A Survey of Automatic Vulnerability Detection and Repair Systems in Software

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements of the Degree
Bachelor of Science, School of Engineering

By

Kevin Melloy
Fall 2020

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature Kevin Melloy Date 12/1/20

Kevin Melloy

Approved Nathan Brunelle Date 12/13/20

Nathan Brunelle, Department of Computer Science

Introduction

In cybersecurity, there exist many different types and varieties of vulnerabilities within software that can be exploited, allowing for a multitude of malicious actions. These vulnerabilities can have a wide range of consequences, ranging from mild inconveniences to major security threats. Some of the most significant of these consequences can often lead to massive financial loss or theft of personal information. Estimates claim that as many as 4.1 billion records were exposed in just under 4,000 cyber attacks in the first half of 2019 (RiskBased). With a global average cost of $3.92 million per data breach as of 2019 (Security Intelligence), the threat that software vulnerabilities pose is incredibly apparent.

As software is becoming more prevalent in society and the cost of cyber attacks is only increasing--the global average cost per data breach reported in 2019 had grown by 12% since 2014 (Security Intelligence)--it is now more important than ever to develop methods of detection, prevention, and repair of software vulnerabilities. To help combat the threat of software vulnerabilities, DARPA held a competition from 2014 to 2016 titled the Cyber Grand Challenge. The intention of the challenge was to develop automatic software defense systems that were capable of detecting and repairing vulnerabilities within software (darpa.mil). DARPA awarded the winner of the challenge $2 million, a testament to the significance of the topic of software vulnerabilities.

In this paper, we showcase several different papers proposing software vulnerability detection systems, some of which made an appearance at DARPA's Cyber Grand Challenge, as well as some general topics within cybersecurity and software vulnerability detection. For each of the systems discussed, we highlight its primary goal or goals, the approach used to reach those goals, as well as its weaknesses and strengths, with the ultimate intention of illustrating the importance and effectiveness of automatic vulnerability detection.

As the systems and topics discussed within this paper each deal with specific vulnerabilities or have varying approaches to vulnerability detection, it is helpful to provide brief descriptions of some of the software vulnerabilities that will be discussed.

- Buffer Overflows and Injected Code
    - These attacks deal with inputting more data than is expected by a given program. In most cases, a buffer--a data structure used to accept a user's input--has a specific number of bytes it is expecting to receive. If a user enters more data than is expected, that extra data can overwrite certain values that are located past the buffer in memory. This allows malicious users to inject their own code to be executed, change return addresses from functions, etc.
- Invalid Memory Writes
    - Typically systems allocate a specific amount of memory for programs to use while running, and any attempts to access memory outside of this allocation are denied. This is due to the fact that operating systems will typically have important data stored at some of these inaccessible memory locations. If a user were able to access any memory location from a given program, he or she would be able to alter the functionality of the operating system.
- Zero-Day Attacks
    - A zero-day vulnerability is one that is unknown by software developers until the software's release. This can be any kind of vulnerability, and is especially dangerous due to it being exploitable without the knowledge of the developers.
- Function Boundary Errors
    - Function boundary errors deal with the beginning and ending memory locations of functions, as well as memory located in between functions. For example, if a wide enough gap of memory exists between functions, it would be possible to hide malicious code within that gap.

- Code Disguised as Data
    - Within most executables, data is stored in specific sections of the program, typically at the beginning of the file. As data and assembly instructions are both represented as bytes, it is possible to conceal malicious instructions within the same sections as data, hiding them from disassemblers.

To help provide context for the majority of the systems in this paper, the first two papers that are discussed provide an example of automatic detection and support for trusting of automated systems, respectively.

Disguising Malicious Code as Data

As mentioned above, disguising code as data is one of the many ways that malicious code can find its way into software. This is an area in which the advantages of automation can be seen quite clearly. The technique involves hiding x86 assembler instructions within areas of a file not revealed by a disassembler. Thus, manual detection of such code would require access to the entirety of a file's contents--disassembling the program would not be sufficient--followed by analysis of each byte to ensure that no malicious instructions were present. This would be a tedious process to say the least. Each individual byte would have to be compared to a list of bytes that indicate an assembly instruction. If any byte were to indicate an instruction, the string of bytes would then have to be compared to the length of the instruction. Finally, if the length of that string of bytes matched the length of the instruction in question, each byte within that string would have to be analyzed to determine if the instruction did indeed make sense. To avoid this process needing to be a manual one, Wartell et al. present their algorithm for the automatic detection of code within data in their paper, *Differentiating Code From Data in x86 Binaries*.

Summary

The algorithm that the authors' system uses combines heuristics used in expert reverse engineering and a language model that captures correlations between byte sequences. To start, the executable on which the system is being run is treated as a string and divided into segments. Each segment is then tagged as either more likely to be code or data, using the authors' algorithm to determine which is more probable. A difficulty with this task is ambiguities. A given string of bytes could have multiple different interpretations and requires an understanding of its surrounding context to determine which meaning is correct. The algorithm must also deal properly with padding bytes.

The tagging algorithm is comprised of two components: an instruction reference array and a utility function. The reference array holds information on the length of x86 instructions given an initial byte. The utility function is what determines the likelihood of a byte sequence being code, using a model built from pre-tagged x86 binaries to estimate the probability.

After the tagging algorithm estimates each string of bytes to be either code or data, each segment must be reclassified using the aforementioned language model and heuristics adapted from expert, manual disassembly. This step is necessary as the tagging algorithm is imperfect and may mark segments incorrectly.

