EXPLORING SEMANTIC REVERSE ENGINEERING FOR SOFTWARE BINARY PROTECTION

by

PENGFEI SUN

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Saman Zonouz

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

MAY, 2019

ABSTRACT OF THE DISSERTATION

Exploring Semantic Reverse Engineering for Software Binary Protection

By Pengfei Sun

Dissertation Director:

Saman Zonouz

Semantic reverse engineering has become the main approach to explore and understand the big picture of the binary code for closed-source software packages. However, semantic reverse engineering still has two unsolved challenges: (1) to recognize and recover data structure instances from binary memory images without execution traces; and (2) to locate the critical algorithm implementation and extract the high-level semantic meaning for the associated memory addresses/registers. These capabilities have many computer security and forensics applications, such as vulnerability discovery, sensitive data protection and so on.

In this dissertation, I present new techniques to perform automatic semantic reverse engineering to address the above-mentioned challenges. First, I present a systematic framework, ReViver, for semantic reverse engineering of data structure instances from live memory without execution trace. Using the discovered data structure instances in live memory, I develop a new domain-specific semantic memory data attack against

power grid controllers. What's more, I propose a framework, Mismo, to analyze embedded system binaries to extract semantic information about the control algorithms that they implement. Finally, I build BinSec, a vulnerability assessment tool which leverages deep learning and dynamic analysis to do cross-platform binary code similarity detection to identify known vulnerabilities. I demonstrate how I integrate these new techniques to explore semantic information for binary protection and exploitation.

I have obtained the following experimental results. ReViver achieved 98.1% average accuracy in recovering memory data structure instances without execution traces for real-world applications. Mismo's accuracy for data discovery was an average of 89.82%, and 84.96% for code and data semantics discovery, respectively. For BinSec, I evaluate 25 existing CVE vulnerability functions for the Google Pixel 2 smartphone and Android Things IoT firmware images. The deep learning model identifies vulnerabilities with an accuracy of over 93% and the dynamic analysis can help to identify the correct matches among the top 3 ranked outcomes 100% of the time.

Acknowledgements

First and foremost, I would like to express an endless amount of gratitude and appreciation to my advisor Prof. Saman Zonouz who gave his extraordinary guidance, support, and encouragement throughout my entire PhD study. In particular, he provided me the plenty of freedom to do the research I was fascinated, and the mental support when I needed it the most. He guided me in working on the research for the right direction and stayed late for reviewing and evaluating our research work. He has built an incredible example of a successful life-work balance for us, and he has always prioritized our mental and body health. I am so fortunate of having him as my advisor in my life.

I am also extremely grateful to my committee members, Prof. Ivan Marsic, Prof. Sheng Wei and Dr. Praveen Murthy. This dissertation greatly benefitted from their careful reading, insightful comments, and high standards. Also, I have learned invaluable lessons from interacting with them.

Finally, I would like to thank my family and friends for their incredible support throughout this entire chapter in my career. Thanks for my parents, Jijun Sun and Shufeng Zhang, and my sister, Qiaoling Sun, my brother-in-law, Hui Wang, for their unlimited understanding and support my life. Thanks for my wife, Jiachuan Wu and my wife's parents, Hui Wu and Yuelian Pan, for their love and continuous encouragement. Especially, for my wife Jiachuan, she has been with me all these years and always understands, loves, and stands by me. Without your support, I cannot reach this point. I hope I have graduated to be a better husband, a better son and a better friend.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables viii

List of Figures ix

1. Introduction 1

1.1. Contributions 2

1.2. Organization 6

2. Trace-Free Memory Data Structure Forensics via Past Inference and Future Speculations 8

2.1. Introduction 8

2.2. ReViver Overview 10

2.3. Past: Statistical Information 17

2.4. Present: Static Memory Analysis 20

2.5. Future: Speculative Forensics 27

2.6. Evaluations 30

2.7. Related Work 39

2.8. Discussions and Limitations 40

2.9. Conclusions 42

3. Compromising Security of Economic Dispatch in Power System Operations 43

3.1. Introduction 43

3.2. Optimal Attacks to Economic Dispatch 49
3.3. Characteristics of Optimal Attack 56
3.4. Computational Results 58
3.5. Implementations 63
3.6. Empirical Attack Deployment Results 67
3.7. Discussions and Potential Mitigation 71
3.8. Related Work 72
3.9. Concluding Remarks 73

4. Tell Me More Than Just Assembly! Reversing Semantics of Embedded IoT and Industrial Control Software Binaries 75
4.1. Introduction 75
4.2. Threat Model 80
4.3. System Overview 81
4.4. Mismo Design 83
4.5. Implementation and Case-Study 90
4.6. Evaluations 97
4.7. Related Work 109
4.8. Conclusions 111

5. I Know What You Didn't Do Last Vulnerability! Firmware Analysis via Deep Learning for Known Security Vulnerabilities 112
5.1. Introduction 112
5.2. Overview 115
5.3. Design 118
5.4. Implementation and Case-Study 127
5.5. Evaluation 134
5.6. Related Work 140
5.7. Discussion and Conclusion 141

6. Conclusion 143

Bibliography 145

List of Tables

2.1. Appendix: Applications' Name-Index Mappings 30

2.2. Results Improvement via speculative execution 36

3.1. Optimal attacker strategy for three-bus test case. 60

3.2. Logical memory structure signatures for critical parameters. 63

3.3. The target parameter value recognition accuracy. 67

3.4. Memory layout (object) forensics accuracy 69

4.1. Result of Comparing Two ASTs 95

4.2. Embedded IoT/CPS firmware vendors 99

4.3. Embedded applications with control algorithm implementations. 101

4.4. Comparison among different approaches 103

4.5. Comparing the reverse engineering results 104

5.1. Function features used in BinSec. 119

5.2. Dynamic features used in BinSec. 124

5.3. The dynamic feature vector profiling 131

5.4. Calculating Function Similarity in BinSec 132

5.5. Calculating Function Similarity 133

5.6. The accuracy based on vulnerable function 138

5.7. The accuracy based on patched function 138

5.8. The final patch detection results for BinSec in Android Things 139

List of Figures

2.1. ReViver's high-level architecture 12

2.2. Memory snapshot of the `groups` application and result 13

2.3. Memory reverse engineering through past information 16

2.4. Memory data type reverse engineering accuracy 32

2.5. Speedup via directed symbolic execution 32

2.6. Forensics accuracy and time requirement 33

2.7. Orzhttpd memory snapshot and reversed data structure 37

2.8. Orzhttpd's post-attack modified root directory 40

3.1. Physics-aware memory attack on control systems. 46

3.2. Static vs dynamic line rating 52

3.3. Three-bus power system. 59

3.4. Results for three-node power grid. 59

3.5. Results for 118-node network 62

3.6. Flowchart for attack implementation. 63

3.7. Code and data pointer-based structural memory patterns 64

3.8. PowerWorld and Powertools controller software attack results 70

4.1. Overview of Mismo framework. 81

4.2. High-level block diagram of a sample embedded CPS control algorithm (Kalman filter). Mismo will map algorithmic logic and parameters of the diagram to their corresponding binary-level control flows and memory variables, respectively. 91

4.3. The identified controller functions 93

4.4. The top candidate of paths selected by Mismo selector 94

4.5. Mapping executable- to algorithm-derived ASTs 94

4.6. Last set of instructions and data flow analysis 97
4.7. MISMO provides semantically rich information for IDA Pro view 98
4.8. Number of symbolic variables and the data sources. 102
4.9. Accuracy of data and code semantics discovery. 102
4.10. MISMO analysis time on 10 real world applications. 102
4.11. MISMO detects the bug in Linux Kernel. 106
4.12. Bug of Linux Kernel PID 106
4.13. Abstract syntax tree for PID algorithm 107
4.14. Comparing controller output 107
4.15. Car crash example 108
5.1. BINSEC vulnerability and patch search workflow. 117
5.2. Detailed BINSEC Architecture for static analysis 120
5.3. Training the neural networks 121
5.4. Concrete execution of potentially vulnerable code segments 122
5.5. Detailed BINSEC architecture for dynamic analysis 125
5.6. Vulnerable code with the associated patch of CVE-2018-9412. 129
5.7. Deep learning training result. 136
5.8. False positive rate on Android Things 136

Chapter 1

Introduction

Binary analysis is a challenging and recurring research area in computer security. In many situations, binary analysis is the only possible way to prove or disclose the properties of the code for closed-source software packages. Access to source code is often unavailable for third-party security analysis of commercial off-the-shelf (COTS) binaries. As the compiler does not preserve a lot of language-level information, the lack of high-level, semantic information about data structures and control constructs makes the binary analysis harder to scale. Therefore reverse engineering is necessary to recover the high-level information about the original source code from binaries. It can help to improve the understanding of the underlying source code for the maintenance and the improvement of the software. Relevant information can be extracted to make a decision for software development and to detect and fix a software bug or software vulnerability. What's more, from the attack perspective, reverse engineering can help the attacker to figure out the possible way to attack the system.

Binary analysis tasks can vary from reliable disassembly of instructions to recovery of control-flow, data structures or full functional semantics. Advanced disassembler and debugger tools such as IDA Pro Rescue (2006) and OllyDbg Yuschuk (2007) offer a variety of techniques to help elevate low-level machine codes to more abstract representations. However, such static binary analysis tools mainly extract syntactical information which only reveals small part of the binary. The semantics of the original program are not guaranteed to be preserved/extracted. However, developers put their domain knowledge into exactly these parts of the source code. Without understanding the semantics of the code, one cannot tell its meaning.

Semantic reverse engineering becomes the main approach to explore and understand

the big picture of the binary code. For semantic reverse engineering, recovering critical algorithms and data structures have become the main goal since software development is essentially composed of data structures and algorithms. The data structure is a particular way of storing and accessing data for programs. And it is composed of a number of fields, and each field has a specific type. The algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions.

There are two representations for data structures in reverse engineering: (1) abstract representation: source code-level definition of the data structure during software development; (2) concrete representation: the memory instance of the data structure during program execution. The discovered data structures enable higher level operations such as algorithm reverse engineering in binaries that are used for specific domains such as industrial control systems. Algorithm reverse engineering is to disclose the domain-specific data- and control-flow to learn about the critical components that if compromised in an attack would result in considerable failures, e.g., undesired industrial control system incidents. This is because the implemented critical algorithms act as functional guarantees for the entire system, such as cyber-physical control system. More specifically, we aim to reverse engineer the data structure definitions and the data- and control-flow of algorithms from binary code, as well as recognizing data structure instances from a memory image. These semantic reverse engineering capabilities have many computer security and forensics applications, such as vulnerability discovery, exploit generation, memory forensics, malware classification and sensitive data protection and so on.

1.1 Contributions

Prior to this work, other researchers considered data structure definitions recovery from an application binary based on static and dynamic analysis Song et al. (2008); Brumley et al. (2011); Chikofsky et al. (1990); Saltaformaggio et al. (2015b). This, however, was far from sufficient, as researchers were still unable to recognize and recover data structure instances from a memory image, especially when the execution trace is not available. Therefore, here are a number of new challenges in (1) recognizing and

recovering data structure instances from a memory image without execution traces, and (2) locating the critical algorithm implementation and extracting the semantic meaning for the associated memory addresses/registers.

Given these challenges, this thesis will develop new techniques to perform automatic semantic reverse engineering that recovers data structure instances from live memory and extracts the critical algorithm implementation. First, we present a systematic framework, ReViver, for semantic reverse engineering of data structure instances from live memory without execution trace. Based on data structure instances in live memory, we implemented a new domain-specific semantic stealthiness/memory data attack against power grid controllers. Further, we propose a framework, Mismo, to analyze embedded system binaries to extract semantic information about the control algorithms that they implement. What's more, we explore the semantic information and integrate deep learning and dynamic analysis to detect known vulnerability in mobile/IoT firmware. Below we will briefly introduce these techniques, the technical contributions made by each, and the unique challenges that they overcome.

1.1.1 Trace-Free Memory Data Structure Forensics

A yet-to-be-solved but very vital problem in forensics analysis is accurate memory dump data type reverse engineering where the target process is not a priori specified and could be any of the running processes within the system. We present ReViver, a lightweight system-wide solution that extracts data type information from the memory dump without its past execution traces. ReViver constructs the dump's accurate data structure layout through collection of statistical information about possible *past* traces, forensics inspection of the *present* memory dump, and speculative investigation of potential *future* executions of the suspended process. First, ReViver analyzes a heavily instrumented set of execution paths of the same executable that end in the same state of the memory dump (the eip and call stack), and collects statistical information the *potential* data structure instances on the captured dump. Second, ReViver uses the statistical information and performs a word-by-word data type forensics inspection of the captured memory dump. Finally, ReViver revives the dump's execution and explores

its potential future execution paths symbolically. REVIVER traces the executions including library/system calls for their known argument/return data types, and performs backward taint analysis to mark the dump bytes with relevant data type information. REVIVER's experimental results on real-world applications are very promising (98.1%), and show that REVIVER improves the accuracy of the past trace-free memory forensics solutions significantly while maintaining a negligible runtime performance overhead (1.8%).

1.1.2 Case-study: Data Attacks against Power System Operations

We demonstrated our contributions in controller software that are widely used in critical infrastructures. We use our solutions to reverse engineer the corresponding software and leverage the extracted semantics to launch an attack vector.

Power grid operations rely on the trustworthy operation of critical control center functionalities, including the so-called Economic Dispatch (ED) problem. The ED problem is a large-scale optimization problem that is periodically solved by the system operator to ensure the balance of supply and load while maintaining reliability constraints. We propose a semantics-based attack generation and implementation approach to study the security of the ED problem. Firstly, we generate optimal attack vectors to transmission line ratings to induce maximum congestion in the critical lines, resulting in the violation of capacity limits. We formulate a bilevel optimization problem in which the attacker chooses manipulations of line capacity ratings to maximinimize the percentage line capacity violations under linear power flows. We reformulate the bilevel problem as a mixed integer linear program that can be solved efficiently. Secondly, we describe how the optimal attack vectors can be implemented in commercial energy management systems (EMSs). The attack explores the dynamic memory space of the EMS, and replaces the true line capacity ratings stored in data regions with the optimal attack vectors. In contrast to the well-known false data injection attacks to control systems that require compromising distributed sensors, our approach directly implements attacks to the control center server. Our experimental results on benchmark power systems and five widely utilized EMSs show the practical feasibility of our

attack generation and implementation approach.

1.1.3 Reversing Semantics of Embedded IoT Software Binaries

The security of critical IoT devices and industrial control systems hinges on their embedded software that implement control algorithms for monitoring and control of the associated physical processes, e.g., robotics and drones. Reverse engineering of the corresponding embedded controller software binaries enable their security analysis by extracting high-level domain-specific semantic information from executables. We present Mismo, a domain-specific reverse engineering framework for embedded binary code in emerging IoT and cyber-physical control application domains. Mismo performs semantic-matching at an algorithmic level that can help with the understanding of any possible cyber-physical security flaws. The extracted semantic information can be leveraged for fine-grained protection of sensitive data in embedded software. Mismo compares low-level binary symbolic values and high-level algorithmic expressions to extract domain-specific semantic information for the binary's code and data. We evaluated Mismo on popular firmware binaries by 10 commercial vendors from 6 application domains including drones, self-driving cars, smart homes, robotics, 3D printers, and the Linux kernel controllers. The results show that Mismo can accurately extract the algorithm-level semantics of the embedded binary code and data regions. We discovered a zero-day vulnerability. in the control algorithm implementation within Linux kernel versions 3.13 and above.

1.1.4 Firmware Analysis via Deep Learning for Known Security Vulnerabilities

Because of the increasing security problems in mobile/IoT devices, one central pillar of keeping these devices secure is providing regular patches. As such, it is critical to ensure that patches are propagated to all affected software in a timely fashion. However, several studies have shown that there is a significant *hidden patch gap*–where several vendors are falsely reporting patches of vulnerabilities. Therefore, it is much more critical to accurately ensure whether the vulnerability has indeed been patched.

Existing approaches rely on approximate graph-matching algorithms, which are slow and inaccurate. Alternatively, deep learning approaches have been proposed which are engineered for speed at the expense of accuracy. This is problematic for large binaries that include a large number of functions. In this paper, we present BinSec, a vulnerability assessment tool which leverages deep learning and dynamic analysis to do cross-platform binary code similarity detection to identify known vulnerabilities with high accuracy. In addition to identifying known vulnerabilities, BinSec also identifies vulnerabilities that have been patched within the same target binary. BinSec does not require access to the source code of vulnerability functions nor the target binary. We evaluate BinSec on 25 existing CVE vulnerability functions for the Google Pixel 2 smartphone and Android Things IoT firmware images. Our deep learning model identifies vulnerabilities with an accuracy of over 93%, i.e., higher than the state-of-the-art. We then demonstrate how dynamic analysis of the vulnerability functions in a controlled environment can be used to significantly reduce the number of candidate functions and, thus, the number of false positives. BinSec identifies the correct matches (candidate functions) among the top 3 ranked outcomes 100% of the time. Finally, we evaluate BinSec's differential engine that distinguishes between functions that are vulnerable and functions that are patched on the same dataset with the same level of accuracy.

1.2 Organization

This dissertation will present the evolution of this body of work. It highlights the progression of semantic reverse engineering and how to use this semantic knowledge for protection and exploitation. The outline of this dissertation is as follows:

- Chapter 1 explains the need for our semantic reverse engineering framework to tackle a number of challenges to extract high-level properties of the binaries without source code access. We present our framework with a specific focus on semantic reverse engineering for data structure instances in live memory and critical algorithm implementation in ICS/IoT domain.

- Chapter 2 explains in detail the motivation, design, implementation, and evaluation of ReViver. ReViver leverages static and dynamic analysis to resolve data structure instances from live memory without execution traces requirement.

