner and Vikram Adve. *LLVM Language Reference Manual*, February 2011.

[82] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, page 6, Berkeley, CA, USA, 1998. USENIX Association.

[83] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1811–1841, 1994.

[84] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[85] Chi K. Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference*

on Programming language design and implementation, volume 40 of *PLDI '05*, pages 190–200, New York, NY, USA, June 2005. ACM.

[86] N. Mabanza, J. Chadwick, and G. S. V. R. Krishna Rao. Performance evaluation of open source native XML databases - a case study. *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, 3:1861–1865, May 2006.

[87] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 CWE/SANS top 25 most dangerous programming errors. http://cwe.mitre.org/top25/, 2010.

[88] Johannes Martin. Leveraging IBM visual age for c++ for reverse engineering tasks. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[89] Ken Martin and Bill Hoffman. *Mastering CMake 4th Edition*. Kitware, Inc., USA, 4th edition, 2008.

[90] Michael O. McCracken. The design and implementation of the LENS program information framework. Technical report, UCSD CSE, 2006.

[91] Collin McCurdy and Charles Fischer. Using pin as a memory reference generator for multiprocessor simulation. *SIGARCH Comput. Archit. News*, 33:39–44, December 2005.

[92] Scott McFarling. Reality-based optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 59–68, Washington, DC, USA, 2003. IEEE Computer Society.

[93] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, pages 136–147, Washington, DC, USA, 1999. IEEE Computer Society.

[94] MSDN. Querying the .pdb file. `http://msdn.microsoft.com/en-us/library/eee38t3h.aspx#Y305`, 2011.

[95] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[96] George C. Necula. Proof-carrying code. pages 106–119, 1997.

[97] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-Checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, 2004.

[98] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[99] R. A. Olsson and G. R. Whitehead. A simple technique for automatic recompilation in modular programming languages. *Softw. Pract. Exper.*, 19:757–773, August 1989.

[100] Oracle. Why can't dbx find my function? `http://developers.sun.com/solaris/articles/dbxerr.html`, 2010.

[101] Bruce Perens. Electric fence. `http://perens.com/FreeSoftware/ElectricFence/`, June 1991.

[102] Matthai Philipose, Craig Chambers, and Susan J. Eggers. Towards automatic construction of staged compilers. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 37 of *POPL '02*, pages 113–125, New York, NY, USA, January 2002. ACM.

[103] César A. Quiroz. Using c++ efficiently in embedded applications.

[104] Hridesh Rajan. Generalizing AOP for Aspect-Oriented testing. In *In the proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005*, pages 14–18. ACM Press, 2005.

[105] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS*. The Internet Society, 2004.

[106] Sven Schreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison-Wesley Professional, May 2001.

[107] A. Schürr, A. J. Winter, and A. Zündorf. *The PROGRES approach: language and environment*, pages 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[108] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 200+. IEEE, April 2004.

[109] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[110] Kevin Scott, Naveen Kumar, Bruce R. Childers, Jack W. Davidson, and Mary L. Soffa. Overhead reduction techniques for software dynamic translation. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, 2004.

[111] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[112] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, page 2, Berkeley, CA, USA, 2005. USENIX Association.

[113] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[114] Daniel P. Siewiorek, G. Bell, and A. C. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, Inc., New York, NY, USA, 1982.

[115] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[116] B. C. Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[117] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, volume 35 of *DYNAMO '00*, pages 1–11, New York, NY, USA, July 2000. ACM.

[118] Danny Soroker, Michael Karasick, John Barton, and David Streeter. Extension mechanisms in montana. In *ICCSSE '97: Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, page 119, Washington, DC, USA, 1997. IEEE Computer Society.

[119] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 175–185, New York, NY, USA, 2006. ACM.

[120] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. ACM.

[121] Mark Summerfield. *Programming in Python 3: A Complete Introduction to the Python Language*. Addison-Wesley Professional, 1st edition, 2008.

[122] The Eclipse Team. The eclipse project. `http://www.eclipse.org`, accessed 2011.

[123] Walter F. Tichy. Rcs a system for version control. *Softw: Pract. Exper.*, 15(7):637–654, 1985.

[124] W3C. W3C XQuery 1.0 and XSLT 2.0 become standards: Tools to query, transform, and access XML and relational data. `http://www.w3.org/2007/01/qt-pressrelease`, January 2007.

[125] Steven Wallace and Kim Hazelwood. SuperPin: Parallelizing dynamic instrumentation for Real-Time performance. pages 209–220, April 2007.

[126] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, volume 0, pages 364–374, Washington, DC, USA, June 2009. IEEE Computer Society.

[127] Daniel Williams and Jack W. Davidson. Metaman: System-wide metadata management. In Robert Cohn, editor, *The Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009.

[128] Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh N. Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy*, 7(1):26–33, 2009.

[129] Daniel Williams, Aprotim Sanyal, Dan Upton, Jason Mars, Sudeep Ghosh, and Kim Hazelwood. A cross-layer approach to heterogeneity and reliability. *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 88–97, August 2009.

[130] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, and Bo Huang. Metadata driven memory optimizations in dynamic binary translator. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 148–157, New York, NY, USA, 2007. ACM.

[131] Jing Yang, Shukang Zhou, and Mary L. Soffa. Dimension: an instrumentation tool for virtual execution environments. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 164–174, New York, NY, USA, 2006. ACM.

[132] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, May 2009.

[133] Alexander Yip, Benjie Chen, and Robert Morris. Pastwatch: a distributed version control system. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 28, Berkeley, CA, USA, 2006. USENIX Association.