Wartell et al. tested this process on 11 programs. For each program, the other 10 were used to build the language model. The authors summed all of the false positives (data tagged as code) and false negatives (code tagged as data) together, converted this sum to a percentage and subtracted it out to give the percentage of correct tags. Using this formula, the authors showed an accuracy of 100% of 6 out of the 11 executables, with near-perfect accuracy on the rest. The classification accuracy was also shown to be near-perfect, with 100% accuracy on 5 out of 11 executables.

Limitations

As mentioned before, some segments of bytes are ambiguous. The authors provide an example of this: a data segment starting with an initial byte that would indicate an assembly

instruction and being of a length that matches that instruction's length in the reference array. This would be incorrectly tagged as code. To attempt to get around this, the heuristics drawn from manual disassembly are used. However, manual disassembly is obviously imperfect, thus this is an incomplete workaround.

The algorithm is also unclear on how to tag padding bytes. Such bytes are legitimate code instructions, but could also be classified as data. The authors claim that classification as code would be preferable.

Additionally, while the authors' algorithm does seem to be effective when looking at the results of the authors' experiments, 11 executables seems too small a number to adequately prove the usefulness of this system.

Trust in Automated Software Repair

As a large goal of this paper is showcasing the automation aspect of software vulnerability detection systems, it is important to discuss the trustworthiness of automated software analysis. It is not difficult to see the necessity of automation within the future of cybersecurity, however it is also necessary to ensure that it is accurate and trustworthy. In other words, automated software vulnerability detection systems will not be useful if they are not trusted, even if they are accurate. This is precisely the topic covered by Ryan et al. in the following paper, *Trust in Automated Software Repair*.

Summary

To study how trustworthiness of automated and manual software repairs compare, Ryan et al. conducted two studies. The first study was performed with software developer students, while the second was performed with software developer experts. In each study, the participants were shown five different pieces of code, both before and after being repaired by an automated repair program. The participants were then asked to rate the trustworthiness of the repair and whether or not they would endorse the use of the repair. In both studies, half of the participants

were informed that the source of the repairs was an automated repair program, while the other half was made to believe that the repairs were written by a human.

This difference between the believed source of the repairs had a significant effect on the results of the studies. In both cases, the participants showed that trust in the “human” written repairs declined severely between the initial and final assessments while trust in the automated repair program did not significantly change. Ryan et al. speculate that this may be due to the unorthodox methods used by the automated repair program. Repairs made using unconventional approaches by a human would inhibit trustworthiness from another human, while those same approaches may seem perfectly normal when coming from a machine. Thus, those in the study that believed that the unorthodox methods came from manual repair stated that they trusted the repair less. It is important to note that each of the repairs shown were equally as effective as each of the rest.

The results from the two studies led Ryan et al. to the following conclusion: trust in automated processes is determined through different factors than in human-to-human trust. Specifically, trust in automated programs is primarily determined by the performance of the program, while trust in humans incorporates other non-performance-based factors, such as methods used and perceptions of efficiency. The authors cite previous research to back this conclusion.

Limitations/Future Work

As with any study, the studies performed could have benefitted from a larger, or perhaps simply more diverse, sample size. Additionally, some of the tools used to receive input from the participants were very black-and-white, usually of the form of a question such as “Would you endorse this or not?”, when perhaps a more expressive response would have allowed for more information.

As mentioned earlier, Ryan et al. stress how important this research of trust in automated repairs is. The authors hope that this paper will be an impetus for future work in the study of human biases with regards to automation in software development.

Automated Detection and Repair Systems

In this next section, we discuss some of the systems that deal with automated detection and repair of software vulnerabilities. The methods used for detection vary between several different techniques, with static analysis, dynamic analysis, and symbolic execution being among them. The repair aspect of these systems also differs, with some systems simply invalidating inputs that would trigger program faults while other systems generate software patches.

Automatically Patching Errors in Deployed Software

The first of the systems in this section is named ClearView. It is a system designed to automatically detect and patch errors in Windows x86 binaries. The ClearView system can be used to defend against attacks or provide immunity against security risks. It achieves this provision of defense and immunity by identifying aspects, called invariants, of how a given executable would run normally, detecting when an error occurs, identifying violations of established invariants, and generating potential repair patches before selecting the most successful patch. ClearView is able to do this patching of errors without requiring the application to restart. Perkins et al. describe this system in their paper, *Automatically Patching Errors in Deployed Software*.

Summary

In order to properly identify an application's invariants, ClearView observes the application when running normally, creating a collection of important parts of the execution such as memory locations and register values. Multiple runs of the application can be executed in

order to improve ClearView's collection of these invariants. ClearView labels each execution of an application as a success or failure depending on its failure detection. This failure detection currently (as of the writing of the paper) uses two monitors: Heap Guard and Determina Memory Firewall. HeapGuard is used to detect out-of-bounds memory writes and Determina Memory Firewall detects illegal transfers of control flow. When a failure is detected, the monitor provides the location within the binary of the failure.

To fix a violated invariant, ClearView generates many candidate repair patches, restoring register values and memory locations or changing the control flow based on previously observed invariants. Each of these candidate patches must be evaluated, as each individual patch may have no effect or even, in some cases, a negative effect. To perform this evaluation, ClearView observes each patch applied and ranks them based on whether or not the application crashed with the patch in place. ClearView then selects the patch that minimizes the probability of a crash.

To test ClearView's effectiveness in providing security against attacks, DARPA hired Sparta Inc. to perform a Red Team evaluation of the system. This evaluation took place on a set of computers all running the Firefox web browser. Within the evaluation, ClearView was not only able to protect against all attacks by the Red Team, but was also able to provide immunity against attacks to machines that had not yet been attacked. ClearView's success was achieved by finding a valid defense against an attack on one computer, and sharing it to all other computers in the community of the evaluation. In addition to blocking all attacks, preventing the execution of injected code, ClearView was also able to provide successful patches to seven of ten attacks, allowing the application to continue execution without interruption. No false positives were encountered during the evaluation.