- Chapter 3 presents a case study attack, i.e., a new domain-specific semantic stealthiness/memory data attack against power grid controllers. Since ReViver has given us the memory data structure knowledge, we leverage this knowledge to develop automated data attacks and modify sensitive parameters of the implemented power system control algorithms. As the result, the underlying power system enters an unsafe state.

- Chapter 4 details Mismo, a domain-specific reverse engineering solution to extract high-level algorithmic control and data flow semantics from embedded binary executables in various IoT and cyber-physical industrial control applications.

- Chapter 5 presents BinSec, a framework that integrates deep learning for binary similarity-checking with dynamic analysis to discover known vulnerabilities as well as to test for path presence without source code access. The framework works cross-platform supporting ARM and X86 architectures. We have evaluate the framework on 25 CVE vulnerabilities, 100 different Android firmware libraries across 4 different architectures.

- Chapter 6 discusses possible future research directions along with the conclusion of this proposal.

Chapter 2

Trace-Free Memory Data Structure Forensics via Past Inference and Future Speculations

2.1 Introduction

Software reverse engineering has been a challenging and recurring problem in computer security that aims at recovery of high-level program abstractions Song et al. (2008); Brumley et al. (2011); Chikofsky et al. (1990); Saltaformaggio et al. (2015b). The results could potentially be used for various purposes such as memory forensics, malware development and analysis, reversing cryptographic algorithms and network protocols, digital right management, and auditing program binaries. Specifically, a desirable capability in many of those security and forensics applications is automatic reverse engineering of data structures given only the memory dump of a process and without the execution trace information and hence the need for heavyweight runtime instrumentation. Such capabilities can reveal semantic information about any execution point of a given program through investigation of its memory space for meaningful data types, their interdependencies, sizes and runtime values. Such memory-level data type information can be linked to higher level concepts, e.g., global configuration variables, confidential credentials and private keys, still-in-memory temporary data from past sessions such as closed web browser tabs.

The existing data type reverse engineering solutions are categorized into three groups. First, static binary executable analysis techniques extract data structures defined within an executable through disassembly Lee et al. (2011) or symbolic execution Slowinska et al. (2011). Second, dynamic execution analysis solutions Lin et al.

(2010); Lee et al. (2011) trace the execution to reverse engineer data types using type-revealing instructions Lin et al. (2010). Third, static memory analysis techniques perform forensics directly on memory dumps for data structures. A past work by Cozzie et al. Cozzie et al. (2008) sweeps the memory for pointers and assumes all pointers point at other data structures.

The past work falls short for practical forensics analysis of applications' memory dumps: *i)* static executable analyses can reverse engineer executable-defined data structure definitions accurately; however, those approaches by themselves are of limited use for memory forensics when the execution trace is not available; *ii)* dynamic execution monitors cause a very high performance overhead $> 6X$ on the target process. Hence, it is infeasible to trace all the running processes on a system as any of them may be misbehaving and needing forensics analysis; and *iii)* the static memory analyses are not sufficiently accurate in practice (e.g., 70% for Laika Cozzie et al. (2008)).

We present ReViver, a hybrid memory forensics solution that leverages the high accuracy of static executable analyses (i) and dynamic execution monitoring (ii) to provide a low overhead and precise static memory dump forensics (iii) when the execution trace is not available. ReViver creates a comprehensive database of data structure definitions automatically through a one-time effort. ReViver investigates potential *past* histories that could have led to the captured dump state. This creates the statistical information about the possible data structure layout of the given dump. ReViver investigates the *present* captured dump using the data structure definition database and the collected statistical information. Finally, ReViver revives the dump's execution and traverses through the potential *future* execution paths symbolically. ReViver monitors each path closely for type-revealing instructions, and performs backward taint analysis to backtrack the data type information ideally to a memory address on the captured dump. This helps to either confirm its former analysis results or correct a wrongly inferred subset of memory-resident structures. Through combining results from various analysis techniques, ReViver produces a highly accurate data structure layout of the memory without the need heavyweight binary/execution instrumentation of running processes.

Our contributions are thus the following:

• We introduce a trace-free memory data structure forensics solution with high reverse engineering accuracy and negligible 1.8% runtime overhead.

• We present a probabilistic information fusion method to combine prior statistical information about possible past traces, results from the present dump forensics and speculative investigation of potential future executions of the suspended process.

• We evaluated a working prototype of ReViver on real-world settings (i.e., CoreUtils suite, and five popular desktop and server applications). ReViver works on stripped binary executable and generate memory data structure layout of a given capture memory dump.

It is noteworthy that ReViver's symbolic execution often cannot fully exhaust the executables in practice. However, this does not affect ReViver's correctness, because ReViver needs just an empirical estimate of the statistical information for its later forensics analysis. Even incomplete symbolic execution code coverage provides ReViver with extra and useful information to improve its ultimate forensics accuracy.

Threat model. ReViver assumes the non-compromised root privilege on the target system. ReViver's TCB includes the analysis system where the memory forensics is performed.

2.2 ReViver Overview

Figure 2.1 shows ReViver's high-level architecture that contains a forensics analysis framework to where the user uploads the target process'es memory image and the application executable. The process starts by ReViver's installation on the user's machine. Upon detection of a suspicious running process by the user or an intrusion detector, ReViver's client suspends the process and uploads its memory image for forensics analysis.

To accelerate the runtime memory forensics (Section 2.4), ReViver hooks the dynamic memory allocation API (e.g., `malloc`). The hooked functions allocate the required memory, and additionally, adds a rare data pattern landmark signature as well

as the allocation size at the end of allocated region. This causes a negligible runtime overhead (1.8%) but improves the memory forensics significantly. All data structure locations and sizes (not their type) on the memory can be inferred later through a quick linear memory sweep for the incorporated signatures.

ReViver performs its forensics analysis through the following three stages: *past*, *present*, and *future*.

Past. ReViver leverages offline analysis to maximize its runtime analyses' efficiency once a particular running process is picked for analysis. To create the data structure definition database, ReViver leverages the past work Lin et al. (2010) to extract the application executable-defined data structure definitions, and implements automated static code analysis modules to investigate available library and kernel sources searching for data structure definitions. There often exists different data structures with identical names defined at different points across the system. ReViver maintains the context where each data structures is defined. ReViver explores and analyzes the memory dump (memdump)'s possible past execution trace. ReViver starts from the executable's entry point, and statically explores the call and control flow graphs for possible paths to the captured dump's execution state (the instruction pointer and the call stack). Using symbolic execution, ReViver filters out the infeasible paths and generates the corresponding test cases. Through an instrumented execution of the test cases, ReViver logs the structure memory allocations and collects statistical information about the potential structures on the captured dump. For instance, if a particular data structure exists in almost all test case executions, it exists in the captured dump with higher probability than a data structure that was never encountered during the test case executions.

Present. ReViver performs a forensics analysis of the captured dump using the created models (data structure definition database and statistical information). ReViver sweeps the dump for landmark signatures (inserted by the hooked API), and extracts every structure's base address and size. ReViver marks every memory address with possible data types using the memory value and ReViver's forensics rules (Section 2.4). ReViver uses its forensics results and the created models to calculates

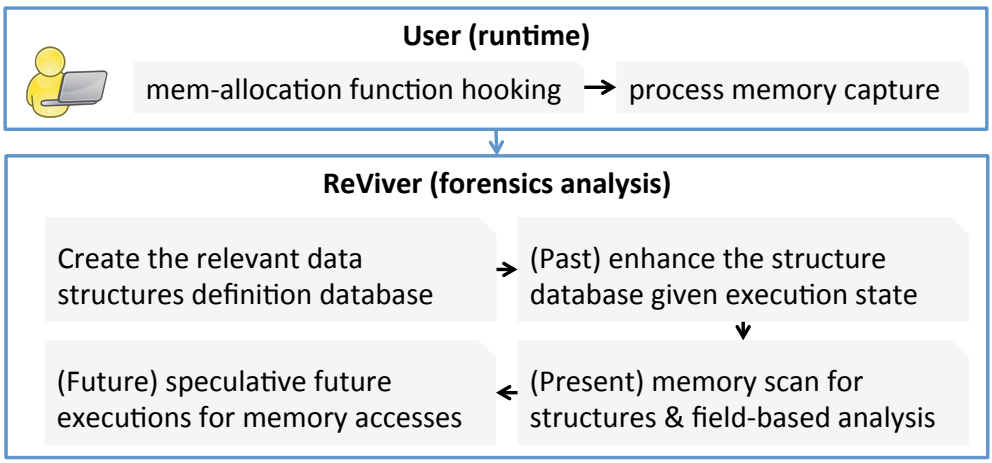


Figure 2.1: ReViver's high-level architecture

a ranked list of the best matching data structures for each memory location.

Future. ReViver revives the captured dump's execution, explores all feasible future branches of the code symbolically to generate test-cases. ReViver runs the executable with the test cases, while it reverse engineers data type revealing instructions and the library/system calls with known return/argument data types. ReViver implements backward data taint analysis on each test-case trace to backtrack the revealed data types to a memory address of the captured dump.

2.2.1 A Running Case-Study Example

We illustrate ReViver on a real case-study application, i.e., `groups` Krumins (2012). Figure 5.1 shows `groups`' partial memory snapshot and how ReViver integrates findings from past, present and future incidents to produce an accurate data structure mapping of the captured memory. Figure 5.1 shows the rare data pattern landmark signatures that the hooked API has stored in memory (four subsequent `0xEF` bytes and four subsequent `0xFE` bytes). Using the landmarks, ReViver recognizes four structures: at `0x09406CA8` with a size of 12 bytes, at `0x09407570` with a size of 8 bytes, at `0x09407588` with a size of 36 bytes; and at `0x094075C0` with the size of 352 bytes. The identified data structures include many zero bytes as their fields whose types are unrecognizable. Consequently, an accurate memory forensics based on

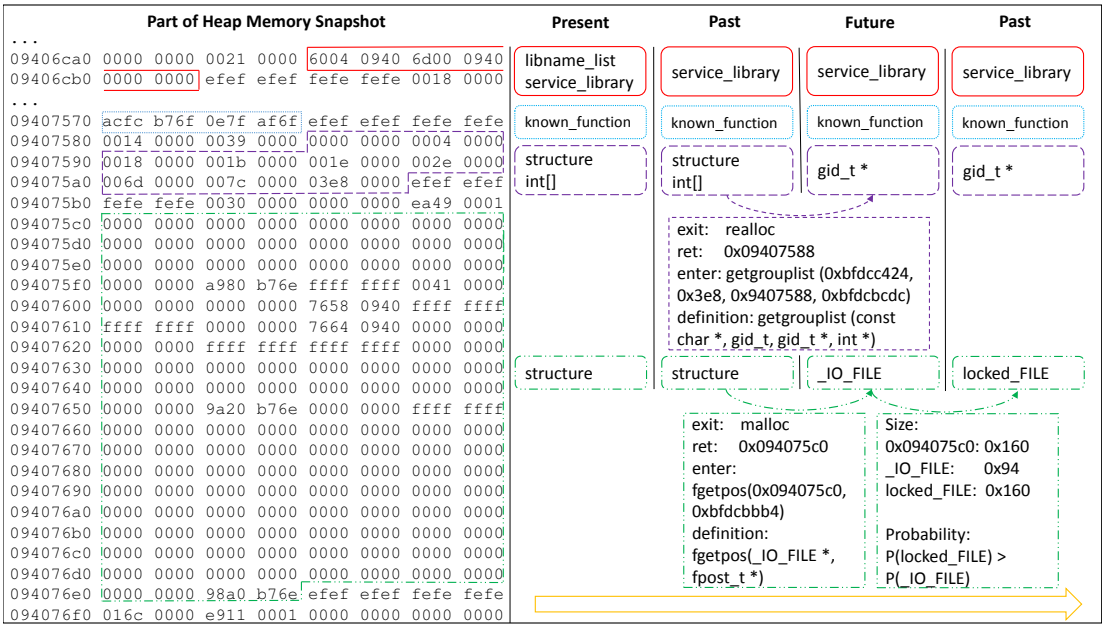


Figure 2.2: Memory snapshot of the `groups` application and the reverse engineered data structures

only the current memory snapshot (present) turns out to be often infeasible in practice. ReViver leverages several information sources to minimize the above-mentioned uncertainties. ReViver starts with collecting statistical information about the potential set of data structures resident within the captured dump. ReViver symbolically executes the executable from its entry point to the captured dump's execution state, and generates test-cases. Concrete execution of each test-case results in a separate memdump. ReViver traces all data structure allocations during individual test-case paths and maintains a list of structures that survive (i.e., not overwritten) until the captured dump state. ReViver uses the lists to create a probability distribution of the structures (Section 2.3). Figure 2.3a shows the frequency of data structures within the test-case memories. There have been totally 17 data structures allocated during at least one of the test-case executions and not overwritten before the memory capture point. For these data structures, ReViver obtains their definitions from its structure definition database.

During the present phase, ReViver inspects individual non-zero fields within the identified structures (Figure 5.1), and creates a superset of possible data types for each field. The zero fields could be of any possible data type. ReViver then goes

through the data structure definition database, and finds the potentially matching data structures whose individual field types match one of ReViver's inferred types for the corresponding fields on the captured dump. The leftmost column (present) on Figure 5.1 shows the results for the four structures. For instance, ReViver has identified the first memory-resident data structure (`0x09406CA8`) to be of either `libname_list` or `service_library` type. In practice, although there are many zero regions, the existing non-zero values are still very helpful to narrow down the possible candidates.

Figure 2.3c shows the data types for individual fields of the two data structures from ReViver's database. The first field type is the same for both the data structures (`char*`) as identified by ReViver's investigation of the memory (pointer-str). The second field for `service_library` is `void*`, i.e., could take almost any value, and would not help ReViver much to distinguish between the structures. ReViver identifies the second field of the data structure of the captured memory as a pointer that match the second field of both `libname_list` and `service_library`. However, the third field of the two structures are of different types (`int` vs. `service_library*`) and could potentially be helpful to determine the matching candidate. Unfortunately, the corresponding field in the memory is zero and ReViver's present static dump reverse engineering could not conclude whether the zero corresponds to a structure pointer or an integer.

ReViver's next step is to enhance the possible structures set using the collected prior statistical information (Figure 2.3a). Going through the shortlisted 17 structures, ReViver calculates the prior distribution as $P($`service_library`$) = 0.015/(0.015 + 0.0) = 1$ and $P($`libname_list`$) = 0.0/(0.015 + 0.0) = 0$, where 0.015 directly comes from Figure 2.3a for `service_library`; 0.0 denotes the probability of `libname_list` that did not appear in any of the test-case memdumps. In `groups`' case, ReViver covered all possible past execution traces during the symbolic execution; therefore, ReViver could confidently exclude `libname_list` from possible data structures in the captured memory. However, ReViver cannot often cover all feasible paths in larger applications during its symbolic execution because of the state explosion problems, and hence its estimated empirical probability distribution of the data structures may not

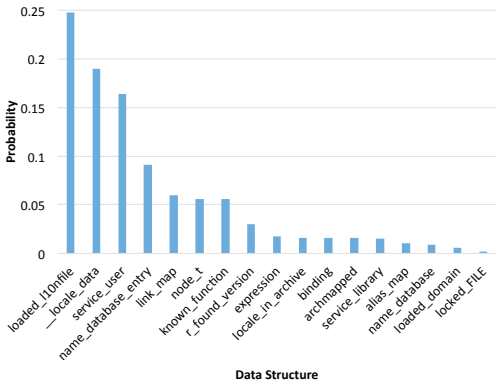
be absolutely accurate. In such cases, ReViver replaces 0.0 with a predefined small number ϵ for the structures that did not appear in any of the test-case memdumps but they were present in at least one of control flow paths during static graph-theoretic traversal of the target application's call graphs.

ReViver's present and past investigations on `0x09406CA8` suffice for accurate forensics outcome. However, determination of exact data structure type for `0x09407588` (like many other practical cases) is infeasible using only past and present information, and leads to ambiguity because the address could represent either a data structure or an array of integers. ReViver implements its speculative future inspection by reviving the suspended execution for a continued symbolic execution where ReViver intercepts the type revealing instructions. Figure 2.3b shows the partial assembly code snippet that ReViver symbolically executes. The call to `realloc_groupbuf` (line 13) returns `0x09407588` (our target data structure; see Figure 5.1) and stores it in register `eax`. This value is later moved to `ebx` and used as the third argument to the `getgrouplist` function call (line 28) by getting pushed in stack on line 24.

ReViver determines the argument type information for the function `getgrouplist` using its database[1]. The third argument is a `gid_t*` pointer. Once the `getgrouplist` function call is logged, ReViver performs backward taint analysis and marks memory addresses with relevant data type information, i.e., `0x09407588` is marked as `gid_t*`. ReViver's taint analysis keeps track of overwritten memory during the test-case executions. For instance, ReViver would not have reported the captured memory `0x09407588` as `gid_t*` had `0x09407588` been written to between the memory capture point and the `getgrouplist` call.

Reverse engineering of the memory `0x094075C0` is even more motivating regarding what ReViver's hybrid approach can achieve. Initial past and present analyses give no information about the allocated memory mainly because of its many zero bytes. During the future speculative execution, ReViver intercepts a function `fgetpos` call that uses `0x094075C0` indirectly (identified by ReViver's taint analysis) as its argument.

[1] ReViver's library function definition database includes both exported API and internal non-exported library function definitions.



(a) Statistical Information for `groups`

```
0804c220 <mgetgroups>:
...
9  804c233:xor    %eax,%eax
10 804c235:mov    $0xa,%edx
11 804c23a:movl   $0xa,0x2c(%esp)
12 804c241:
13 804c242:call   804c1e0 <realloc_groupbuf>
...
23 804c271:mov    %ebp,0xc(%esp)
24 804c275:mov    %ebx,0x8(%esp)
25 804c279:mov    %edx,0x1c(%esp)
26 804c27d:mov    %eax,0x4(%esp)
27 804c281:mov    %edi,(%esp)
28 804c284:call   8048db0 <getgrouplist@plt>
...
```

(b) Partial `groups` Code Snippet (AT&T Syntax)

Struct Name (size)	`libname_list` (12 bytes)	`service_library` (12 bytes)	`0x9406CA8`
First field	`const char*`	`const char*`	Pointer-str
Second field	`struct libname_list*`	`void*`	Pointer-stru
Third field	`int`	`struct service_library*`	Zero

(c) Two Data Structure Candidates for Memory `0x9406CA8`

Figure 2.3: Memory reverse engineering through statistical data structure information

ReViver infers the data type as `_IO_FILE` according to its library function definition database; however, ReViver finds the allocated memory size (352 bytes) is larger than `_IO_FILE`. ReViver's investigation through statistical information (Figure 2.3a) does not comply with ReViver's speculative analysis outcome, because `_IO_FILE` does not appear as any of the 17 data structures on Figure 2.3a. Finally, ReViver's nested data structure inspection finds out the 352-byte `locked_FILE` from the statistical information contains `_IO_FILE` as one of its fields. Consequently, ReViver marks `0x094075C0` as `locked_FILE`.

For accurate outcomes, ReViver's field-level type reverse engineering module had to deal with pointer mangling occasionally that is a security feature which aims to increase the difficulty of maliciously manipulating function pointers in structures. For instance, without such consideration, ReViver would not determine the memory `0x09407570` to be of type `known_function`, because the second field of `known_function` definition is a pointer. However, its second field value on the captured memory cannot be a pointer as it does not point to the acceptable memory address range. The reason for such a mismatch is that the second field value is mangled by the program before getting saved in memory. ReViver addresses pointer encryption through its initial decryption pass over the memory (Section 2.4).

2.3 Past: Statistical Information

ReViver creates statistical information about the executable's use of data structures, and leverages that information as prior knowledge about the structures ReViver may be facing in the captured memory dump. This enables ReViver to compute and report a probabilistic ranking of the structure candidates for every memory region. Please note that ReViver's objective is not to determine the *exact* execution path that the process has gone through before the memory capture point (like Zamfir and Candea (2010)). Instead, ReViver investigates all the past possible execution paths and creates a probabilistic knowledge base for its later analyses.

2.3.1 Best-Effort Partial Symbolic Execution

ReViver profiles structure allocation of the executable by instrumented execution. To maximize the use of the analysis time, ReViver implements a symbolic execution of the binary to enumerate unique feasible execution paths, and generates concrete test-cases for each path.

Typical binary symbolic execution goes through each path all the way from the binary's entry point to one of its exit points, e.g., main function return. However, such a complete full-path investigation would be unnecessary for what ReViver tries to achieve; it could even reduce ReViver's forensics accuracy, because it may traverse the code segments that the application's captured memdump had not gotten to before its execution suspension. Instead, ReViver only considers partial paths that start from its entry point and ends at the `eip` register value and the call stack of the captured memdump. ReViver's *partial* symbolic execution reduces the number and complexity of typical path condition satisfiability checks for the symbolic execution. This improves the analysis time and ReViver's ultimate accuracy significantly.

In addition to the partial-path symbolic execution, ReViver takes another step to further optimize its analysis when collecting the executable's data structure allocation profile. A typical testing-oriented symbolic execution's objective is to maximize its code coverage and exhaust all the paths; however, that is not ReViver's ideal goal.

ReViver directs the symbolic execution towards only the execution paths that include memory allocation function calls, and does not waste time on the remaining paths that are irrelevant to ReViver's forensics analyses. To that end, ReViver deploys LLVM passes to obtain the call and control flow graphs for the application executable and its libraries.

ReViver statically analyzes the graphs and enumerate possible paths that *i)* start from the application's entry point and ends at the captured memdump's execution state; and *ii)* include memory allocation function calls. Note that some of these paths may be infeasible because of the static analysis. ReViver then implements directed symbolic execution to further prune down the path set to include the feasible ones only, and generates the corresponding test cases. The two above-mentioned optimizations reduced ReViver's search space and analysis time requirement by approximately an order of magnitude in our experiments (Section 2.6.4). It is important to note that ReViver's correct functionality does not require symbolic execution of *all* the refined paths, as ReViver uses the outcomes as statistical information to help with its later forensics analysis routines.

We developed ReViver's partial symbolic execution as a module on top of KLEE Cadar et al. (2008a). ReViver disassembles the given application's binary executable and extracts the corresponding LLVM intermediate level code that KLEE can execute symbolically. ReViver's static analysis modules enumerate the relevant execution paths, and produce a list of binary vectors in a file. Each binary vector represents an execution path, where the subsequent 0s and 1s in the binary vector indicate whether the corresponding branch instruction during the execution should be taken. ReViver feeds the generated list of vectors to the extended KLEE for partial and directed symbolic execution. We have developed customized *searcher* Cadar et al. (2008a) functions for KLEE to direct its symbolic executions. Upon every branch instruction, ReViver's searcher function reads the next bit within the currently active binary vector, and mandates whether KLEE should take the branch. Once the last bit in the vector is consumed, ReViver generates the corresponding test case and terminates the symbolic execution of that particular path. To improve ReViver's performance, our implementations do

not go through the binary vectors one by one sequentially. Instead ReViver maximizes the use of KLEE's SMT solver cache through processing the binary vectors in parallel. ReViver maintains the bit pointers on the individual binary vectors until all the corresponding test cases are generated.

2.3.2 Prior Knowledge Collection

ReViver implements an instrumented execution of the binary executable using the generated test-cases to *i)* create its data structure definition database; and *ii)* collect statistical information about the application's memory allocation for different data structure types.

ReViver extracts the data structure definitions for individual applications. ReViver leverages the past work Lin et al. (2010) that implements extended PIN tools to reverse engineer data types for the test cases. The solution requires full execution trace to complete the reverse engineering. Note that we can use the solution for ReViver's past analysis step here, because the test cases and hence their execution traces are available. However, ReViver cannot deploy the past solution to reverse engineer the captured memdump directly because ReViver does not have access to the traces before the memory capture point.

Given the extracted structure definitions, ReViver runs the generated test cases on a debugger and dynamically logs the memory allocation calls and their corresponding data types. The set of data structure types during the test-case executions are collected and rendered into a probability distribution regarding how likely it is to see a data structure of a particular type within the application's memory while its instruction pointer and call stack match those of the captured memory. ReViver assumes equal probabilities for individual execution paths unless the corresponding probability distribution is available.Note that only the data structures that survive all the way through each execution to the corresponding final memory. For instance, ReViver is not interested in a data structure that is always allocated and released during the initial phases of the execution and never makes it to the execution stage of the captured memory. To address this problem, ReViver implements backward taint analysis using Lin et al.

(2010) during the test-case executions. Consequently, ReViver's ultimate statistical information about the memdump data structures include only those that are allocated during at least one of the test-case executions and make to the final suspended memory.

2.4 Present: Static Memory Analysis

The captured memdump is the only information provided to ReViver's forensics analysis engine. The exact execution path before the capture point is not available, because to minimize the performance overhead by ReViver's local agent on the user's machine, it does not trace the instruction-level executions of running processes before a specific one is suspended and marked for analysis. However, as shown by the past research Cozzie et al. (2008), the mere identification of data structure locations (not their types) on a bare memdump without the execution trace is the initial and practically a challenging step in memory reverse engineering. For instance, running Laika Cozzie et al. (2008) on `groups`' memory (Figure 5.1) wrongly results in four data structures at addresses four data structures which base address are `0x09406004`, `0x09406D00`, `0x09407658` and `0x09407664`. To address the problem, ReViver's local agent hooks the system's memory allocation API[2] via library `LD_PRELOAD` and modifies them such that every allocated memory is followed by a fixed rare data pattern signature (8 bytes) along with the size of the allocation (4 bytes[3]). This causes a negligible overhead (1.8% on average), and allows ReViver to identify the data structure locations and their sizes on the memory accurately later with a single linear sweep over the memory. Note that in case any of the application's memory content ever matches the landmark's rare data pattern, this could at most result in a single misrecognition only if the following number on the memdump matches one of the data structure sizes in ReViver's structure definition database. We did not experience a single match in our experiments. ReViver distinguishes various memory allocation call sites within the application executable and

[2]Specifically, `malloc`, `calloc`, and `realloc`. ReViver ignores specific call-sites whose allocated regions do not survive the initial execution stages, e.g., the `dl-minimal` allocator.

[3]The maximum size of an allocated data structure could be `MMAP_THRESHOLD` that is set to 128 KB by default.

its libraries recursively. Therefore, to improve the overall reverse engineering accuracy, ReViver stores different rare data landmark signatures for different memory allocation API functions.

2.4.1 Trace-Free Reverse Engineering

To reverse engineer the captured memory, ReViver first locates the allocated memory regions via a quick pass through the captured dump looking for its signature landmarks. ReViver's next objective is to identify the type of the allocated memory regions. Given only the memdump, it is often infeasible to determine the exact type of every allocated region because of limited available information, e.g., lack of access to the binary's past execution trace before the capture point. For ultimate accuracy, ReViver comes up with a superset of possible types for each memory region, and excludes irrelevant types during the later analyses to determine the most likely single type for individual regions. ReViver marks each allocated memory as one of the following four types: *i)* a string of characters; *ii)* a pointer-to-pointer (e.g., `struct link_map **p`) represented as an array of pointers on the memory; *iii)* a data structure including various field variables of primitive data types (e.g., an integer); or *iv)* a str-zero for allocated regions filled only with zeros. The str-zero regions could in fact represent one of the former types that cannot be recognized because of insufficient information from the captured memory (due to the zero memory bytes).

ReViver labels every allocated memory region with a four-entry confidence vector. Each vector entry indicates the confidence level that the allocated region is of one of the above-mentioned types. To that end, ReViver employs the forensics rules $r \in R$ for each type that we have developed based on our experience and investigation of application executable and library data structures. Each rule has a confidence weight $\omega \in \{\text{low (L)}, \text{medium (M)}, \text{high (H)}\}$, where $L/M/H$ are mapped to numerical values 0.25/0.5/0.75. ReViver calculates the confidence vectors for each allocated memory region using the average of the forensics rules' that hold true for that memory region:

$$C_t(m) = \frac{\sum_{r \in R_t} \omega_r \cdot I_{m \text{ satisfies } r}}{\sum_{r \in R_t} \omega_r}, \tag{2.1}$$

where $C_t(m)$ is the confidence that the memory address m is of type t. I is the indicator function, and R_t represents the forensics rules for the data type t. In the following, we review the major forensics rules only (for space limitations). The rules have high weights except explicitly mentioned otherwise.

Strings. ReViver analyzes the captured memory mostly at the machine word granularity except for character strings. ReViver traverses the memory byte by byte looking for strings of ASCII characters (`0x00-0x7F`) that include both control (`0x00-0x1F`) and printable (`0x20-0x7F`) characters. Most strings in real-world applications contain printable characters only, hence ReViver marks a memory region as control character strings with forensics confidence of L. The first byte of a string could be either its initial character (`0x01-0x7F`) or zero (`0x00`) that indicates a null string. The remaining bytes follow similarly. If the first byte is printable, the memory stores a printable string and hence the remaining bytes are also between `0x20-0x7F` (forensics confidence: M). ReViver assumes big endian byte ordering for allocated strings, and little endian for pointers and other data (e.g., int, float, long and enum).

Pointer-to-pointers. ReViver extracts the address space layout of the captured memdump, and looks for the pointer-typed memory addresses based on whether they point to one of the acceptable address ranges such as the heap and the memory-mapped segment for dynamic libraries and mapped files, e.g., frequently used `local-archive`. If ReViver detects an allocated memory of five or more subsequent pointers, that memory most likely stores a pointer-to-pointer data type; fewer number of subsequent pointers are likely the initial fields of an allocated data structure (forensics confidence: M). If the allocated memory is a sequence of four-byte zeros and pointer (memory address) values, ReViver marks the memory as pointer-to-pointer and data structure with forensics confidences of M and H respectively. ReViver will further refine the results to more accurately distinguish pointer-to-pointer and data structure types. In particular, ReViver leverages the fact that pointer-to-pointer regions (e.g., `struct link_map **p`) contain pointers with identical types. Once ReViver determines the type of individual pointers in its later analysis stages, the pointer-to-pointer label will be removed from the allocated regions that contain different pointer types.

Data structures. Data structures store a set of one or more fields with possibly different data types (such as int, long, long long, enum, char, char[], floating point, void *ptr, nested data struct, union). ReViver marks an allocated memory as a data structure if it cannot be recognized as a string of characters, str-zero, or pointer-to-pointer. Note that the allocated data types such as integer arrays are also labeled as data structures at this analysis phase. ReViver will remove their label later if it does not match any data structure definition in ReViver's database. Once a memory region is marked as data structure, ReViver starts reverse engineering its individual field's data types. ReViver labels the data fields as the following types: *pointer-stru*: a pointer that points to another allocated memory region. ReViver knows about all of the allocated memory addresses based on the memory landmarks, and checks whether a given data structure field points to an allocated region; *pointer-str*: a pointer field that points to a character string. ReViver checks this for a given field value given its former string forensics results (discussed above); *pointer-usr*: a pointer field that points to a non-heap memory segment. ReViver marks a field as pointer-usr using the information it obtains from the captured dump regarding the various memory segments; *data*: primitive data types such as int, floating point, and long. ReViver marks the fields, with no pointer labels already, that have byte values between `0x7F-0xFF`, or between `0x01-0x7F` without and ending null character, as data. If the byte values are between `0x00-0x20`, it may be data or control character strings. ReViver marks them as data with forensics confidence of M; and *array-char*: a character array. ReViver marks a field as an array-char using its string forensics rules.

Str-zeros. ReViver marks a memory region that is filled with only subsequent zeros for the allocated memory space as the str-zero type (forensics confidence: H).

During the field-level forensics, ReViver's initial implementation identified some of the pointer types as data due to the pointer protection (`PTR_MANGLE`) to XOR the pointer value with a random number that is generated during the application's launch time (see the listing below). We extended ReViver to extract the random number from the captured memdump and implement `PTR_DEMANGLE` before reverse engineering the

memory.

```
typedef struct known_function{
    const char *fct_name;
    void *fct_ptr;
}

known_function *k = malloc(sizeof*k);
k->fct_ptr = result;
PTR_MANGLE(k->fct_ptr);
```

It is noteworthy that because ReViver bases its investigation mostly on four-byte memory words, it misses shorter sized types, such as two-byte integers (short), char or Boolean variables. Every variables of those types takes up four bytes in memory because of the compiler-enforced alignments, and ReViver marks them as integers initially. To minimize the affect on the ultimate result accuracy, ReViver considers the integer type inclusive of a single or multiple adjacent (up to four bytes) instance(s) of the above-mentioned types during its later analyses (Section 2.4.2).

There are often many zero bytes in a memory-resident data structure that makes the type reverse engineering of the fields much more challenging. To address the zero-byte problem, ReViver leverages an observation that often occurs in practice. On a suspended memory dump, there often exist more than one instance of a data structure where a particular field in some of those instances is filled with zeros while the same field on other instances store a non-zero value whose type ReViver can reverse engineer. ReViver picks any pair of allocated equally-sized data structures on the memory and verifies whether the instances could represent the same data structure, i.e., whether there is a non-zero field in both instances with different reverse engineered data types. If the instances could represent the same structure, ReViver labels the zero fields in each instance with the revered engineered type information of the same field on the other instance if it is non-zero. ReViver marks those type information with the forensics confidence of M because the above-mentioned verification outcome could possibly be incorrect, e.g., two different data structures having identical non-zero field data types.

2.4.2 Pruning Candidate Data Structures

Given the field-level type information for allocated data structures, ReViver will investigate its data structure definition database to find the best matching candidates. However, this would result in many false positives because of the large number of data structures, e.g., 815 structures per application on average in our experiments (Section 4.6). ReViver will prune the set of candidate data structures before its data structure match-making analysis (Section 2.4.3).

In practice, most of the application's memory-resident data structures are defined and allocated by the application binary executable or by the libraries' exported and other internal functions. ReViver performs a static (offline) recursive traversal of the call graphs of the executable and its loaded libraries, and looks for memory allocation call sites. ReViver marks each corresponding data structure type as a relevant candidate in the captured memory. Note that ReViver considers only the paths from the executable's entry point to the execution state of the memory capture point. This optimizes ReViver's search space and performance significantly. For instance, ReViver's graph-theoretic analysis prunes the `groups` application function list down by $5X$ to the list of only relevant functions that allocate memory space. Consequently, ReViver narrows down the possible data structures on `groups` captured memory from 815 down to 69 possible candidates, i.e., 92% improvement. Additionally, to improve forensics accuracy, ReViver distinguishes between different memory allocation functions, e.g., through different memory landmark signature values for each allocation API (discussed earlier in this section). Therefore, ReViver labels each candidate structure with its corresponding memory allocation function. ReViver uses this information later to avoid, for instance, matching a memory-resident `malloc`-allocated data structure with a candidate data structure that is allocated using only the `calloc` function.

2.4.3 Dump-Database Structure Matching

At this analysis stage, ReViver has the memory data structure reverse engineered type information and the narrowed down candidate data structure definitions. ReViver will

determine the best matching candidate for each memory-resident structure.