Limitations

While extremely effective at detecting, blocking, and patching errors as seen from the Red Team evaluation, ClearView is only able to be successful against errors detectable by its

monitors. For errors beyond the scope of HeapGuard, ShadowStack, and Determina Memory Firewall, additional monitors would need to be incorporated.

There is always the chance that a patch generated and selected by ClearView may negatively affect the application. False positives may also happen, for example a valid input could theoretically cause ClearView to generate an unnecessary patch. These limitations could be solved, or at least minimized, with a better learning model. It is also possible that ClearView is unable to create a patch that protects against an error to a satisfiable degree.

Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing

The next of the systems within this section is called ShieldGen. It is a system designed to automatically generate a data patch or vulnerability signature for an unknown vulnerability given an attack instance. This automatic generation of patches would allow for far more efficient defense against zero-day attacks, an attack that exploits a vulnerability that is unknown to the developer or vendor of a given piece of software, as prior to the ShieldGen system, the majority of new vulnerability analysis and defense generation has been mostly manual. Cui et al. describe this system in their paper, *ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing.*

Summary

The ShieldGen system works by using knowledge of the input data format to generate multiple new potential attack instances, which the authors call *probes*. Each probe is sent to a zero-day detector to determine if it is still capable of exploiting the vulnerability in question. The results from the zero-day detector are used to construct more probes as well as to discard attack-specific parts of the original attack while retaining the parts that relate to the vulnerability itself, ultimately leading to the generation of the vulnerability signature. The generated vulnerability signature would act as a filter, processing input to the program and removing data that would attack the vulnerability. Using a ShieldGen prototype to experiment with three known

vulnerabilities, Cui et al. were able to generate vulnerability signatures with no false positives and a low rate of false negatives.

ShieldGen consists of two main components: an oracle (zero-day detector) and a data analyzer. ShieldGen's oracle is based on dynamic data flow analysis – it monitors and tracks how the input data propagates and changes as the program executes. The oracle has three vulnerability conditions: arbitrary execution control, arbitrary code execution, and arbitrary function arguments. Arbitrary execution control checks whether input data is about to be moved into the instruction pointer, this monitor attempts to overwrite return addresses and stack or function pointers. Arbitrary code execution tests whether a given instruction depends on the input data. This monitors for possible attempts at execution of injected code. The arbitrary function arguments condition is similar to arbitrary code execution, but is specifically related to critical system calls. The oracle will check whether said calls depend on input data. If any of these three conditions is violated, the oracle will issue an alert, providing detailed information on the exploit and vulnerability, including data flow history and application state at the moment the vulnerability was detected as well as the locations of the values that triggered the alert.

The data patch generation of ShieldGen starts with the derivation of the vulnerability predicate. The data analyzer checks for any violations of the data format constraints. For any violated constraints, probes are constructed that satisfy the constraints. If these probes were unsuccessful in exploiting the vulnerability, the data patch is simply the data format specification that enforces the data format constraint. If a probe is successful, the process moves on to the generation of the attack predicate. This predicate is composed of boolean conditions with each data field being equal to the values in the attack input. From here, the data patch generation algorithm will relax or remove conditions specific to the original attack input, allowing for the admittance of more attack variants. If all values of a data field have been tried and been classified as attacks, the oracle marks that field as a "don't-care" field and is removed from the predicate. At the end of this process, the predicate with the minimal necessary fields is left.

Cui et al. ran a ShieldGen prototype on three known vulnerabilities to test the efficiency and accuracy of ShieldGen. A pencil-and-paper evaluation was also performed to estimate ShieldGen's accuracy. Each probe was found to take roughly 10 seconds, however the authors believe that this can be reduced significantly. Two of the known vulnerabilities were of the stack buffer overrun variety, while the last one was not. ShieldGen was able to identify the two buffer overrun vulnerabilities and address them correctly. For the third vulnerability, ShieldGen was able to provide a vulnerability signature after some initial unsuccessful attempts. In the pencil-and-paper evaluation, it was found that ShieldGen was able to produce precise filters for 19 of the 25 vulnerabilities in question.

Limitations

While ShieldGen is quite effective at accomplishing the authors' goal of automatic data patch generation for the cases showcased in this paper, it is limited in what types of vulnerabilities it can address. ShieldGen is very dependent on data format specification, meaning that a vulnerability with a complex data format would likely be able to stump it. SheildGen's vulnerability detector is used as a black box, only capable of giving yes or no outputs when perhaps something less black and white could be useful. As mentioned earlier, as of the writing of the paper, each probe had a total time of approximately 10 seconds. Future work on this topic could aim to improve this time, as Cui et al. mention.

FuzzBomb: Fully-Autonomous Detection and Repair of Cyber Vulnerabilities

The next automated detection system is detailed in *FuzzBomb: Fully-Autonomous Detection and Repair of Cyber Vulnerabilities* by Musliner et al. FuzzBomb is an autonomous cyber vulnerability detection and repair system built upon the authors previous work, Fuzzbuster and FuzzBALL. It was created as part of DARPA's Cyber Grand Challenge (CGC) and performed rather well in the challenge's practice scoring events before placing just outside the qualifying range in the CGC Qualifying Event.

Summary

As mentioned, FuzzBomb is built upon Fuzzbuster, a system used to find flaws within software through symbolic analysis and fuzz testing. Fuzzbuster works by creating what Musliner et al. call “reactive exemplars” and “proactive exemplars.” These exemplars are created when an attacker triggers a program fault (reactive) or when Fuzzbuster’s fuzz testing determines a potential exploit (proactive). Using the exemplars, Fuzzbuster builds vulnerability profiles through its analysis tools and further fuzz-testing. From there, the system is able to develop constraints, determining things such as the minimal portion of an input string that triggers a given exploit. Fuzzbuster can then create a filter blocking that specific input, protecting the software from that particular path to the vulnerability. Fuzzbuster is able to continuously refine its understanding of the flaw, leading to better constraints and filters over time, eventually preventing exploitation of all the known vulnerabilities while maintaining full program functionality.