ReViver cannot base its matching analysis on the size equality between the data structure definitions and the allocated memory regions, because the allocated memory is sometimes larger than the corresponding data structure size due to reasons such as variable-length arrays with dynamically determined sizes. The C language does not support variable-length arrays, so they are often implemented using a zero-length array field where the array's content follows the former fields of the data structure on the memory (see the following snippet from glibc networking module).

```
struct binding{
    struct binding *next;
    char *dirname;
    char *codeset;
    char domain_name[0];
};
// for a domain_name with size len
struct binding *n = malloc(sizeof(struct binding) + len);
```

For each data structure on the capture memory, ReViver uses its landmark signature to determine the API function used to allocated the memory region. ReViver then goes through the pruned list of candidate data structures (Section 2.4.2) and selects the structures that *i)* are allocated with the same API function; and *ii)* have sizes smaller than the allocation memory region. ReViver uses the following equation to determine the likelihood that a memdump region represents a specific data type.

$$S(M, L) = \prod_{l \in L} \frac{C_{t(l)}(M_l)}{\sum_{t \in T} C_t(M_l)}, \tag{2.2}$$

where $S(M, L)$ represents the similarity measure between the data structure on the capture memory M and the candidate data structure L that contains various fields $l \in L$. T represents possible data types, and the function $t(l)$ returns the type of the field l within the candidate data structure L. Intuitively, Equation 2.2 calculates how closely the types of each field on the captured memory and candidate data structure match and computers the product across all of the fields. The similarity measure is a real number $0 \leqslant S \leqslant 1$. A high similarity measure indicates that the memory-resident data structure M matches with the candidate structure L very closely and it is very distinct

from other candidate data structures. Consequently, ReViver creates a ranked list of candidate data structures for each memory-resident structure based on the calculate similarity measures. The ranked quantified list of data structures for the memdump enables flexible use cases for ReViver in practical real-world forensics scenarios. As its ultimate fully-automated outcome, ReViver can pick the most likely candidate for each memdump region and deliver the whole memory with labeled semantic information about the data structures and their field data types. Alternatively, ReViver can be used for human-assisted forensics analysis, where the security admin is provided with the ranked structure lists and ReViver's automated justification for why each memory region is identified by the corresponding structure type rankings. The admin could possibly decide to take it as is, or update the final rankings and tune ReViver's internal forensics rulesets accordingly.

2.5 Future: Speculative Forensics

The past research Lin et al. (2010); Slowinska et al. (2011); Lee et al. (2011) has shown that execution traces play an invaluable role in accurate memory forensics. ReViver does not have access to the binary's past trace before the memory capture point. However, ReViver produces traces for the paths that would have been executed potentially if the execution had not been suspended. ReViver analyzes those traces to either confirm or correct its former memdump's reverse engineered data types (the previous two sections) using backward taint analysis. To produce the traces, ReViver loads the captured dump on memory and symbolically explores *all* feasible future branches of the code from the suspended point. ReViver monitors the speculative executions closely and logs important incidents, such as memory accesses, of each execution path for its later backward taint analysis and type reverse engineering.

2.5.1 Revived Speculative Execution

ReViver's first step is to suspend and store a running suspicious process on the user's machine for ReViver's forensics analysis. A mere static memory forensics analysis

(Section 2.4) would only require a simple memory dump transfer. However, ReViver's speculative symbolic execution necessitates to capture more detailed information that are needed for restoring the execution in ReViver. ReViver leverages DMTCP Ansel et al. (2009) to implement the process migration (checkpoint and restore). On the user-side, ReViver captures the process'es internal content such as its virtual memory mappings, and its external system-wide dependencies. The external dependencies include process IDs, its process tree relations, open file handles, network sockets, inter-process buffers (and the corresponding processes recursively).

ReViver looks for type-revealing instructions within the executable during the speculative execution. A simple restored concrete execution would continue a *single* path starting from the memory capture point. To maximize the analysis gain, ReViver instead loads the revived execution on a symbolic execution engine so that all feasible future paths, including the type revealing instructions, are explored. During the execution restoration, ReViver marks the process inputs as symbolic using the functions from the symbolic execution engine. These inputs include any external data that come from calls made during the forward execution. However, ReViver first has to load the necessary libraries on the target executable's address space. Our implementations inject `dlopen` to the restored execution and consequently load the shared library needed to enable symbolic execution. The loaded shared library overrides the corresponding library functions, e.g., `read()`, so any data processed by the overridden library functions are marked as a symbolic input. During the symbolic execution, ReViver generates the corresponding test-cases, and uses them for an instrumented concrete execution of the process. Note that ReViver restores the execution from the captured dump for every concrete test-case execution. ReViver instruments the test-case executions to intercept type revealing instructions and backward data taint analysis (see below).

2.5.2 Type Propagation for Type Forensics

During the speculative executions, ReViver looks for hints to help with its data type forensics result accuracy. Two great sources of data type information are the calls

to *i)* type-revealing x86 instructions such as string instructions `MOVS/B/D/W` and `STOS/B/D/W`; and *ii)* system call and library functions with their well documented return and argument types. For instance, a call to `fread(..., FILE *)` would reveal a file pointer on memory. During a one-time offline analysis, ReViver automatically investigates the libraries and the kernel to extract the library functions and system call definitions automatically (using techniques discussed in Section 2.3). Note that ReViver has already created the data structure definition database for function arguments/returns of data structure pointer types.

ReViver intercepts the type-revealing instructions, and library and system calls during the symbolic executions, and logs the revealed data types, and their argument and return values, respectively. Memory data type exposure by the instructions, and library and system calls could be either direct (e.g., when a function argument points to a user memory address) or transitive (e.g., when the argument points to the kernel memory address whose value originates from the captured user memory). ReViver extract the direct type sources by a simple post-execution log parsing. For the transitive type sources, ReViver implements backward data taint analysis to possibly back-trace the transitive data flows to the captured memdump. At the stripped binary level, type source variables exist in either memory locations or registers. At type revealing call sites, ReViver labels memory addresses or registers with their type information. ReViver create and maintain *constraint sets* Lin et al. (2010) during the executions. Each constraint set includes a set of other memory addresses that should have the same type according the previous data flow up to this point. The constraint set information refers to the memory at the current execution point that is most likely different from the captured dump because of overwritten memory regions since the revived execution. ReViver is interested only in the data structure type information in the constraint set that are present in the captured dump. ReViver keeps track of the memory writes during each symbolic execution path, and does not report the type information results about the overwritten regions as its ultimate results. It is noteworthy that ReViver still uses that information for its internal backward taint analysis for potential indirect type information about the captured memory content.

Index.Application					
1.base64	2.basename	3.cat	4.chcon	5.chgrp	6.chmod
7.chown	8.chroot	9.cksum	10.comm	11.cp	12.csplit
13.cut	14.date	15.dd	16.df	17.dir	18.dircolors
19.dirname	20.du	21.echo	22.env	23.expand	24.expr
25.factor	26.false	27.fmt	28.fold	29.ginstall	30.groups
31.head	32.hostid	33.id	34.join	35.link	36.ln
37.logname	38.ls	39.md5sum	40.mkdir	41.mkfifo	42.mknod
43.mktemp	44.mv	45.nice	46.nl	47.nohup	48.od
49.paste	50.pathchk	51.pinky	52.pr	53.printenv	54.printf
55.ptx	56.pwd	57.readlink	58.rm	59.rmdir	60.runcon
61.seq	62.sha1sum	63.sha224sum	64.sha256sum	65.sha384sum	66.sha512sum
67.shred	68.shuf	69.sleep	70.sort	71.split	72.stat
73.stty	74.sum	75.sync	76.tac	77.tail	78.tee
79.test	80.touch	81.tr	82.true	83.tsort	84.tty
85.uname	86.unexpand	87.uniq	88.unlink	89.uptime	90.users
91.vdir	92.wc	93.who	94.whoami	95.arch	96.nproc
97.numfmt	98.realpath	99.stdbuf	100.timeout	101.truncate	102.yes

Table 2.1: Applications' Name-Index Mappings

2.6 Evaluations

We evaluated ReViver on CoreUtils v8.22, and 5 large and medium-size popular desktop and server applications (Table 2.1 shows the mappings between the applications and their indices that we used for presentation clarity). We suspended each process at a random execution point, i.e., half way through its finish time. The user desktop ran Linux kernel v3.11. We implemented a plugin for KLEE Cadar et al. (2008a) for mid-point and selective symbolic execution. We developed and employed an application in Intel PIN Luk et al. (2005) for backward taint analysis using techniques introduced in Lin et al. (2010). We designed a set of experiments to verify whether ReViver can be useful in real-world practical scenarios by answering the following questions empirically: How accurately does ReViver reverse engineer the memory data types without the access to execution traces? How efficiently does ReViver complete the memdump reverse engineering for real-world applications? How well does ReViver scale up for data structures from complex widely-used applications?

2.6.1 Case-Study: The Groups Application

We ran `groups` without a commandline option. ReViver initially shrunk the number of relevant functions from 60 to 12, and the data structures from 815 to 69 relevant candidates (Section 2.4.2). ReViver was able to match and recognize several instances of same structures correctly despite existing zero bytes (Section 2.4.1). The following

shows three partial snapshots of `groups` memory along with ReViver reverse engineered data types. All the memory portions represent `node_t` structure instances (determined using `gdb`). Although ReViver cannot confirm this confidently because of several zero fields, ReViver keeps `node_t` as one of the options for these allocated regions. ReViver later marks `node_t` as the most likely candidate for these memory regions using its collected statistical information (Section 2.3).

```
address      content       reversed type/size
-------------------------------------------
0x09405e58   09 40 5e 78   Pointer-stru    4
0x09405e5c   00 00 00 00   Zero           4
0x09405e60   00 00 00 00   Zero           4
0x09405e64   00 00 00 01   Data           4
...
0x09406018   09 40 60 38   Pointer-stru    4
0x0940601c   00 00 00 00   Zero           4
0x09406020   00 00 00 00   Zero           4
0x09406024   00 00 00 00   Zero           4
...
0x094074a8   09 40 74 c8   Pointer-stru    4
0x094074ac   09 40 75 18   Pointer-stru    4
0x094074b0   09 40 60 18   Pointer-stru    4
0x094074b4   00 00 00 00   Zero           4

data structure definition (16 bytes)
------------------------------------
struct node_t{
    const void*     4
    struct node_t*  4
    struct node_t*  4
    unsigned int    4
};
```

2.6.2 Accuracy

We measured the accuracy of ReViver's ultimate data type forensics. Figure 2.4 shows the results for strings and other data structures of each application separately. ReViver correctly recognized 99.2% of the strings and 96.7% of the memdump data structures correctly. ReViver's overall accuracy level 98.1% is very promising even though ReViver did not have access to the execution traces before the memory capture point.

We investigated the relatively low accuracy for four applications (#12.`csplit`, #30.`groups`, #55.`ptx`, #81.`tr`). ReViver recognized some of the data structures as

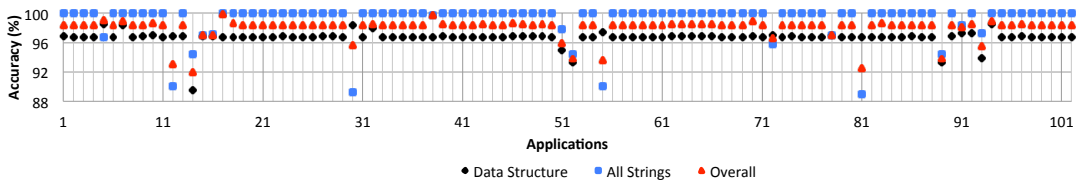
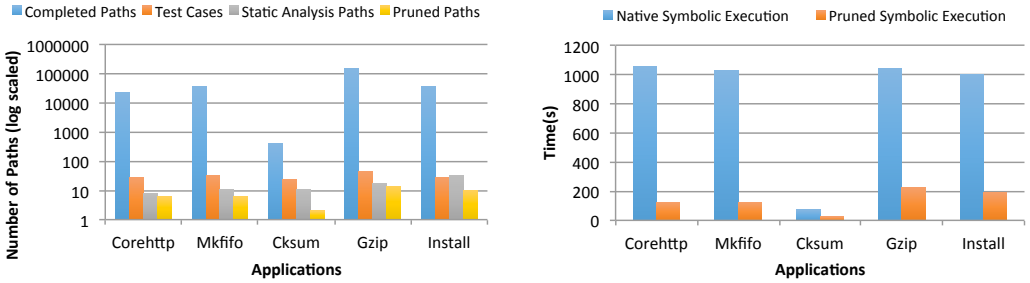


Figure 2.4: Memory data type reverse engineering accuracy



(a) Search Space Reduction

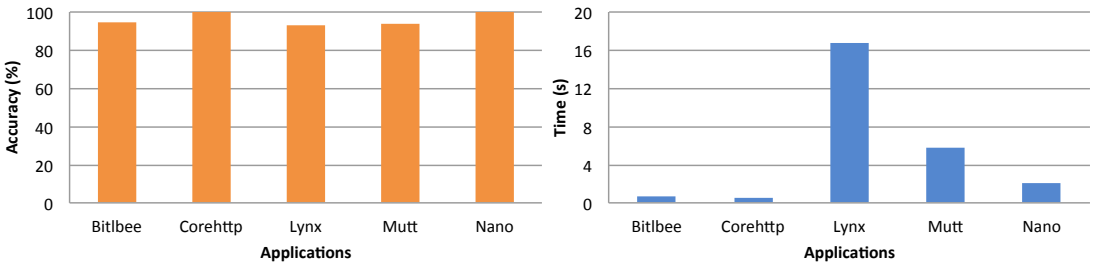
(b) Time Requirement Reduction

Figure 2.5: Speedup via directed symbolic execution

(more likely) strings due to lack of any helpful information (all zero bytes). Regarding the accuracy of #14.`date`, there were three instances of 3-field `tzstring_l` structures. The last fields of those instances on the memdump were all zero. Consequently, ReViver marked those regions as `key_call_private` that also has three fields, and its first two field types are identical to `tzstring_l`.

We also evaluated ReViver's speculative execution (Section 2.5) that ReViver uses to improve the forensics accuracy. Table 2.2 shows some of the cases in our experiments that the speculative execution helped ReViver to either confirm its former forensics results about a particular memory region or correct the region's data type (the last column). The table shows the library/syscall function names and instructions that revealed a data type along with the argument and data type details. The data type information about a specific memory address by ReViver's speculative execution replaces its former analysis results in case of a mismatch. Because, unlike ReViver's former analyses, the results from speculative execution come from instruction syntax and library/syscall function definitions and are *always* correct.

ReViver shrinks its analysis search space through static executable analysis (Section 2.4.2). On CoreUtils, each application may call 67 functions on average. ReViver



(a) Forensics Accuracy (b) ReViver Time Requirements

Figure 2.6: Forensics accuracy and time requirement

shrunk this number to 26 functions per application based on the captured memory's execution state. Consequently, ReViver excluded 63% of functions for each application from its later analyses. ReViver parses through the remaining 37% of the functions recursively, and determine the data structures that may have been allocated before the memory capture point. ReViver filters out 92% of the total data structures as irrelevant, and uses only the remaining 8% for its forensics analyses. This is a significant search space reduction in order to improve ReViver's overall performance.

2.6.3 Performance

From the usability viewpoint, upon the user's request to analyze a particular local process, ReViver needs to efficiently capture the memory dump locally and revive the process for ReViver's forensics analysis (Section 2.5.1). ReViver requires 24.9 ms on average to suspend each application and 26.6 ms to restore its execution. We measured how much data needs to be transferred to ReViver upon every user request. ReViver transfer 2.62MB per application on average. The captured data includes the process'es internal memory and external dependencies.

ReViver does not require the prior execution trace due to its high overhead, and instead overrides memory allocation API to store landmark signatures in memory to maximize its information-gain/overhead (Section 2.4). We measured the time it takes for each application to finish a predefined fixed workload before and after ReViver's instrumentation. ReViver's average runtime overhead is negligible (1.8% on average over 1000 runs), and reasonably low for practical deployments. We measured the time

it takes for ReViver to complete the major steps of its memory forensics. ReViver's static memory analysis (Section 2.4.1) takes 0.53 seconds on average. ReViver then takes 523 seconds to extract the data structure definitions and match each one with every allocation memory structure.

2.6.4 Fine-Grained Measurements

For better accuracy, ReViver puts different landmarks for different memory allocation API (Section 2.4). We measured the call frequency of the allocation API across different applications. The frequencies of `realloc` (0.26 calls/application on average) and `calloc` (0.59) are significantly lower than `malloc` (21.7). ReViver was able to correctly recognize all the structures that were allocated using `realloc` and `calloc` functions.

ReViver use statistical information to improve its forensics accuracy. We measured how informative (helpful) the collected prior knowledge (Section 2.3.2) is for memory forensics analyses. We used information theoretic entropy measure $H(D) = -\sum_{d \in D} P(d) \cdot \log P(d)$, where $P(d)$ is the empirical probability of the data structure $d \in D$ by ReViver. Based on our experiments, the entropy level of the statistical information varies across different applications. Higher entropy indicates higher similarity between the calculated data structure frequency distribution and the uniform distribution, and hence conveys less information. Lower entropy measures allow ReViver to better rank structures during its later forensics analysis, and hence improve its accuracy.

ReViver generates its statistical information through its best-effort partial symbolic execution given a fixed 10-minute deadline for each application. Due to the state space explosion problem, ReViver's symbolic execution often cannot fully exhaust the executable. However, this does not affect ReViver's correctness, because ReViver needs just an empirical estimate of the statistical information for its later forensics analysis. Incomplete code coverage will still provide ReViver with a decent estimate of the probability distribution over the allocated data structures.

General symbolic execution techniques suffer from state space explosion problem.

To address this concern, ReViver implements a partial best-effort (directed) symbolic execution of only relevant paths, where data structures get allocated or used through type-revealing instructions and function calls. Figure 2.5a shows how ReViver's static path pruning shrinks its symbolic execution search space significantly (please note the logarithmic vertical axis scale). The results are shown for five applications including a web server `corehttp`. The figure reports four numbers for each application: *i)* the total number of static execution paths; *ii)* the number of feasible paths for native symbolic execution by KLEE; *iii)* ReViver's calculated pruned number of static paths that traverse from the application's entry point to the memory dump's suspended execution state and include data structure allocation or type-revealing instructions; and *iv)* the feasible subset of pruned paths that ReViver's directed symbolic execution goes through for test-case generation purposes. ReViver's directed symbolic execution results in approximately $5X$ search space size reduction compared to the native KLEE (Figure 2.5a). The native KLEE-based symbolic execution goes through many irrelevant useless execution paths that do not contain any relevant function calls. The demonstrated remarkable search space refinement results in a significant performance speedup that reduces the time ReViver takes to complete its past analysis step. Figure 2.5b shows the results for the same set of applications. ReViver's offline executable analysis and directed symbolic execution reduces the overall analysis time requirement by approximately an order of magnitude ($10X$) on average. Despite ReViver's reduced-time symbolic analysis, it is noteworthy that ReViver's correct functionality does not definitely require complete symbolic execution (100% exhaustion of relevant paths), because ReViver treats the past analysis step outcomes as statistical information that may include inaccurate entries. Rather than an exhaustive directed symbolic execution, ReViver's partial best effort analysis searches through the relevant paths until a predefined deadline is met.

Finally, we implemented ReViver to reverse engineer data structures on memory dumps of five popular applications (Figure 4.15): a web browser (`Lynx`), an email client (`Mutt`), an instant messaging application (`Bitlbee`), a web server application (`Corehttp`), and a text editor (`Nano`). Figure 2.6a shows the ultimate data structure

App	Structure	Lib-func/Syscall	Resolved Fields	Result
stat	char*	memcpy()	char *dest	Conf.
	passwd*	getpwuid()	struct passwd *pw_ent	Corr.
	group*	getgrgid()	struct group *gw_ent	Corr.
who	utmp*	strncat()	pid_t ut_pid	Corr.
		stzncpy()	char ut_line[]	Conf.
		stzncpy()	char ut_host[]	Conf.
		localtime()	int32_t tv_sec	Conf.
	char*	sprintf()	char *x_exitstr	Conf.
groups	gid_t*	getgrouplist()	gid_t *g	Corr.
	passwd*	getpwuid()	struct passed *pwd	Corr.
	group*	getgrgid()	struct group *grp	Corr.
shred	randread_ _source*	fread()	FILE *source	Corr.
		memcpy()	unsigned char b[]	Conf.
	char*	strlen()	char *qdir	Conf.
	char*	sprintf()	void *fill_pattern_mem	Conf.
csplit	char*	strcpy()	char *filename_space	Conf.
	buffer_ _record*	read()	size_t bytes_alloc	Corr.
		read()	char *buffer	Corr.
	char*	memcpy()	char *buffer	Conf.
tsort	item*	strcmp()	const char *str	Conf.
wc	argv_iterator*	getc_unlocked()	FILE *fp	Corr.
	fstatus*	fstat()	struct stat st	Corr.
chmod	FTS*	fstatat()	int fts_cwd_fd	Conf.
cp	char*	memcpy()	char *p_concat	Conf.
	char*	read()	char *buf_alloc	Corr.
ptx	char*	fopen()	char *input_file_name	Corr.
	re_dfa_t*	strncpy()	char *re_str	Corr.
		pthread_mutex_init()	pthread_mutex *lock	Corr.

Table 2.2: Results Improvement via speculative execution

reverse engineering accuracy for the above-mentioned five applications. Figure 2.6b shows how long ReViver takes to complete its analyses. Among the applications, ReViver needed more time to reverse engineer the data structures of the `Lynx` memory dump due to its larger memory footprint. Both the accuracy and time requirement for real desktop and server applications are very promising and motivating for ReViver's potential real-world deployability.

ReViver's overall average accuracy is 98.1% without the need to access the execution traces unlike other previous related work. Rewards Lin et al. (2010) achieves an average of 97% accuracy with execution traces. TIE's Lee et al. (2011) accuracy is conservative 90% on structural types. Rewards implements Intel Pin tools for dynamic execution and heavyweight execution trace logging that causes over 6X performance overhead. Howard Slowinska et al. (2011) implements all dynamic analysis techniques based on Qemu. In a target system with hundreds of processes and initially unknown bugs/vulnerabilities, runtime instrumentation and execution trace logging of individual

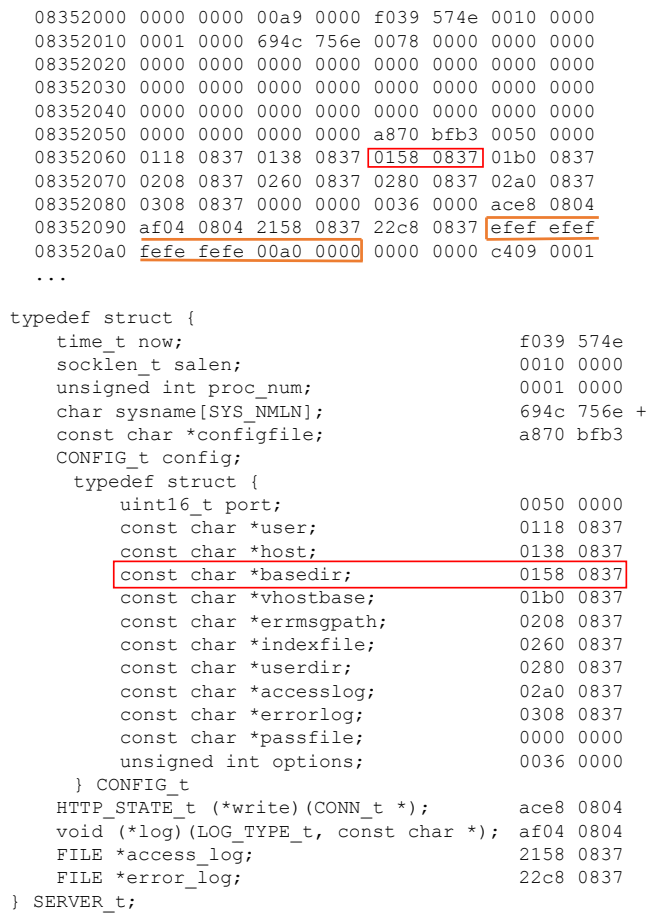


Figure 2.7: Orzhttpd memory snapshot and reversed data structure

applications (processes) within the system is impractical due to unacceptable performance overhead. ReViver does not need to record any execution traces, and hence can be used in such scenarios to protect any of the running processes within the target system.

2.6.5 Applications: Non-Control-Data Attacks

Due to practical mitigations against control flow attacks (e.g., ROP), recent attacks locate and corrupt memory variables to achieve their objectives. We demonstrate an example use-case for memory forensics to assist with detecting some types of such non-control data attacks Chen et al. (2005); Hu et al. (2015); Hu et al.. As a case in point, the semantic memory dump data structure layout provided by ReViver

can be used to determine if a sensitive global variable value has changed by the attackers. We demonstrate ReViver against data attacks on `orzhttpd` Hu et al. (2015) web server with a format string vulnerability (bugtraq ID: 41956). The attacker can exploit the vulnerable point to gain control over the program memory and modify the web root directory string on the heap. This attack changes the configuration information - root directory of the orzhttpd server to "/". The original root directory is "/home/orzhttpd/orzhttpd-read-only/www/data" specified in file "/home/orzhttpd/orzhttpd-read-only/config.xml". Upon the attack's success, the intruder can access any system file, e.g., "/etc/passwd".

Figure 2.7 shows the web server's partial memory snapshot and ReViver's forensics results of data types. ReViver identified the memory region as the `SERVER_t` data structure type and marks its individual data fields. In `SERVER_t`, there is one nested data structure `CONFIG_t` that contains the memory addresses for the memory addresses, where all the configuration information is stored. Through pointer value tracking, ReViver located the root directory in address 0x08352068, i.e., `*basedir`. `*basedir` stores 0x08370158 as the address of the target string.

Figure 3.8b shows `orzhttpd`'s post-attack partial memory snapshot, where the root directory has been modified. The red dashed box on the figure shows the string value of root directory (address 0x08370158). ReViver knows the type of `basedir`, i.e., `const char *`. The second byte is `00` (null), so when `orzhttpd` accesses the root directory strings, only the first four bytes (null-terminated) are read. ReViver was able to reverse engineer the root directory name, and a simple comparison between the extracted string and the configuration file value indicates a anomalous discrepancy. Such indicators can help the forensics analysts with detecting non-control data overflow exploitations.

2.7 Related Work

Static memory image analysis. Past research in memory image forensics analysis has mainly concentrated on reverse engineering data structure instances using signature-based brute force scanning Saltaformaggio et al. (2014a, 2015a). Generally, those techniques can be categorized into value-invariant based Betz (2015); Bugcheck (2006); Dolan-Gavitt et al. (2009); Petroni et al. (2006); Schuster (2006); Walters (2007); Walls et al. (2011) and structural-invariant based Carbone et al. (2009); Lin et al. (2012, 2011). Value-invariant signatures seek to classify data structures by the expected number and value of their fields. As a case in point, Decode Walls et al. (2011) makes use of value-based signatures with probabilistic finite state machines to recover evidence from smartphones. Structural-invariant based signatures are derived by mapping interconnected data structures. For instance, SigGraph Lin et al. (2011) employs similar signatures for target memory image scanning. DIMSUM Lin et al. (2012) attempts to probabilistically locate data structure instances in un-mappable memory. Further, numerous forensic tools and reverse engineering systems Case et al. (2008); Lee et al. (2011); Lin et al. (2010); Movall et al. (2005); Zeng et al. (2013) make use of data structure traversal. Compared with these techniques, ReViver does not stop its forensics procedures at the static memory analysis phase, and further improves its results through speculative future execution monitoring of the target process. ReViver's provided techniques may also be used to investigate the execution and the usage history of the target application Arasteh and Debbabi (2007).

Static binary analysis. Binary reverse engineering techniques Lee et al. (2011); Lin et al. (2010); Slowinska et al. (2011) can reverse engineer data types from binaries accurately. They can also reverse engineer semantic information to a certain extent. As such, they can be used in forensic analysis. These techniques, however, only can reverse engineer the executable-defined data structure definitions and their field data types through offline binary analysis techniques; they cannot directly reverse engineer data structures resident on an application memory dump when the execution trace is not available. ReViver relies on the past work to partially fill in its relevant data

```
08370110 0000 0000 0021 0000 7777 2d77 6164 6174
08370120 ef00 efef feef fefe 15fe 0000 0000 0000
08370130 0000 0000 0021 0000 726f 2e7a 7962 6873
08370140 6e65 6e2e 7465 ef00 efef feef fefe 1bfe
08370150 0000 0000 0059 0000 0100 0000 2f65 6f64
08370160 2f70 6c66 776f 7473 7469 6863 612d 7474
08370170 6361 736b 6f2f 7a72 7468 7074 2f64 726f
08370180 687a 7474 6470 722d 6165 2d64 6e6f 796c
08370190 772f 7777 642f 7461 0061 efef efef fefe
083701a0 fefe 004e 0000 0000 0000 0000 0059 0000
083701b0 682f 6d6f 2f65 6f64 2f70 6c66 776f 7473
083701c0 7469 6863 612d 7474 6361 736b 6f2f 7a72
083701d0 7468 7074 2f64 726f 687a 7474 6470 722d
```

Figure 2.8: Orzhttpd's post-attack modified root directory

structure definition database. ReViver go beyond current techniques with its present and future analyses to map the memory dump-resident structures completely through memdump forensics and speculative symbolic inspection.

Dynamic execution analysis. Rewards Lin et al. (2010) and Polishchuk et al. (2007) require execution traces and monitor the application's execution in depth. SigPath Urbina et al. (2014) requires access to the execution snapshots prior to the memory-dump's capture point. Rewards builds on a technique originally pioneered by aggregate structure identification Ramalingam et al. (1999). Whenever the program makes a call to a well-known function (like a system call), the authors label the corresponding argument and return value memory locations according to the function definition. Due to its runtime extensive execution monitoring, Rewards slows down the system's throughput over 6X. More recently, Dscrete Saltaformaggio et al. (2014a) provides memory rendering through reuse of application logic to reverse engineer a memory image for more semantic information. Dscrete heavily relies on exact identification of the so-called P function within the binary that takes as input the target data structure instance and produce the human readable application output.

2.8 Discussions and Limitations

We review some of ReViver's current limitations and how they could be addressed:

i) Data allocation functions. Can ReViver reverse engineer data structures allocated using other API? ReViver's current implementation only supports memory allocations

through `malloc`, `calloc` and `realloc`. ReViver could possibly be extended to support other memory allocators like `TCMalloc` and `jemalloc`.

ii) Deterministic replay. ReViver does not pursue a *deterministic* replay, and instead, revives and analyzes the continued execution in ReViver even though it may be slightly different from its potential trace if the process continued executing on the user's local machine.

iii) Memory modification attacks. What if the attacker modifies ReViver's landmarks on the memory, e.g., through buffer overflows? To prevent against such attacks one could update ReViver's implementation to make the preloaded memory allocation function store the landmark information in a randomly selected memory region (like Kuznetsov et al. (2014)) or to an external read-only space such as another process memory or a log file.

iv) Data-based type forensics uncertainty. Unlike some recent work Lee et al. (2011); Slowinska et al. (2011), ReViver does not have access to the *past* execution trace and type-revealing instructions, and hence has to perform forensics mostly based on the memory data values. One challenge in practice is the allocated memory regions with unpopulated fields filled with zero values. This reduces the analysis accuracy. To that end, we developed two solutions: ReViver's prior knowledge collection module narrows down the search space; and its future speculative forensics makes use of type-revealing future instructions to infer memory data types.

v) Symbolic analysis limitations. ReViver's future speculative analysis extends and is based on the existing symbolic execution technologies. Hence, ReViver's practical deployment inherits their limitations for complex path conditions, such as finding a string m for which the one-way hash value is `SHA-2(m)=0xdeca3ed7f8eb4d`. Additionally, the current symbolic execution solutions do not perform efficiently for applications with intensive graphical user interfaces. Consequently, ReViver can extract data types for those applications by past and present analyses only.

vi) Obfuscated application executables. ReViver's prior knowledge collection module develops executable static analysis techniques to extract control flow graphs and search for memory allocation functions. ReViver's current implementations assumes the

executable does not include anti-disassembly and is not obfuscated. More advanced solutions Kruegel et al. (2004) may be used to handle obfuscated binaries. We consider this as a potential future work.

2.9 Conclusions

We presented ReViver, a hybrid data structure reverse engineering solution that takes the memory image for a selected running process on the user's machine, and determines its semantic data structure layout without the need for execution traces before the memory capture point. ReViver performs a static forensics analysis of the captured memory dump, and its potential past and future execution traces. ReViver correctly reverse engineers 98.1% of the data structures in real-world application dumps with 1.8% runtime performance overhead.

Chapter 3

Compromising Security of Economic Dispatch in Power System Operations

3.1 Introduction

Critical national infrastructure has become increasingly complex. The power grid exemplifies a cyber-physical infrastructure, with data collected from its physical components and processed by control algorithms running on computers to provide for accurate and safe monitoring and control. Such a large-scale trusted computing base introduces a hard-to-protect attack surface. Events such as proliferation of the Stuxnet worm Falliere et al. (2010), the coordinated attack on the Ukranian power grid Assante (2016), and the emergence of new threats that leverage existing weaknesses in these systems Rashid (2013) demonstrate that cyber-physical infrastructures are unprepared to maintain their safe and secure operation in the face of malicious adversaries.

Despite the failures, the past intrusions had two features: *i)* they mostly required full ownership of the target controllers (e.g., Siemens Step7 server compromise by Stuxnet Falliere et al. (2010)) to perform the attacks; and *ii)* they did not fully optimize their adversarial impact via utilization of the underlying physical model. A semantics-based attack can do a lot more using much less resources. For instance, an attacker with access to only few power system parameters can leverage its dynamical model to calculate the malicious replacing parameter values such that the ultimate damage to the power system is maximized.

In the literature, there has been an extensive body of work on false data injection attacks Liu et al. (2011), where the compromised sensors send corrupted measurements to mislead the operators regarding the power system state. Such attacks assume the attacker can compromise a large number of geographically and logically distributed set

of sensors remotely. In addition to the scalability barrier, remote malicious access to (analog) sensors with serial connections may not be feasible in practice. Additionally, by design, false data injection attacks target sensors or actuators only, and cannot manipulate core system parameters such as the network topology and line parameters (e.g., capacities). This information often resides within the control center servers and are used for power system operations such as state estimation and operational control. However, almost all the past real attacks (e.g., Falliere et al. (2010); Assante (2016)) against critical infrastructures have targeted control center assets (as opposed to individual sensors or actuators).

3.1.1 Our focus

This article presents a semantics-aware attack against a widely used power grid network control functionality, and demonstrates its practical feasibility on well-known *Energy Management System (EMS)* softwares. Specifically, we conduct a vulnerability assessment of an important functionality provided by all EMSs – the so-called *Economic Dispatch (ED)* problem. In critical infrastructures, ED is routinely solved to set the generator output levels over a control area of a regional transmission grid. We show that software security vulnerabilities in power system controllers can be exploited by an attacker (an external hacker or a strategic market participant) to gain a backdoor entry into power grid operations.[1] By utilizing the knowledge of an approximate power flow model – specifically, DC approximation – the attacker can launch a semantic memory attack to change the critical parameters such as transmission line ratings (capacities). A transmission line's rating reflects the maximum amount of power that it can carry without violating safety codes or damaging the line. We design experiments using ED implementation on real-world EMS software packages to demonstrate the economic and safety risks posed by use of manipulated line ratings.