FuzzBALL, the other system that FuzzBomb is built upon, is a system used for symbolic execution with a specific focus on binary software. This particular emphasis on binary execution sets FuzzBALL apart from many other symbolic execution tools. FuzzBALL works by exploring various executions of a given binary, building what Musliner et al. call a “decision tree.” The nodes of this tree represent occurrences of symbolic branches for a given execution. The tree is used to ensure that each execution explored is unique. This system’s program analysis capabilities, along with FuzzBuster’s logic framework, were integrated into FuzzBomb for DARPA’s CGC.

FuzzBomb improves upon FuzzBALL’s symbolic execution engine. Static analysis identifies areas of the software that could potentially contain a vulnerability. The system then performs a symbolic execution to find an execution path to the potential vulnerability. This symbolic execution generates a number of input constraints that lead to the vulnerability, determining the inputs that the system should block. It is noted that this procedure of generating

input constraints through various execution paths requires a large amount of space and computing power. To mitigate this, Musliner et al. applied parallelization techniques and heuristic search improvements. For example, if reachability analysis determines a vulnerability to be unreachable, the vulnerability is ignored.

The authors acknowledge that the FuzzBomb system does not eliminate the underlying problem of the software that led to the vulnerabilities. Rather, the system applies a remedy, mitigating access to the vulnerabilities, often simply terminating the program when a vulnerable execution path is attempted.

Limitations/Future Work

While FuzzBomb performed well in the CGC's practice events, the authors acknowledge that the system ran into some unanticipated challenges during the qualifying event. In particular, there was an analysis that determined every element in the software to be a constant, giving the fuzzing tools nothing to work with. Additionally, FuzzBomb was only able to solve 7 of 24 of the challenge's problems on its own. It was, however, able to fully solve each of them when given the proof of vulnerabilities for the problems, perhaps leading to the conclusion that the system's vulnerability detection could stand to be improved. Musliner et al. end by stating that they are actively looking into real world applications of the FuzzBomb system.

Identifying Open-Source Functions in Malware Binaries

Alrabaee et al. present FOSSIL, a three-component system they have designed to efficiently identify free open-source software (FOSS) packages within binaries when the source code is unavailable. This system is detailed in their paper, *FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries*.

Summary

The first component of FOSSIL extracts syntactical features of functions. The second component extracts the semantics of functions. The third and final component of the system

applies a z-score to the normalized instructions to extract the behavior of instructions within functions. Each of these components is incorporated within a Bayesian network model which uses the results of the components to determine the FOSS function. Alrabaee et al. show that their system is able to identify FOSS packages with a mean precision of 0.95 and a mean recall of 0.85, pointing out that this is significant due to the frequency with which FOSS packages are used within malware. By identifying which FOSS functions are present within a given binary, insight into the binary's functionality can be gained.

The system's process of identifying FOSS functions is divided into four steps. The first step normalizes the binary's instructions, helping to account for variations in the code based on different compilers and compiler settings. Then, the instructions are sent to the system's previously mentioned three components for feature extraction. The third step is feature processing, making use of a hidden Markov model (HMM) to detect the behavior of a function given opcode frequencies. During this same step, control-flow graphs may be used. The final step of the process uses a Bayesian Network to identify the FOSS function. Alrabaee et al. use a collection of known FOSS packages compiled with different compilers and compilation settings to evaluate the efficiency and accuracy of their system's process of identifying FOSS functions.

The normalization step of the system's process is accomplished by categorizing x86 assembly instructions into three categories: memory references, registers, and constants. The control-flow graphs (CFGs) consist of blocks, or nodes, which represent sets of instructions. By using these graphs to develop "walks" the authors are able to represent the semantic relations of functions. Thus, by converting a potential FOSS function into one of these CFGs, the authors can reference the already existing CFGs to find a match, or approximate match. Furthermore, the system will also take opcode frequency distributions into account. Alrabaee et al. mention that this is due to the hypothesis that FOSS functions performing the same task will often have similar opcode distributions. These opcode distributions then have their opcodes ranked

according to their importance within the function. The HMM mentioned above makes use of these rankings to create confidence intervals. Additionally, z-scores are utilized to convert opcode distributions into scores. Finally, the Bayesian Network (BN) makes use of all the aforementioned information to model the interaction between each of the components, providing a probability function.

To test their system and evaluate its efficiency and accuracy, Alrabaee et al. used 160 projects that reuse FOSS packages. All experiments were run on machines running Windows 7 and Ubuntu 15.04. Within their evaluations, the authors were primarily concerned with finding as many relevant functions as possible with little concern for false positives. The results of the evaluations showed several points:

- Compiler functions should be filtered out prior to running FOSSIL to allow for better precision.
- Accuracy of the system is dependent on the project. For example, running on cryptography related libraries will often yield higher accuracy than parser related libraries.
- FOSSIL scales very well, being minimally affected by the size of the project.
- Applying different values to the Bayesian network model can allow for trade offs between precision and recall.

When comparing their system to other modern systems of similar nature, the authors show that FOSSIL is generally superior in terms of precision (average of 95%), recall (average of 89%), and efficiency (average of 48.5s). Additionally, the authors show that their system is scalable when running on up to at least 1.5 million functions, with only a slight decrease in accuracy.

Limitations/Future Work

The FOSSIL system has many limitations. Alrabaee et al. list several:

- Function Inlining:

    - The authors state that they do not currently support the ability to fingerprint a function with partial code in another program, thus this compiler optimization would cause issues.
- Multiple Architecture:
    - Currently, the system only deals with x86. However, the authors state that future work will involve researching how to deal with multiple architectures.
- Type Inference:
    - FOSSIL does not currently support type inference, which the authors state would help reduce the number of false positives.
- Advanced Obfuscation:
    - One of the system's greatest limitations is being unable to deal with advanced obfuscation techniques. One of the assumptions made by the system is that the code is already unpacked. Thus, obfuscation would render the system nearly useless.
- Dataset Size:
    - While the repository of FOSS functions used in this paper is quite large, the authors state that its size would still need to be dramatically increased, something they state will be addressed in future work.

Function Boundary Detection in Stripped Binaries

In this next paper, *Function Boundary Detection in Stripped Binaries,* Alves-Foss and Song present an automated function detection algorithm as a part of the Jima tool suite. The algorithm takes in stripped binaries and returns a list of possible function boundary locations. This is useful, as stripped binaries will typically not have symbol tables containing function addresses. Alves-Foss and Song developed this tool for DARPA's Cyber Grand Challenge and evaluated it and related tools against the SPEC CPU 2017 test suite and the Chrome browser.

Summary

The problem that the algorithm addresses deals with determining the list of functions within a binary without the symbol table or debug information. In particular, the algorithm has the goal of finding function starts and function boundaries. With this goal in mind, Alves-Foss and Song assume that all instructions are contiguous in memory. Other systems and algorithms that also deal with discovering function boundaries exist, such as Nucleus and Ghidra, and the authors reference them as the primary systems to which they compare Jima.

The Jima function detection algorithm consists of six phases. The first of these phases is disassembly. Jima uses the objdump command to list out all of the assembly instructions. These instructions are then parsed and stored in a data structure which the authors call Jil. During this parsing, Jima records all control flow operations along with their respective target addresses, if possible. The second phase is exception handler analysis. Here, Jima maps exception handling code to the parent function. Phase three deals with jump pointer analysis—Jima iterates through each jump pointer operation and attempts to detect the location and size of the related jump tables. In phase four, Jima iterates through all call destinations. For each call destination, Jima moves forward in the code until the function terminates or another call destination is reached. Phase five is the detection of missing functions. These missing functions are defined as executable code within the gaps of the functions detected from phase four. The final phase detects all terminal function calls. The authors define a terminal function as any function that does not return from execution.

During the disassembly phase, Jima categorizes each observed instruction, recording returns and the source and target addresses if the instruction is an explicit jump. To analyze jump instructions, Jima stores the register used to calculate the jump, then works backwards through the code, searching for the location where that register was set with a comparison or bound using logical operators. When working backwards to find the location of the register

calculation, Jima will stop the analysis if it finds the start of the function, or if more than 50 instructions are encountered.

The bulk of the function detection process begins with a sorted list of possible function addresses, found during the disassembly. The algorithm starts with the first address as a possible start address of a function and then processes instructions until the next address in the list is reached or if a function exit point is reached. If the last detected instruction is a sequence of NOPs (no operation) followed by a jump, the algorithm assumes it has found the end of the function. If there is a gap between the end of a function and the next address in the sorted list, the algorithm attempts to find the “missing function” as mentioned above.

The authors tested their function with three different datasets. The first dataset, Unix Utilities, deals with programs from various utils packages. The second dataset, SPEC CPU 2017, consists of a range of different programs written in C++, C, and Fortran. The third and final dataset used was the Chrome browser. Alves-Foss and Song compared the Jima algorithm against other similar algorithms mentioned above, Ghidra and Nucleus, as well as a few others. The results of the authors’ experiments showed that Jima was comparable to these other algorithms, outperforming most in terms of accuracy, but requiring a large amount of time and space, primarily due to the algorithm’s use of objdump generating a large amount of overhead.

Limitations/Future Work

One of the primary areas of improvement that Alves-Foss and Song mention is that of performance. By integrating disassembly into Jima without the use of the objdump command, the algorithm could be sped up significantly. Additionally, the algorithm struggled in some cases in which the compiler used different levels of optimization. Jima performed best with no optimizations in place, and performed the worst on binaries compiled with O2 and O3 optimizations.

The authors also mention that future work will involve porting the Jima tool to work on Windows PE binaries, as it currently only works on Linux ELF binaries.

Toward Smarter Vulnerability Discovery Using Machine Learning

The final system in this section of automated vulnerability detection does not deal with the detection of vulnerabilities itself. Rather, Grieco and Dinaburg present a system called Central Exploit Organizer (CEO) that deals with selecting optimal vulnerability detection tools. CEO is a machine learning based system that predicts the effectiveness of any given combination of a vulnerability detection tool (VDT), configuration, and input. Grieco and Dinaburg state that efficiency of selected VDTs can be improved over random selection by 11% to 21%. CEO was primarily tested on binaries from DARPA's Cyber Grand Challenge, using a variety of VDTs, such as fuzzing and symbolic executors. With regards to fuzzing, the authors state that CEO only deals with mutational fuzzers, those that modify existing program inputs, as the CEO system has the requirement of being provided an initial input.

Summary

Grieco and Dinaburg briefly give an overview as to why their system is useful for the cybersecurity community, explaining the drawbacks of typical VDTs. As fuzzers and symbolic executors are resource intensive, potentially consuming a massive amount of time and space only to produce no useful result, finding the optimal VDT is a valuable saver of time and effort.

The CEO system works by executing a target program with a given input, extracting features from the execution and identifying patterns in the program's behavior. CEO will then try to anticipate the result of the execution based on the program's behavior. Lastly, CEO will try to select the optimal combination of VDT, configuration, and input through the use of the system's machine learning techniques.