The core of our attack generation approach against the power grid infrastructure is a bilevel optimization problem that encodes the attacker's partial knowledge of power

[1]Throughout the chapter, we use the term *controller* as the ED implementation software packages that solve economic dispatch problem.

system operations to compute the target malicious power system parameters. This physics-aware attack generation approach enables us to identify key features of power system data and software operations whose exposure can significantly increase security risks. The implementation of our optimal attack against power system operation involves targeted manipulation of specific power system parameters that reside within the EMS's dynamic memory space. The exploit performs an online memory data search using lightweight pattern matching to locate the sensitive power system parameters used by the ED software to calculate the generation output levels. The use of manipulated parameter values makes the EMS issue incorrect dispatch (generation and power flow) commands, and consequently drive the power system towards unsafe states. The merit of our overall approach lies in the combination of the semantics-based optimal attack generation and a generic implementation procedure for EMS's memory data corruption.

The bilevel problem for attack generation can be viewed as a sequential game between the attacker (leader) and the follower (grid operator). In the first stage, the attacker chooses power system parameter manipulations with the objective of maximizing the violation of capacity limits; in the second stage, the operator solves the ED to determine generator output levels while facing the manipulated parameters chosen by the attacker in the first stage. We show that the optimal power injections and nodal voltages computed using the manipulated parameters yield suboptimal and unsafe power flow allocations. This significantly increases the possibility of cascading failures and the risk of subsequent emergency actions.

Thus, the main contributions of this chapter are as follows:

- We introduce a new domain-specific semantic data attack against power grid controllers. The attack leverages an approximate model of power system to manipulate the controller runtime memory such that the execution of the legitimate controller software, using partially corrupted values, drives the physical plant towards unsafe states.

- We formulate the problem using a game-theoretic framework to optimize the attack strategy in terms of which available data regions in the controller memory

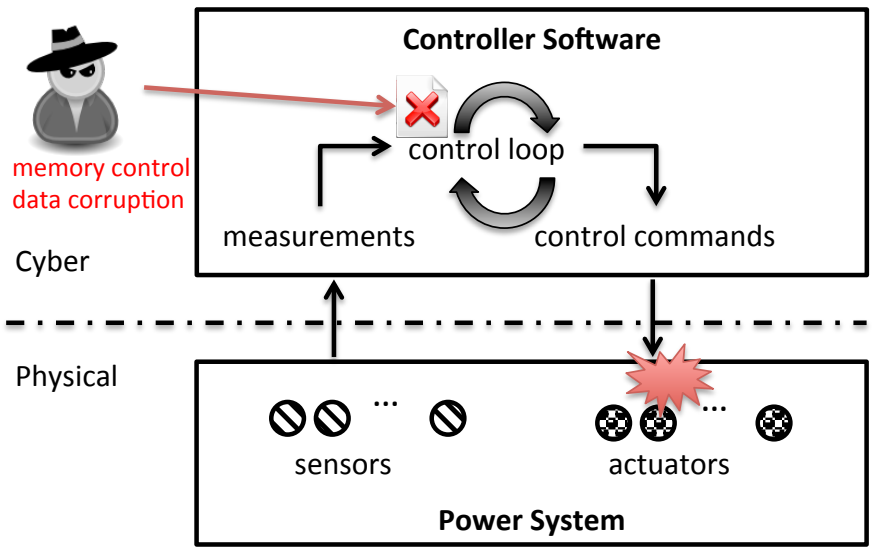


Figure 3.1: Physics-aware memory attack on control systems.

space should be modified. The adversary-optimal values are calculated using fast bilevel optimization procedures.

- We implemented working prototypes of the proposed controller attack against real-world large-scale and widely-used energy management systems. Our implementations leverage logical memory invariants to locate the sensitive power system parameters in the controller's memory space. The evaluation results prove the feasibility of domain-specific data corruption attacks to optimize for the physical damage.

In the remaining of this section, we present an overview of our proposed attack. Section 3.2 and Section 3.3 present the attack model and optimization algorithm to calculate the parameter manipulations that will maximize the ultimate adversarial impact of resulting power flows. Section 4.6 and Section 3.6 present our empirical experiments with real-world commercial power grid monitoring and control software solutions. Section 3.7 discusses the potential mitigation strategies, and Section 3.8 reviews the related work.

3.1.2 Solution Overview

Our contribution builds on two perspectives that have evolved in the emerging field of cybersecurity of networked control systems. The first perspective involves the analysis of state estimation and control algorithms under a class of attacks to sensor measurements or actuator outputs Sandberg et al. (2015). These attack models reflect the loss of availability (resp. integrity) of measurements/outputs when the communication network linking the physical system and remote devices is compromised. Recent work has studied how the physical system's performance and stability can be compromised by such attacks Liu et al. (2011). Typically the attacker is assumed to be a resource-constrained adversary with only partial (or possibly full) knowledge of system, and a resilient control design problem is to ensure a reliable and safe performance against arbitrary actions that can be performed by the attacker. These results are grounded in the theory of robust and intrusion tolerant control, which provides a quantitative framework to study the tradeoffs between efficiency in nominal conditions and robustness during non-nominal ones including the attacker-induced failures. In contrast, as illustrated in Figure 3.1, our attack model considers direct data corruption (specifically, manipulation of power system critical parameters) in the live memory of EMS software, where all distributed sensor measurements are received and processed, i.e., single point of compromise. Hence, individual infections of distributed sensors are not required unlike previous work on false data injection attacks Lu and Zhang (2007). This allows us to study how the vulnerabilities in control software implementations and in their links to external data sources can be exploited by the attackers.

A second perspective has emerged in the vulnerability assessment of large-scale power grids against physical attacks Bienstock (2016). Here the objective is to find worst-case disturbance or an *adversary-optimal attack* to physical components that can maximize the impact on grid functionality, even under perfect observability and best response by the operator (defender). Various classes of failures have been considered, for e.g., line failures, sudden loss of generation, and load disconnects. Typically, these

problems are formulated as bilevel optimization problems, and involve explicit consideration of both physical constraints (e.g., power flows, generation constraints, and line capability limits) as well as resource constraints of the attacker. Examples of physical security problems that have been considered using this framework include $N-k$ contingency analysis problem Davis and Overbye (2011), network interdiction under line failures, and modeling of cascading failures that originate due to local component failures in one sub-network and progressively propagate to other sub-networks of the grid. However, existing work on adversary-optimal attack does not consider how such an attack can be executed in controller software. In our work, we combine the computation of adversary-optimal attack with analysis of EMS software to execute the attack.

Threat model. Our adversary model is concerned with stealthy memory data corruption of EMS (that typically sits within the control center); thus, we require a compromised controller process within the EMS server. This is a realistic assumption, because it requires lesser privileges compared to the past real incidents such as Stuxnet Falliere et al. (2010) and BlackEnergy Assante (2016) that took complete control of the servers. With the access to EMS dynamic memory, the exploit targets the true memory-resident power system critical parameters, and implements calculated adversary-optimal incorrect values in EMS memory.

We emphasize two aspects of our model: Firstly, our attack generation and implementation approach is *generalizable*. However, to concretely illustrate our approach and to evaluate its feasibility, we assume that the attacker is concerned with generating "optimal" dynamic line ratings (DLRs) to maximize capacity violations. Indeed, other variations of attack generation are possible, for e.g. manipulation of other parameters such as generator/loads/voltage bounds, etc. Secondly, our implementation approach is motivated by server-side attacks to EMS software and emphasizes the *stealthiness* of the attack. Specifically, the in-memory parameter manipulations are still within acceptable limits and hence pass the typical out-of-bound checks for false data injections. Thus, they can remain dormant in controller's memory and can produce the intended consequences (e.g. thermal overloading, or even physical damage) before the last line

of defense (i.e., physical fail-safe mechanisms) are triggered. Again, other ways of implementing our attack are possible, for e.g. intercepting network communication and injecting false data.

Implementations. We perform off-line binary analysis to locate the power system parameters in the controller's memory space. We use this information to extract logic-based structural pattern signatures (invariants) about the memory around power system parameter value addresses. The signature predicates are checked during attack-time to identify the real parameters on the victim controller memory space. Such pattern-based search (as opposed to absolute memory address-based search) is required because analysis-time (offline) and attack-time (online) parameter value addresses in memory often differ. This is because of unpredictable execution paths (due to potentially different workloads) across different runs that result in different heap memory allocation function call/return sequences, and hence different allocated memory addresses. Finally, the attack achieves a certain level of stealthiness by ensuring that the incorrect parameters reflect similar general trends as the true ones.

3.2 Optimal Attacks to Economic Dispatch

In this section, we describe how the attacker generates a semantic attack that utilizes the knowledge of an approximate model of power flow to manipulate the model parameters used by the ED software. We choose DC model as the approximate model known by the attacker, and line capacities as the targeted model parameters.

We show that under our adversary model, the allocation generated by the ED implementation under the manipulated capacity ratings, causes the power flows on the transmission lines to exceed the actual line capacity ratings. Specifically, its implementation on the power system will lead to the violation of safe thermal limits of the lines. This can cause the lines to rapidly deteriorate or degrade, increasing their likelihood of tripping. The sudden disconnection of power lines can cause an outage. It may cause a short circuit between two lines that can ignite a fire. Coming in contact with a line that is live, can also kill people, seriously injure them. Thus, such a semantic attack

increases both reliability and safety risks in power system operations to a significant degree.

In our attack model, the attacker chooses the DLR manipulations in a way such that his actions are not obvious to the System Operator (SO). If the effect of the attack is not visible to the SO (for e.g., via line flow measurements or emergency signals), the SO will not invoke generation curtailment and/or line disconnect operations. In fact, under partial network observability, the operator may not be able to implement the necessary preventive actions in a timely manner. As a result, the SO will implement the false ED solution that will violate the line limits.

3.2.1 Attacker Knowledge

We first describe the attacker's system knowledge which consists of DC-approximation of the actual nonlinear AC power flow equations. The topology of a transmission network can be described as a connected graph with the set of nodes $\mathcal{V}$ and the set of edges $\mathcal{E}$. In power systems terminology, each node refers to a bus and each edge refers to a transmission line. We let $n = |\mathcal{V}|$. Let $\{i, j\}$ denote the line joining the nodes i and j, and its susceptance (inverse of reactance) be denoted as β_{ij}. The set of generators at a bus i is denoted as $\mathcal{G}_i$. The set of all generators is denoted by $\mathcal{G} := \mathcal{G}_i$. For each $i \in \mathcal{G}$, p_i^{min} and p_i^{max} are the lower and upper generation bounds that are specific to the $i-$th generator. The generation bounds can be expressed as constraints on individual p_i:

$$p_i^{min} \leqslant p_i \leqslant p_i^{max}. \tag{3.1}$$

Following the standard formulation of economic dispatch, the cost of power generation for the $i-$th generator is modeled as a convex quadratic function $C_i(p_i)$ in p_i. Let $p \in \mathbb{R}^{\mathcal{G}}$ and $d \in \mathbb{R}^{\mathcal{V}}$ denote the generation and demand vectors, respectively. The total cost of generating p is:

$$C(p) = \sum_{i \in \mathcal{G}} C_i(p_i), \tag{3.2}$$

where

$$C_i(p_i) = a_i {p_i}^2 + b_i p_i + c_i. \tag{3.3}$$

$a_i, b_i, c_i \in \mathbb{R}_+ \ \forall \ i \in \mathcal{G}$. a_i and b_i are not simultaneously zero, i.e., the cost of generation is an increasing function of power (MWs) supplied.

The power flow f_{ij} from node i to node j can be expressed as a linear function of the difference between the voltage phase angles at nodes i and j Bienstock (2016):

$$f_{ij} = \beta_{ij}(\theta_i - \theta_j), \tag{3.4}$$

where $\theta \in \mathbb{R}^{\mathcal{V}}$ is the vector of voltage phase angles.

The conservation law for the power flows is:

$$\sum_{j:\{i,j\}\in\mathcal{E}} f_{ij} = \sum_{k\in\mathcal{G}_i} p_k - d_i, \tag{3.5}$$

which states that the net generation at a node i is equal to the sum of outflows from node i to its neighbors. The DC power flow (3.4)-(3.5) is said to be feasible if and only if total supply is equal to total demand (see Bienstock (2016)), i.e.,

$$\sum_{i\in\mathcal{G}} p_i - \sum_{j\in\mathcal{V}} d_j = 0. \tag{3.6}$$

The power flows satisfy the capacity line constraints, i.e.,

$$|f_{ij}| \leqslant u_{ij}. \tag{3.7}$$

Thus the DC-optimal power flow problem faced by the SO can be posed as follows:

$$\min_{p,\theta} \quad C(p) \qquad\qquad \text{s.t. } (3.1) - (3.6), (3.7). \tag{3.8}$$

3.2.2 Attacker Resources

The true capacities of the transmission lines dynamically vary over time due to weather conditions (ambient temperature, wind, etc.) Department of Energy (2016a), and are, in fact, greater than the static line ratings assumed by the SO for economic dispatch problem (fig. 3.2). Dynamic Line Rating (DLR) lines are the transmission lines with

DLR sensors that report the true line capacities to the system operator.

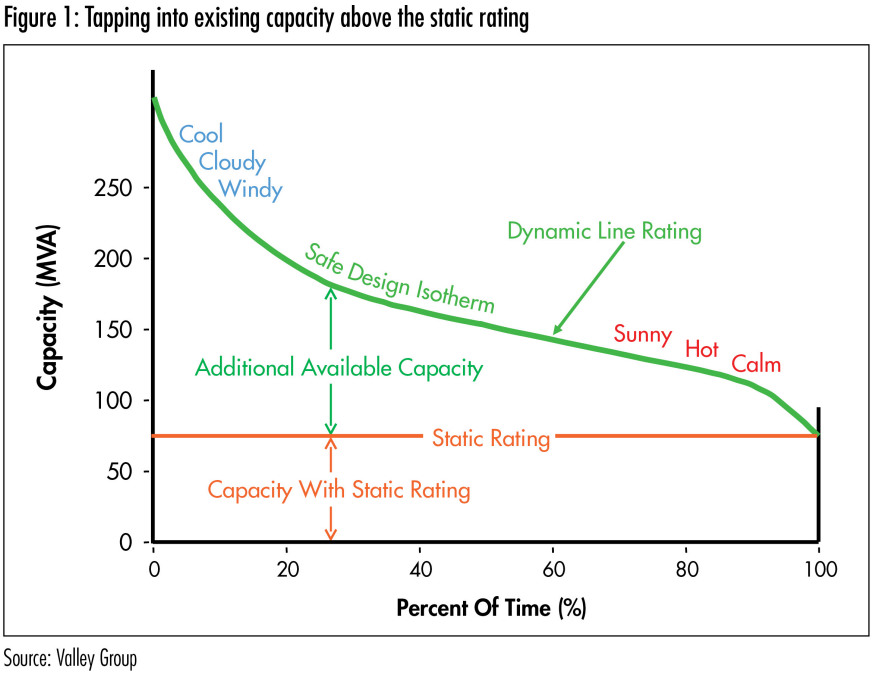


Figure 3.2: Static vs dynamic line rating

Let $\mathcal{E}_D \subset \mathcal{E}$ denote the set of lines that are equipped with DLR devices. The complementary set $\mathcal{E}_S = \mathcal{E} \backslash \mathcal{E}_D$ denotes the set of lines that are not equipped with DLR technology, and hence their rating will be fixed to the respective static line capacity values. Given that DLR deployments are done as part of government sponsored smart grid projects Department of Energy (2016a,b), the set of lines $\mathcal{E}_D$ equipped with DLR technology is public knowledge. These lines will be the ones that are routinely prone to congestion and hence receive priority DLR implementation by the operator.

For a line $\{i, j\} \in \mathcal{E}_D$, we denote u_{ij}^d as the actual line rating computed by the DLR software using measurements collected from the Supervisory Control and Data Acquisition (SCADA) system.

$$u_{ij} = \begin{cases} u_{ij}^s & \text{if } \{i,j\} \in \mathcal{E}_S \\ u_{ij}^d & \text{if } \{i,j\} \in \mathcal{E}_D, \end{cases} \tag{3.9}$$

where

$$\forall\ \{i,j\} \in \mathcal{E}_D \quad u_{ij}^{min} \leqslant u_{ij}^d \leqslant u_{ij}^{max} \tag{3.10}$$

i.e. the DLRs can only take values between a certain range.

Thus the DC-optimal power flow problem faced by the SO can be posed as follows:

$$\min_{p,\theta} \quad C(p) \qquad \text{s.t. } (3.1) - (3.6), (3.7), (3.9). \tag{3.11}$$

We assume an *informed attacker*. Specifically, the attacker's knowledge includes the network topology, line susceptances, set of generators, and their corresponding generation limits, and the cost of generation. The attacker also knows the nominal demand d_j at each node j and the nominal generator output p_i for each $i \in \mathcal{G}$. In power systems terminology, with this knowledge, the attacker can solve for an DC ED solution which is an approximation of AC ED solution that the EMS implements on the power system. Note that our assumption on attacker's knowledge is not unrealistic given that all major ISOs publicly disclose historical generation and demand patterns and the locational marginal prices in day ahead and hourly power markets.

Since the SO knows the static line ratings and these are fixed in ED software implementations, we assume that the attacker cannot compromise them in ED implementation's memory. Any compromise to static line ratings can be overridden by simple built-in checks in power flow implementations. Also, since the static ratings are typically calculated for constant (worst-case) weather conditions over an extended period of time (few months to years), we assume that the attacker knows their values. This assumption can be justified by the fact that the manufacturers of transmission line conductors supply static line ratings in their product specifications. Thus, under the aforementioned constraints, the set of lines $\mathcal{E}_D$ constitutes the attacker's constraint since the attacker only targets DLR ratings and not the static ones.