A large part of the authors' contribution to this system is their dataset used by CEO to optimize the selection of VDTs. The dataset consists of target programs, a set of VDTs, a random action generator, a labeling procedure, and a feature extraction process. As previously stated, the target programs were binaries obtained from DARPA's Cyber Grand Challenge. The

set of VDTs within the dataset only contained tools based on dynamic analysis, as static analysis does not take an initial input. The VDTs used were *Manticore*, a symbolic execution tool designed to find memory safety violations, *Grr*, a fuzzing tool, and *American Fuzzy Lop*, another fuzzing tool.

The action generator within Grieco and Dinaburg's dataset generates a random configuration and input for a given VDT on a target program. Each action generated was performed and labelled according to the authors' labeling procedure:

1. Failure: the VDT did not start or encounter a new program state
2. New Input: the VDT encountered a new program state
3. Valuable Input: the VDT encountered a vulnerable condition

Grieco and Dinaburg's feature extraction process deals with two classes: exec features, and config features. The exec features are extracted from the execution of the target program and consist of a sequence of assembly instructions, a sequence of system calls, a control flow graph of the target program, and transmitted words. The config features are simply the configuration options of the given VDT.

In the authors' evaluation section, they state that CEO was able to perform well with each of the three VDTs mentioned above. For *American Fuzzy Lop*, the system was able to identify the optimal combination of configuration and input with an average accuracy of 71%. For *Grr*, the result was an average accuracy of 61%. *Manticore* had an average accuracy of 70%.

Limitations/Future Work

One of the primary limitations of Grieco and Dinaburg's system and evaluation was the limited dataset of target programs. As the binaries used were a part of DARPA's Cyber Grand Challenge, there is a chance that each of the binaries are sufficiently different from each other that few patterns can be found in multiple binaries. Secondly, the authors state that some of the

VDTs used are non-deterministic. As such, there is the possibility of varied behavior in any given tool, even with an identical configuration and input.

Grieco and Dinaburg state that one of the first steps in improving CEO would be to include a more diverse set of larger binaries. This would allow for more accurate evaluation of the system. The authors are also investigating the use of other machine learning techniques, in the hopes of improving data labeling and feature extraction. They are additionally considering testing CEO with more VDTs.

Reverse Engineering

In the previous section, we showcase several systems that deal with automatic vulnerability detection, with some of the systems also dealing with vulnerability repair. This next section will highlight a few systems that deal with another side of software defense, reverse engineering. This is an important topic within cybersecurity as it is common for the source code of malware to be unavailable or unobtainable. Reverse engineering allows cybersecurity professionals to gain insight into how potentially malicious executables work without access to the source code. This allows for better development of defenses.

Reverse Engineering of ARM Binaries

The first reverse engineering paper within this section is *Reverse Engineering of ARM Binaries Using Formal Transformations*, written by Pfeffer et al. In this paper, the authors aim to create an approach for reverse engineering ARM binaries. The goal of this reverse engineering process is to produce a high-level, well-comprehensible abstraction of the program. The authors also aim to have this process automated and require no information aside from what the binary provides. Additionally, the process is intended to semantically preserve the original program and run within a reasonable time.

Summary

Because few approaches to decompiling ARM binaries--as opposed to binaries of the x86 architecture--exist, Pfeffer et al. make use of program transformation rules. Specifically, they make use of the FermaT transformation system, developed by Martin Ward. This transformation system will apply transformation rules to code given in the Wide Spectrum Language (WSL), transforming the code into a more readable version, allowing for easier analysis. As the FermaT system has never before been used on ARM binaries, Pfeffer et al. first apply translation rules to transform the ARM assembly into WSL, before feeding the WSL representation to FermaT.

During the transformation process, each line of assembler instructions is translated into one WSL action that preserves its semantic effects. All of these created WSL actions together create an overarching action system that represents the original code. Here, the authors split this action system into separate procedures, maintaining control flow of the original code. This process is highly modular and reusable, another goal the authors had in mind.

The translation process from ARM to WSL makes use of a model of assembler representation described by Ward and Bennett, referenced in the paper. This model uses one WSL action per line of code to represent the assembler instructions. Additionally, this model makes use of global variables to represent the 16 ARM registers. Memory is represented as an array. Condition codes are split into four variables: N (Negative), Z (Zero), C (Carry), and V (oVerflow).

The translation rules themselves are made up of three parts. The first is an expression used to represent the rule itself and accounts for the semantic effects of the instruction. The second is an epilogue which models the control flow of the code. The last is a list of status bits. For some of the more complicated instructions, such as conditional execution, the condition codes along with IF/ELSE logic within the epilogue are used. Access to memory is either direct or with an offset. Some of the specific instructions, such as CMP and CMN, are translated using simple arithmetic instructions in conjunction with temporary variables used to set the CPSR bits.

Once the code is fully translated into WSL with its semantic effects preserved, the WSL representation is fed to FermaT. The primary goal throughout this part of the process is control flow recovery, gaining a representation of the code that is free from any control flow obscuring code. This allows the representation to be easily restructured into high level control flow structures. Here, Pfeffer et al. apply six rephrasing transformations:

- Removal of the program counter, connecting the actions within the system together.
- Flag removal, made possible due to the code now being a regular action system and removing the need for condition checks.
- Splitting of the action system into separate processes.
- Removal of the link register from each process, changing the system into a non-regular action system.
- Removal of all calls to unidentifiable labels and collapsing of the action systems, recovering control flow.