3.2.3 Attack Objective

Now, we present the constraints faced by the attacker so that the attack remains stealthy, and the SO's ED software admits the DLR ratings manipulated by the attacker. Then, we formulate the attack policy of the attacker as a bilevel optimization problem.

Under our attack model, the attacker accesses the actual DLR values u_{ij}^d for lines $\{i,j\} \in \mathcal{E}_D$ in ED's dynamic memory and replaces them with incorrect values u_{ij}^a (Section 3.6). The attacker knows u_{ij}^d and computes u_{ij}^a in order to maximize the violation of line ratings by the resulting power flows. To avoid detection by in-built checks, each u_{ij}^a is constrained by minimum and maximum permissible limits of line ratings, denoted as u_{ij}^{min} and u_{ij}^{max}, respectively. These limits are also known by the attacker. For ease of presentation, we introduce an auxiliary variable $\widehat{u}_{ij}$ to denote the manipulated dynamic line rating for the line $\{i,j\} \in \mathcal{E}_D$. The attacker is subject to following constraints:

$$\forall\ \{i,j\} \in \mathcal{E}_D \quad \begin{cases} \widehat{u}_{ij} = u_{ij}^a \\ u_{ij}^{min} \leqslant \widehat{u}_{ij} \leqslant u_{ij}^{max}, \end{cases} \tag{3.12}$$

because, the attacker wants to maintain a level of stealthiness, so he does not choose an out-of-bound DLR value, that may set off an alarm.

If the DLR value of a line $\{i,j\} \in \mathcal{E}_D$ is manipulated by the attacker, then the EMS software will obey the following constraint for the power flow on the line:

$$\forall\ \{i,j\} \in \mathcal{E} \quad |f_{ij}| \leqslant \widehat{u}_{ij}. \tag{3.13}$$

We pose the problem of optimal attack generation – from the attacker's viewpoint – as the following bilevel optimization problem:

$$\max_{u^a} \qquad U_{cap}(f; u^d) \qquad = \max_{\{i,j\} \in \mathcal{E}_D} 100 \left(\frac{|f_{ij}|}{u_{ij}^d} - 1 \right)_+ \tag{3.14a}$$

$$\min_{p,\theta} \quad C(p) \qquad \text{s.t. } (3.1) - (3.6), (3.12), (3.13), \tag{3.14b}$$

where $a_+ := \max(a, 0)$. This problem is equivalent to a 2-stage sequential (Stackelberg) game, in which the attacker (leader) chooses his strategy assuming a best response from the defender (follower). Specifically, in the first stage, the attacker chooses the incorrect DLR ratings u^a (or equivalently $\widehat{u}_{ij}$) that are subsequently implemented in runtime by localizing and corrupting true DLR values in the nonlinear ED controller's

memory (Section 3.6). The attacker's objective is to maximize the maximum percentage capacity bound violation of the power flows f_{ij} on lines $\{i,j\} \in \mathcal{E}_D$ over the true DLR values u^d_{ij} after the defender responds optimally in the second stage. This objective can be expressed as $U_{cap}(f; u^d)$ in (3.14a). In the second stage, the defender chooses the generator outputs p and voltage phase angles θ that achieves min-cost solution to DC-ED, i.e., minimize the generation costs (3.2) subject to the constraints (3.1)-(3.6),(3.12),(3.13). The attacker ensures that under the manipulated DLR ratings $\widehat{u}_{ij}$ for lines $\{i,j\} \in \mathcal{E}_D$ and given static ratings u^s_{ij} for lines $\{i,j\} \in \mathcal{E}_S$, there exists a feasible flow allocation that minimizes the generation cost (3.2), otherwise the SO will be require to setting off an alarm causing the SO to initiate other actions such as load curtailment.

Note that the actual generation cost faced by the operator when incorrect u^a are used in the SO's nonlinear ED formulation will be different than the defender cost obtained in the stage 2 subgame. In fact, the nonlinear ED is likely to be infeasible in the sense that the power flows on certain lines can exceed the permissible line ratings.

The attack model can be summarized as follows. The physical system consists of the physical components, e.g., generators, transmission network, and the loads. Each of these components send data to the EMS via means of SCADA, which is part of the attacker knowledge. The generators submit the cost functions, the transmission network submits the topology and the line ratings, and the loads submit the demand. The attacker uses this data to compute a DLR manipulation based on his attack policy, and then compromises the DLR values utilized by the EMS while solving the ED problem. Finally, the EMS implements the false ED solution by dispatching the new generation set-points to the individual generators.

Next, we present our computational approach to compute the optimal maximin attack.

3.3 Characteristics of Optimal Attack

The optimal attack generation problem posed in (3.14) is a linear-quadratic bilevel (LQBP) that is, in general, computationally hard to solve. One of the standard approaches to solve a LQBP is to reformulated it as a Mixed Integer Linear Program (MILP), which can be implemented using commonly available optimization solvers.

Our approach for solving the bilevel optimization problem (3.14) is as follows. First, we divide the main problem as $2\,|\mathcal{E}_D|$ parallel optimization problems where the attacker's objective is to just maximize the capacity violation of one DLR line, in either flow direction. This converts the attacker's objective function from nonlinear to an affine function. This subproblem can be represented as follows:

$$\begin{aligned} \max_{x \in X} \quad & {g_1}^T x + {g_2}^T y^\star \\ \text{s.t.} \quad & A_1 x + B_1 y^\star \leqslant k_1 \\ y^\star \in \min_{y} \quad & \frac{1}{2} y^T H y + {h_1}^T y + h_2 \\ \text{s.t.} \quad & A_2 x + B_2 y \leqslant k_2, \end{aligned} \tag{3.15}$$

where x denotes the attacker actions; X denotes the non-negativity and/or integrality constraints. In the subproblem of (3.14), $x = u^a$, $y = (p, \theta)$, $X = \{u \in \mathbb{R}^{\mathcal{E}_D} : u^{min} \leqslant u \leqslant u^{max}\}$. Also, g_1, B_1 are zero vector and zero matrix, respectively.

Second, we note that, for fixed attacker action x, the inner problem is a convex minimization problem, and therefore strong duality applies. Applying the Karush-Kuhn-Tucker (KKT) conditions for the optimal solution of the inner problem, we can pose the overall bilevel problem as a MILP Zeng and An (2014). Let, for fixed attacker action x, $(y^\star, \lambda^\star)$ denote the optimal primal-dual pair for the inner problem. Then the

KKT optimality conditions are as follows.

$$A_2 x + B_2 y^\star \leqslant k_2 \tag{3.16a}$$

$$\lambda^\star \geqslant 0 \tag{3.16b}$$

$$H y^\star + {B_2}^T \lambda^\star + h_1 = 0 \tag{3.16c}$$

$$\lambda^\star \leqslant M(\mathbf{1}_n - \mu)$$

$$A_2 x + B_2 y^\star - k_2 \leqslant M\mu \tag{3.16d}$$

$$\forall\ i \in \{1, 2, \cdots, m\}, \qquad \mu_i \in \{0, 1\},$$

where $m = length(k_2)$, M is infinity (chosen as a significantly large number). (3.16a), (3.16b), (3.16c) and (3.16d) are primal feasibility, dual feasibility, stationarity and complementary slackness conditions. Note that the complementary slackness conditions are reformulated into integrality constraints.

Thus, the bilevel subproblem can be restated as a single-level mixed-integer linear program (MILP).

$$\begin{aligned} \max_{x \in X} \quad & {g_1}^T x + {g_2}^T y^\star \\ \text{s.t.} \quad & A_1 x + B_1 y^\star \leqslant k_1, \text{ and (3.16).} \end{aligned} \tag{3.17}$$

Third, we solve for $2\,|\mathcal{E}_D|$ copies of the above MILP (3.17), and choose the maximum over all DLR lines, the non-negative percentage capacity bound violation, in either flow direction.

Our approach is summarized in algorithm 1. The procedure GetOptimalAttack() initializes the optimal attacker strategy and optimal attacker gain to zero. It constructs the MILP model with the KKT conditions for the inner problem and the feasibility constraints for the outer decision variables, by calling the procedure GetMILPModel(). Then, for each DLR line and each flow direction, GetEdgeAttack sets the objective function as the percentage capacity violation for that line. During each iteration, if the attacker's gain computed is larger than the previously computed

Algorithm 1 Optimal security strategy

```
1: (U*_cap, u^a*) ← GetOptimalAttack()
2: procedure GetOptimalAttack()
3:     U*_cap ← 0, u^a* ← 0
4:     m = GetMILPModel()  using (3.17)
5:     for {i, j} ∈ E_D do                                   ▷ for each DLR line
6:         for dir ∈ {−1, 1} do                              ▷ for each flow direction
7:             SetObjective(m, 100 ((dir × f_ij)/u^d_ij − 1))
8:             solve(m)
9:             U_cap ←getObjectiveValue(m)
10:            u^a ← getValue(m,u^a)
11:            if U_cap > U*_cap then
12:                (U*_cap, u^a*) ← (U_cap, u^a)             ▷ update values
13:            end if
14:        end for
15:    end for
16:    return U*_cap, u^a*
17: end procedure
```

value, then the values for the optimal attacker's gain and the corresponding optimal attack strategy are updated. As we will see in Section 3.4.2, this computational approach is indeed scalable to larger networks.

3.4 Computational Results

We discuss the structure of optimal attacks on benchmark power networks with DLRs, and discuss its implications on line capacity violations and increased generation costs.

3.4.1 3-node Example

We now illustrate the optimal attacker strategy with the help of a benchmark example. We consider a 3-node network as shown in Figure 3.3. It consists of 2 generators G_1, G_2 at bus 1 and 2, respectively, and a load L on bus 3.

The following assumptions enable the computation of optimal attack in closed form. The nominal voltage magnitude is $V^{\text{nom}} = 230\ kV$ and the upper and lower voltage bounds are given by $\overline{V} = 1.1V^{\text{nom}}, \underline{V} = 0.9V^{\text{nom}}$, respectively. The three lines are identical, each with impedance $z = 0.002 + 0.05\mathbf{j}$ in per unit system. Thus, the susceptance of each line is the inverse of reactance given by $\beta = \frac{1}{0.05}$. Assume that for the given instance, the active DLR for each of the three lines is 160 MW. The generation

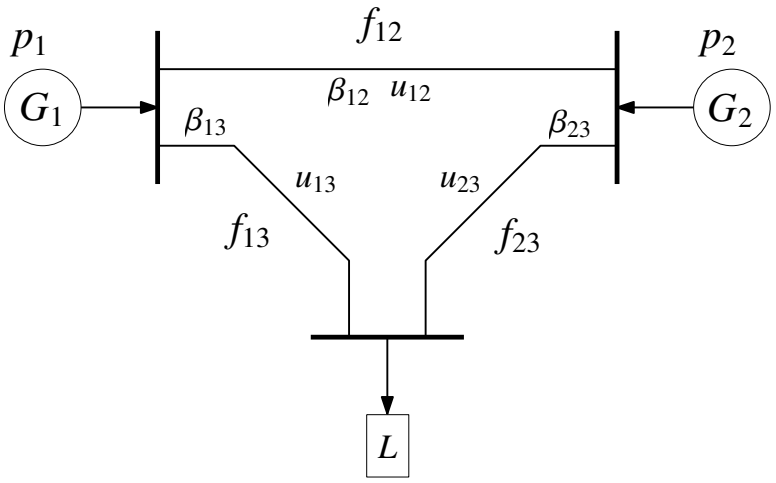
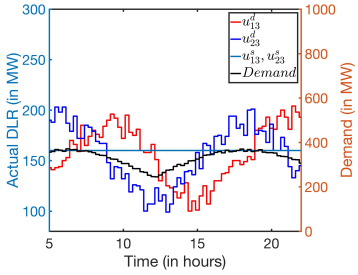
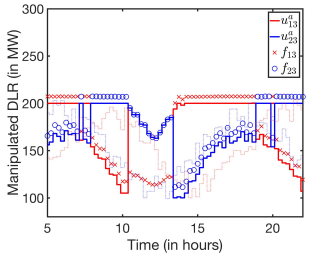


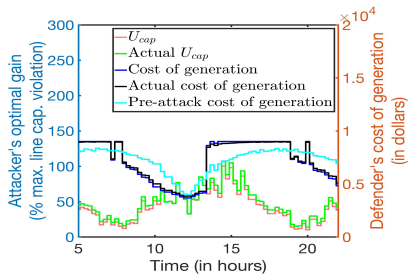
Figure 3.3: Three-bus power system.



(a) Possible DLR and demand pattern over 24 hour horizon.



(b) Time of attack. The actual DLR ratings u^d are shown as lightly dashed lines in the background for comparison.



(c) Attacker's gain and SO's cost of generation as predicted by the bilevel model (3.14), and as computed by MATPOWER.

Figure 3.4: Results for three-node power grid.

output of the two generators must satisfy the bounds $0 \leqslant p_1, p_2 \leqslant 300$ MW. Bus 3 has a constant power load having demand $d = 300$ MW.

Consider, for simplicity, a linear power flow model (3.4)-(3.5), and the linear cost of generation given by

$$C(p) = b_1 p_1 + b_2 p_2, \tag{3.18}$$

where we choose $b_1 = 2b_2 = 2b > 0$. Simplifying further, we get, $C(p) = b_1 p_1 + b_2(d - p_1) = bp_1 + bd$.

In the "no attack" case, the optimal generation turns out to be $(p_1, p_2) = (120, 180)$. The power flows at this point are $f_{12} = -20$, $f_{13} = 140$, and $f_{23} = 160$, respectively. As a result, the most congested line among all the three lines is line $\{2, 3\}$. This is expected

u^d_{13}	u^d_{23}	u^a_{13}	u^a_{23}	f_{13}	f_{23}	U_{cap} (in 10^5\$)
130	120	100	200	100	200	80
130	150	200	100	200	100	70
160	150	100	200	100	200	50
160	180	200	100	200	100	40

Table 3.1: Optimal attacker strategy for three-bus test case.

as the G_2 has lower cost of production, so it generates more causing the congestion in line $\{2, 3\}$.

Assume for the sake of illustration that only the DLRs of lines $\{1, 3\}$ and $\{2, 3\}$ can be manipulated. The attacker's strategy will be either to maximize the capacity violation on line $\{2, 3\}$ (strategy A) or that on line $\{1, 3\}$ (strategy B). The attacker's optimal strategy is the one which leads to larger of these two violations. Assuming that the demand is fixed at 300, under strategy A (resp. strategy B), the optimal manipulated DLRs will be $u^a_{13}, u^a_{23} = (100, 200)$ (resp. $(200, 100)$). Table 3.1 lists some possible combinations for the actual DLR values of lines $\{1, 3\}$ and $\{2, 3\}$, and the corresponding optimal attacker strategies. For example, if $(u^d_{13}, u^d_{23}) = (120, 120)$, then the optimal attacker strategy is strategy A, i.e. $(u^a_{13}, u^a_{23}) = (100, 200)$, which yields attacker objective value as $U_{cap} = 80$.

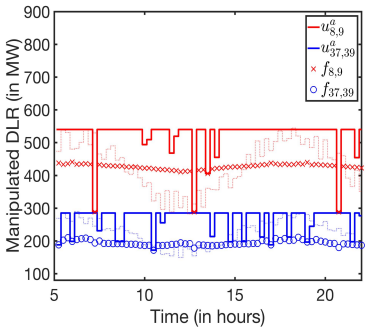
Now let us use the aforementioned approach to generate optimal DLR manipulations when the demand and DLRs vary over time, and OPF calculations account for manipulated line ratings to generate power flow allocations. For the 3-node network (Figure 3.3), consider the demand pattern at node 3 and the representative DLR for two lines $\{1, 3\}$ and $\{2, 3\}$ as shown in fig. 3.4a. We instantiate the OPF models at every 15 minutes using this demand pattern. The aggregate demand curve has two peaks corresponding to the morning and evening peak periods. We chose the lower and upper bounds for the DLR values to be 100 and 200 MW. Then we varied DLRs between these bounds to generate patterns for 24 hour period. For the sake of illustration, we consider the two DLR curves to have sinusoidal patterns with certain offset between the two. The pattern also models the increased capacity due to favorable conditions (e.g. wind) during certain parts of the day. For these DLR and demand patterns, we determine how the attacker strategy and the attacker's gain varies over time with respect to the

true DLRs and the demand.

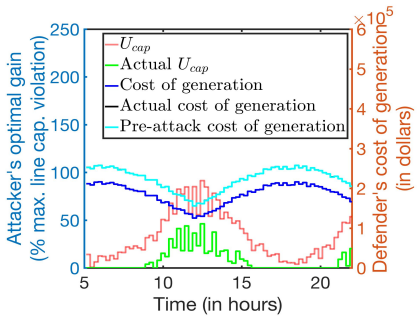
fig. 3.4b shows the non-linear power flows along the DLR lines when the attacker's DLR ratings are in effect. We observe that the non-linear power flows are greater than the attacker's DLR ratings because of the presence of the reactive power which is not accounted by the linear power flow model assumed by the attacker in generating the optimal attack.

We also note that if the attacker targets line $\{2,3\}$ (strategy A), then the optimal attack can reach to maximum DLR rating, i.e., u^a_{23} can assume the value u^{max}_{23} for certain time periods. Recall that the bilevel formulation is constrained by the supply-demand balance in the defender's response. This constraint becomes tight for a range of time-periods during which the optimal attack u^a_{13} tracks the power flow f_{13} on line $\{1,3\}$. If the true DLRs are such that $u^d_{23} > u^d_{13}$, then the attacker chooses $u^a_{23} = u^{max}_{23}$. To ensure that the supply = demand constraint is met u^a_{23} is just equal to the power flow required to flow on line $\{1,3\}$. On the other hand if $u^d_{23} < u^d_{13}$, then the optimal attacker strategy is to violate the capacity of line $\{1,3\}$ (strategy B).