To evaluate the quality of their approach, the authors applied their process on binaries chosen from the Debian coreutils package. Plotting the size of the binaries against the time taken by the transformation process, the authors showed that the process has a linear runtime, fulfilling the goal of reasonable runtime. Additionally, the size of the resulting representation of the code is generally larger than that of the original, but not unreasonably so. In terms of quality of the transformation, Pfeffer et al. provide an example of a translation, stating that the result is significantly more comprehensible, claiming this as a success of their process.

Limitations/Future Work

For the most part, Pfeffer et al. achieved all of the goals laid out at the beginning of their paper. Their process is modular, reusable, automated, applicable within a reasonable time, and accurate while also giving the results a fair amount of readability. While the process provides control flow recovery, the main area of future work the authors list is the inclusion of data flow recovery.

Reverse Engineering of Types in Binary Programs

Lee et al. present TIE, Type Inference on Executables, in their paper, *TIE: Principled Reverse Engineering of Types in Binary Programs*. TIE is an end-to-end reverse engineering system with the goal of inferring the most amount of accurate information on variable types given binary code. Lee et al. compare their system against a couple of other similar systems, such as REWARDS and Hex-rays, and claim that TIE consistently yields more conservative and precise results when it comes to identifying variable types. The authors define conservativeness as never inferring more information than is available in the binary, in other words, not guessing types. Precision is how close the inferred type is to the original type. The system was run on 87 programs from coreutils, with results backing up the authors' claims of higher conservativeness and precision than other systems.

Summary

The start of the process of TIE uses a binary analysis platform, BAP, to convert the binary code into a binary analysis language called BIL. BAP is able to consider both static analysis scenarios and dynamic analysis scenarios, another advantage of TIE over other similar systems. For static analysis, the binary is disassembled and functions identified. For dynamic analysis, the program is run and the instructions executed are output. In both scenarios, an assembly program is the result, with dynamic analysis giving the single path actually executed. These results are converted to BIL for further analysis, with a mapping to the original assembly kept to allow final results in terms of the original assembly.

The next step in the process is variable recovery and type reconstruction. This phase takes the BIL code produced as input. Variables are inferred through analysis of memory access patterns. These recovered variables are then passed to TIE's type reconstruction algorithm along with the BIL code. The algorithm will first assign each variable a placeholder type, before generating a constraint on the possibilities of the variables type. These constraints

are determined by things like how the variable itself is used, what operations are performed on the variable, etc. The constraints are then solved, finding the most precise yet conservative range of possible types for the variable. The types inferred through this algorithm are within the TIE type system. This is a system of types that can be easily converted to C types. Another advantage of TIE is that this system of types can be retargeted to convert to other languages, thus TIE is not limited to just reverse engineering the language of C.

In addition to TIE's type reconstruction algorithm, the system also has an inter-procedural constraint generation step. This is a pre-processing step in which TIE creates a context *F* of known functions. When function is called, TIE first searches *F* for the function description. If a match is found, the function's parameters and return type can be used to aid in the recovery of the variables within the function call.

The final step in TIE is constraint solving. In this step, a list of remaining unsolved constraints is kept and drawn from to find the next type to be recovered. Each entry in this list is removed and processed to find upper and lower bounds to the variable's type. Once this list is empty, each constraint is considered to be solved. The algorithm terminates after all constraints are solved.

Limitations

While TIE is versatile in that it is not limited to just C, it is somewhat limited to the x86 architecture, due to it relying on things like memory access patterns and calling conventions to determine type ranges. Additionally, many of the results from this system's algorithm will only yield a range of possible types for recovered variables. Thus, in some cases it is not possible for TIE to give a complete reverse engineering given a binary. However, this limitation is not necessarily exclusive to TIE and applies to most reverse engineering in general.

A Decompilation Framework for Static Analysis of Binaries

The final system we discuss is REcompile, detailed in the paper, *REcompile: A Decompilation Framework for Static Analysis of Binaries*, written by Yakdan et al. REcompile is a decompilation framework that transforms low-level machine code into a high-level representation, allowing for easier readability. The REcompile system uses a variety of techniques to regain some of the information lost—variable names, types, etc.—during compilation. Yakdan et al. tested their framework against real malware samples and compared the results to that of a state-of-the-art decompiler.

Summary

As malware is becoming increasingly complex, Yakdan et al. created REcompile with the goals of readability, extensibility, and modularity in mind. Being able to generate more readable representations of code is particularly useful when the given compiled code would be nearly impossible to decipher due to its size or complexity. Extensibility allows the REcompile system to grow and change along with the malware it is used against. Finally, modularity divides the system into explicit input, output, and function components to aid the development process.

REcompile is comprised of five main components: an intermediate representation (IR) generator, a data flow analyzer, a type analyzer, a control flow analyzer, and a code generator. The intermediate representation generator takes the input program and transforms it into REcompile's IR. REcompile's IR uses a *static single assignment* form. This is a type of representation in which every variable has a single definition. The data flow analyzer performs code optimizations, removes dead code—such as a variable that is defined but never used again—and identifies parameters and return values of functions. The type analyzer determines the types of variables by examining how the variables are used in the program. For example, an assignment instruction such as $x = y$ would show that the variables are of the same type. Thus, if y is known, the type of x could be inferred. The control flow analyzer is responsible for the reconstruction of high-level language controls structures, such as loops and conditionals. Finally, the code generator generates the decompiled code once all analyses are completed.

Yakdan et al. evaluated the REcompile system in two tests. The first test was performed with samples with available source code, to allow comparison between the output of REcompile with the given input. The second test compared REcompile with the Hex-Rays decompiler. Both decompilers were evaluated on a large set of real malware samples. Within these tests, performance was measured based on functional equivalence, structuring efficiency, and reduction ratio. The first metric, functional equivalence, was simply measuring how closely the decompiled output resembled the functionality of the given input. The second metric measured how well high-level control structures could be represented. The final metric measured the difference in size between the input and output. The results of the tests showed that REcompile achieved functional equivalence on programs with functions of varying complexity, performed slightly better than the Hex-Rays decompiler in terms of structuring efficiency, but underperformed slightly in terms of reduction ratio.

Limitations/Future Work

The authors state the REcompile is implemented as a plugin for IDA, a commonly used disassembly. Thus, the functionality of REcompile is dependent on IDA. If IDA were to be fooled by any anti-disassembly techniques, REcompile would incorrectly decompile the code. However, due to the modularity aspect of REcompile, the authors state that it could easily be ported to also be used with other disassemblers.

Conclusion

Within this paper, we show several systems each created with the intention of addressing automatic vulnerability detection, vulnerability repair, or reverse engineering of executables. Each of these systems of course have limitations, some of which are listed after discussing a general overview of that particular system.

Though many of the systems differ in various ways, they are each evaluated by their authors and are typically found to perform their particular task or goal reasonably well. Some

systems, such as Cui et al.'s ShieldGen, perform exceptionally well when given particular executables on which to act, but lack versatility. Other systems, such as the ClearView system created by Perkins et al., are able to deal with a lack of versatility by being highly modular. In the example of ClearView, the system is able to adapt to cover a wider range of vulnerabilities by simply adding more monitors in addition to HeapGuard, ShadowStack, and Determina Memory Firewall already in place. Given the complexity and rapid growth and change of malware, this high modularity of some of the systems detailed within this paper is what sets them apart from other systems of similar natures.

Several of these systems were evaluated with executables from DARPA's Cyber Grand Challenge. While it is certainly impressive for those systems to perform as well as they did given the complexity of the executables and difficulty of the challenge overall, the executables are not entirely representative of malware that the systems would encounter in the real world. As a result, these systems that deal with CGC executables could benefit greatly from further evaluation, using real world malware similar to some of the other systems discussed.

Overall, the systems discussed throughout this paper and the generally positive results from their authors' evaluations offer an optimistic outlook of the future of cybersecurity. As previously mentioned, software is continually evolving and becoming more complex, rendering manual detection of software vulnerabilities increasingly obsolete. These systems show that automatic detection and repair is a reliable and accurate alternative, with room to match the rapid growth of malware.

Sources

Differentiating Code from Data in x86 Binaries

Wartell R., Zhou Y., Hamlen K.W., Kantarcioglu M., Thuraisingham B. (2011) Differentiating Code from Data in x86 Binaries. In: Gunopulos D., Hofmann T., Malerba D., Vazirgiannis M. (eds) Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2011. Lecture Notes in Computer Science, vol 6913. Springer, Berlin, Heidelberg

Trust in Automated Software Repair

The Effects of Repair Source, Transparency, and Programmer Experience on Perceived Trustworthiness and Trust

Ryan T.J., Alarcon G.M., Walter C., Gamble R., Jessup S.A., Capiola A., Pfahler M.D. (2019) Trust in Automated Software Repair. In: Moallem A. (eds) HCI for Cybersecurity, Privacy and Trust. HCII 2019. Lecture Notes in Computer Science, vol 11594. Springer, Cham

Automatically Patching Errors in Deployed Software

Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard

SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, October 2009, Pages 87-102

ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing

W. Cui, M. Peinado, H. J. Wang and M. E. Locasto, "ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing," *2007 IEEE Symposium on Security and Privacy (SP '07)*, Berkeley, CA, 2007, pp. 252-266.

FuzzBomb: Fully-Autonomous Detection and Repair of Cyber Vulnerabilities

Musliner, David & Friedman, Scott & Boldt, Michael & Benton, J. & Schuchard, M & Keller, P. (2015). FuzzBomb: Autonomous Cyber Vulnerability Detection and Repair.

FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries

S. Alrabaee, P. Shirani and L. Wang, "FOSSIL: A resilient and efficient system for identifying FOSS functions in Malware binaries", *ACM Trans. Privacy Secur.*, vol. 21, no. 2, pp. 1-34, 2018.

Function Boundary Detection in Stripped Binaries

J. Alves-Foss & J. Song, ACSAC '19: Proceedings of the 35th Annual Computer Security Applications Conference, December 2019, Pages 84–96, https://doi.org/10.1145/3359789.3359825

Toward Smarter Vulnerability Discovery Using Machine Learning

G. Grieco & A. Dinaburg, AISec '18: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, January 2018, Pages 48–56, https://doi.org/10.1145/3270101.3270107

Reverse Engineering of ARM Binaries Using Formal Transformations

T. Pfeffer, P. Herber, and J. Schneider, SIN '14: Proceedings of the 7th International Conference on Security of Information and Networks, September 2014, Pages 345–350, https://doi.org/10.1145/2659651.2659697

TIE: Principled Reverse Engineering of Types in Binary Programs

J. Lee, T. Avgerinos, and D. Brumley (2011). TIE: Principled Reverse Engineering of Types in Binary Programs. *NDSS*.

REcompile: A Decompilation Framework for Static Analysis of Binaries

K. Yakdan, S. Eschweiler and E. Gerhards-Padilla, "REcompile: A decompilation framework for static analysis of binaries," 2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), Fajardo, PR, 2013, pp. 95-102, doi: 10.1109/MALWARE.2013.6703690.

Risk Based Security

https://pages.riskbasedsecurity.com/2019-midyear-data-breach-quickview-report

Security Intelligence

L. Ponemon,

https://securityintelligence.com/posts/whats-new-in-the-2019-cost-of-a-data-breach-report/

Defense Advanced Research Projects Agency, Cyber Grand Challenge

https://www.darpa.mil/program/cyber-grand-challenge