We evaluated the attacker's gain (U_{cap}) and the defender's cost of generation both estimated by the bilevel formulation (3.14) and by the nonlinear computations using MATPOWER (see fig. 3.4c). The respective curves closely follow each other. The actual cost of generation under nonlinear power flows is slightly larger than the cost of generation estimated under linear power flows. The same is also true for the attacker's gain U_{cap}. Comparing the demand and DLR variations in fig. 3.4a and the objective functions in fig. 3.4c, we can see that the optimal attacker gain is not achieved when the network experiences heavy demand. Rather, the optimal gain is achieved when the network is heavily congested, i.e., relative to the network's capacity, the aggregate demand is high. This gives an important insight into the optimal time for the attack. For e.g., during the hot summers and low windy conditions, the lines have lower capacities than during the winters. Also, the high temperatures lead to more aggregate demand during the summers. Hence, the attacker is better off manipulating the DLRs in high temperature conditions.



(a) Time of attack for 118-node power network.



(b) Loss functions for 118-node network.

Figure 3.5: Results for 118-node network

3.4.2 Scalability of attack

To demonstrate the scalability of our approach, we implemented algorithm 1 on an 118-node network. We choose the DLR and demand patterns for the 118-node network similar to the ones in 3-node network, but in contrast to the linear generation cost (3.18), we adopt the more realistic convex quadratic cost function (3.3). In this chapter, we have used Gurobi which is a state-of-the-art optimization toolbox and has built-in support for solving MILP problems. figs. 3.5a and 3.5b show the corresponding computational results for an 118 node network. Due to the fact that actual power flows also consist of reactive power flows in addition to real power flows, there are higher line losses, resulting in more total power generation that increases the cost of generation. However, we see that the actual attacker's gain is lower than the estimate obtained by solving (3.14) (fig. 3.4c). This can be explained as follows. The generators have different quadratic curves for the cost of generation. As a result for lower network load, one set of generators may be more contributing to the generation, but for higher loads, other set of generators may be the more contributing ones. This results in lower power flows along the DLR lines during high demand conditions. Hence, in the case of low aggregate demand, the DLR lines are violated to a larger extent than in the case of high demand. Another important observation is that the attacker's gain can be high even if the demand is low, because the actual DLRs may be even lower.

In the next section, we describe how an attacker can implement the optimal attack

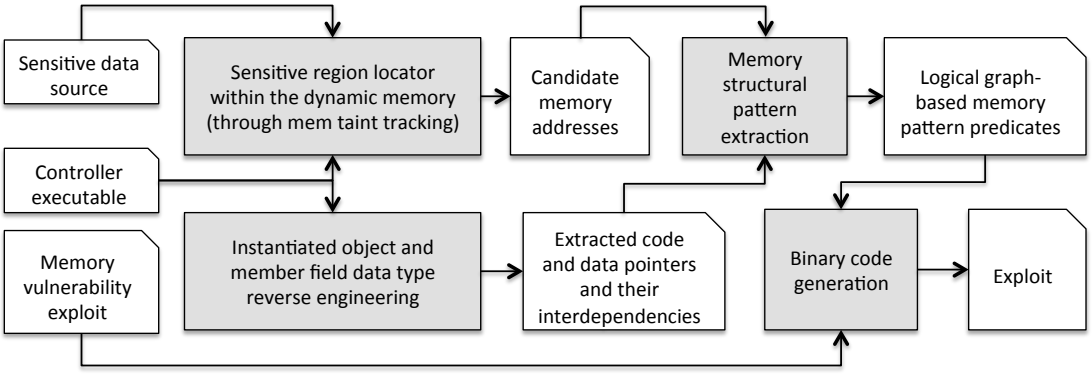


Figure 3.6: Flowchart for attack implementation.

Intra-Class Pattern (type)	Code Pointer Pattern (content)	Data Pointer Pattern (relation)

class A size(20):
+---
0 | {vfptr} // virtual fn table*
4 | line-rating // target parameter
8 | mem-var2
12 | mem-var3
16 | line-name // char* string
+---
12 bytes
	class B size(8):
	+---
	0 | {vfptr} // virtual fn table*
	4 | line-rating // target parameter
	+---
	B's vftable:
	0 | &A::A_virt1
	4 | &A::A_virt2
	53 | push ebx
	56 | push esi
	8B F2 | mov esi, edx
		class C size(16):
		0 | {vfptr}
		4 | linked_list_prev // previous node
		8 | linked_list_next // next node
		12 | lr // target parameter
		class C size(16):
		0 | {vfptr}
		4 | linked_list_prev // previous node
		8 | linked_list_next // next node
		12 | lr // target parameter

type(&line-rating + 0x0C) == string	*(*(&line-rating-0x04)+0x04) == 0x53568BF2	*(*(&lr - 0x08) + 0x04) == (&lr - 0x10)

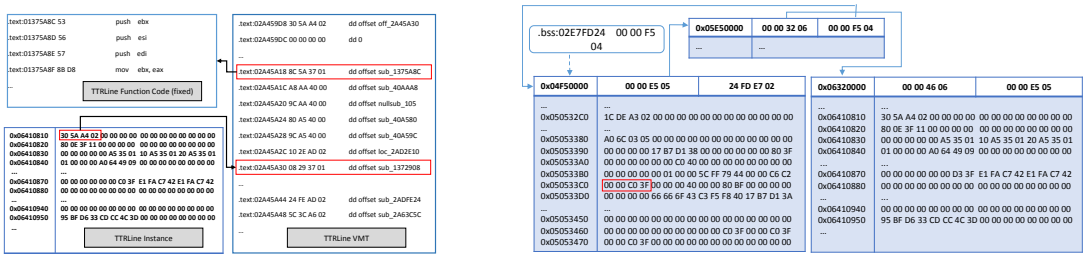
Table 3.2: Logical memory structure signatures for critical parameters.

as computed by the bilevel formulation (3.14), as a cyberattack targeting the EMS softwares. Specifically, we will show how an EMS software (e.g., PowerWorld[2]) be targeted such that the values of the DLRs in the memory of the software will change during run-time. This will cause the ED implementation in the EMS to yield a false ED solution.

3.5 Implementations

We implemented our proposed attack in real controller software packages. Figure 3.6 shows the stages of the implemented attack. Initially, we assume a controller executable file (vulnerable point) and sensitive data sources (e.g., inputs such as DLRs originating from an external source) are given. Next, through memory taint analysis, we narrow down our search space to identify the the memory regions where the sensitive parameters

[2]We have taken the necessary responsible disclosure steps and have informed the vendors about our research findings. It is noteworthy that we are not reporting a security software vulnerability in this chapter. Instead, assuming there is a potential exploit, we demonstrate how the adversaries can perform domain-specific data corruption in memory to impact the produced control actuation commands. The steps are not specific to any commercial software package.



(a) Code pointer-instruction pattern. (b) Linked-list as data pointer-based pattern.

Figure 3.7: Code and data pointer-based structural memory patterns in PowerWorld used for graphical predicate generation.

may reside in memory during the controller execution. Accordingly, all the memory regions affected by the target input are marked (tainted). The tainted areas are then searched for the values of interest (e.g., target DLRs), and candidates are shortlisted. To identify the correct candidate from the set of candidates, we generate structural memory pattern signatures around the correct candidates during the offline binary analysis phase. We use our past work Sun et al. (2016) to extract binary-level data type and code, and data pointers and their interdependencies (discussed below). Given the reverse engineered logical memory layout, we create structural patterns of the memory regarding where the target parameters reside. Those patterns are then used to generate the exploit binary. During the attack phase the exploit searches the dynamic memory address space to locate the target parameters using the patterns. Finally, it changes the identified parameter values to the optimal attack values, as discussed in Section 3.3.

Every control algorithm implementation by controller software executables involve code and data. The code instructions encode the algorithm logic (e.g., iterative optimization loops), whereas the data stores the controller parameters such as the OPF constraints and DLRs. Modification of the code instructions are often infeasible due to $W \oplus X$ protections. However, the data regions should be (and are set as) writable, because the EMS operators often update their values dynamically according to the most recent power system configuration.

Maintenance of control-sensitive variable values such as DLRs by the controller software provides an attack surface to modify them in memory space during the attack. Our investigations of EMS software binaries showed heavy use of data structures and

class objects to store those values that are used directly by OPF. During the offline phase, we analyzed the EMS software binary to determine its memory's structural layout. We are interested in structural information such as the allocated class instances (objects), the class hierarchy, and the logical interdependencies between the instantiated objects within the memory, e.g., cross-object code and data pointers. We are not interested in exact object memory addresses, because the addresses will likely differ during the attack due to unpredictable (inputs and hence) dynamic execution paths. Instead, by capturing the logical interconnections among the instantiated memory-resident objects, we extracted invariants about their interdependencies that remain the same across different runs. The attacker later uses the invariants during the attack to locate (and corrupt) the DLR values.

Search for a specific DLR value during the attack results in several memory-resident candidates that are mostly (except one) false positives. To identify the correct candidate, our implementation uses the invariants, expressed as propositional logic predicates, that capture the logical memory structural patterns around the target DLR parameters. We use three kinds of memory patterns: address-relative intra-class type patterns, code pointer-instruction patterns, and data pointer-based patterns (Table 3.2).

Address-relative intra-class type patterns. The attack extracts execution-agnostic memory structural patterns around the target DLR values in memory. We concentrate on intra-class patterns that capture fixed offset relations among members of the same class as the target DLR parameter, and their types and/or values. If the DLR parameter is stored as a member of a class that also contains other variable(s), whose type is (are) easy to identify, we use that information as a local signature for the target parameter. In memory forensics, types such as character strings, pointers Lin et al. (2010), and fixed-value member fields can be identified simply. We investigate the vicinity of the target parameter within the same object looking for addresses that store easy-to-identify data types. If one or more of such samples are found, their type/-value and corresponding offset from the target parameter address is used to produce the signature. The attack creates simple-to-check logical predicates for each candidate (e.g., "candidate_addr + $0x08$ stores $0x00000001$"). Our implementation aggregates

the produced predicates into a single conjunctive logic signature.

Code pointer-instruction patterns. We leverage the code pointer relations within the memory regions to extract invariants (logical predicates) about the structural memory layout around the target DLR parameters. We extract such invariants given the reverse engineered class object pointers, and their logical interdependencies with the corresponding member and virtual functions. We use the fact that code segments (e.g., instructions of member and virtual functions) within the controller software binary are typically set as read-only with fixed content. Table 3.2 shows a sample code pointer-based predicate for the illustrated pattern. The signature checks whether the first four byte content of the target parameter's object's second virtual function is equal to the corresponding function prologue. As denoted, the signature does not depend on the absolute address values given the target parameter candidate's location. The attack can automatically generate the code pointer patterns for the object's individual member and virtual functions. Finally, the generated predicates are combined into a single conjunctive logical predicate to check against all the identified candidates within the EMS memory space attack time.

Data pointer-based patterns. The data pointer-based patterns do not often assume fixed data values in memory, and is purely based on memory structure and the relations between various objects. We perform a recursive pointer traversal among the recognized objects on the controller's memory space following its earlier forensics analyses of the allocated objects and the stored pointer values within them (member fields). The algorithm implements a depth-first search starting from individual recognized pointers within the memory space. For each pointer under the consideration, we determine if its destination is an memory-resident object. If so, the attack recursively traverses all the member pointer fields within the destination object. During its recursive search, our implementation generates the corresponding directed graph, where nodes represent allocated objects, and the outgoing edges indicate the member pointer fields within the source object. The generated directed graph represents the inter-object dependencies within the memory space. Once the generation of the graph in completed, our implementation searches for cycles. Such cycles are very popular in widely used data

Param. values	#Hits	#Relevant	#Recognized	Accuracy
0x3FC00000	143	3	3	100%
0x02A45A30	2038	4	4	100%
0x06410570	30	1	1	100%
0x06410810	30	1	1	100%
0x06410810	28	1	1	100%

Table 3.3: The target parameter value recognition accuracy.

structures such as linked lists (the rightmost entry on Table 3.2). The attack turns each cycle within the graph into a logical predicate that corresponds to a data pointer-based signature.

3.6 Empirical Attack Deployment Results

To assess the proposed attack feasibility in practice, we implemented it against widely-used commercial and open-source industrial controller software packages. The implemented attack involves the following steps: *i)* during the offline phase, we reverse engineer the EMS software binary to locate DLR parameters within the controller and create the corresponding invariants that hold true regardless of their absolute memory addresses; *ii)* during the online phase (attack time), the exploit searches the controller memory for the known legitimate DLR values and collects the candidates; *iii)* the attack recognizes the only true candidate by applying the invariants on the collected set of candidates; and *iv)* our implementation modifies the value maliciously according to the optimal attack generation algorithms discussed in the previous section. We now explain the results for our empirical validation.

3.6.1 EMS Software Attack

We validated the proposed attack on real-world widely-used industrial controller software packages. We first present the detailed results on PowerWorld, and later compare the attack's performance for other controllers (NEPLAN, PowerFactory, PowerTools, and SmartGridToolbox).

Figure 3.7a shows a generated code pointer-based memory signature in PowerWorld. The corresponding pattern predicate for runtime memory search was "*(*(candidate_addr - 0x54) - 0x24) == 0x5356578B", where 0x5356578B is the hex representation of the sub_1375A8C function's first four instruction bytes. The rating of every transmission line is stored in offset 0x24 of the corresponding TTRLine object. The information about the transmission lines of the power system is stored as a doubly linked list of TTRLine objects in PowerWorld memory space. The attack used "*(*(candidate_addr - 0x24) + 0x04) == (candidate_addr - 0x24)" as the pattern predicate for line ratings. Let us call the linked list node that stores the target line rating A. The pattern predicate above essentially verifies the following linked list invariant: whether A's previous node's next pointer points to A. More complex patterns can be extracted if needed; however, our empirical studies on PowerWorld shows simple patterns always suffice to identify and isolate the exact candidate uniquely.

Figure 3.7b shows another PowerWorld data pointer pattern for line ratings. PowerWorld allocates linked list nodes (0x13FFF0 sizes each) allocated by VirtualAlloc for objects instances of different classes (e.g., TGen, TBus and TTRLine). Only three nodes are shown. If our objective is to look for line rating 0x3FC00000, its corresponding pattern predicate will encode the offset to get the node's initial member value 0x05E50000 that points to the next node shown (summarized) on the top of the figure. The second element of each node (0x04F50000 in the top node) points to the previous node. A relatively more complex second-degree predicate would be "*(*(*(*(candidate_addr - 0x1033C0)) + 0x04)+ 0x04) == candidate_addr - 0x1033C0", i.e., $A \rightarrow next \rightarrow next \rightarrow previous \rightarrow previous == A$, where A represents the data structure that stores the line rating 0x3FC00000.

The attack payload checks for patterns on the identified candidates before corrupting their values. The code searches for the specific value in memory, and modifies the identified candidate. Table 3.3 shows how many hits our implementation finds for individual target power system parameter values on PowerWorld memory space. The number empirically proves the infeasibility of memory corruption attacks without the use of signature predicates. The next column shows how well the signatures dismiss the

EMS Software	vfTable	Line	Bus	Gen.	Accuracy
`PowerWorld`	8527	3	3	2	100%
`NEPLAN`	6549	51	30	5	100%
`PowerFactory`	110	34	39	10	100%
`Powertools`	3	185	118	53	100%
`SmartGridToolbox`	194	79	57	4	100%

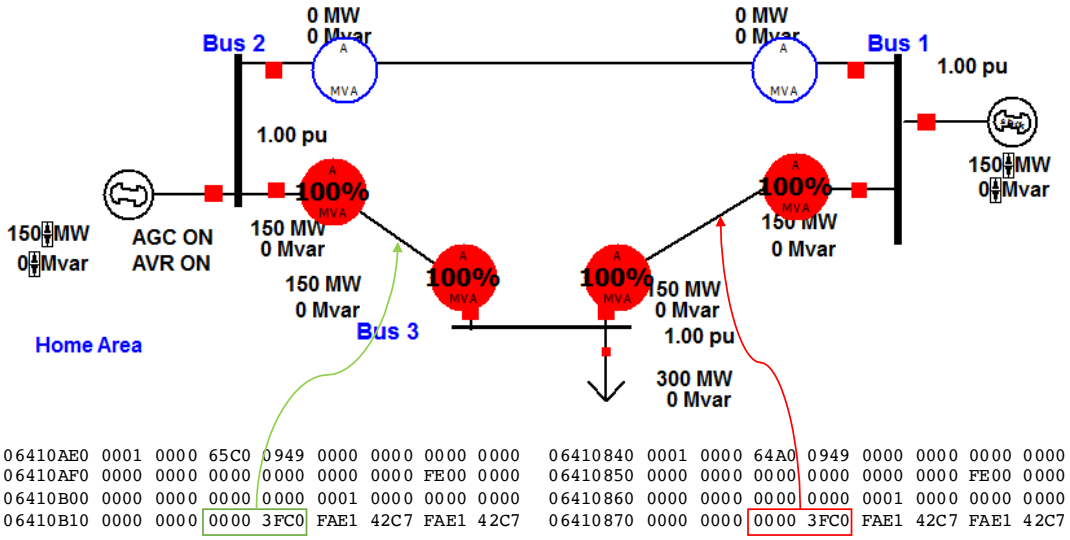
Table 3.4: Memory layout (object) forensics accuracy. The instances were correctly marked with their types.

irrelevant candidates and identify the true target values. Table 3.4 shows the forensics analysis accuracy for five different EMS software packages. Through the use of the code pointer signatures and its extracted knowledge about the class hierarchies, our implementation was able to correctly recognize the class types of all object instances within the EMS memory. The payload initializes the OPF algorithm in its corresponding thread. Once it changes the identified memory addresses, it restarts the control loop through the call to `CreateThread` function within `kernel32.dll` that is loaded by almost all windows processes.

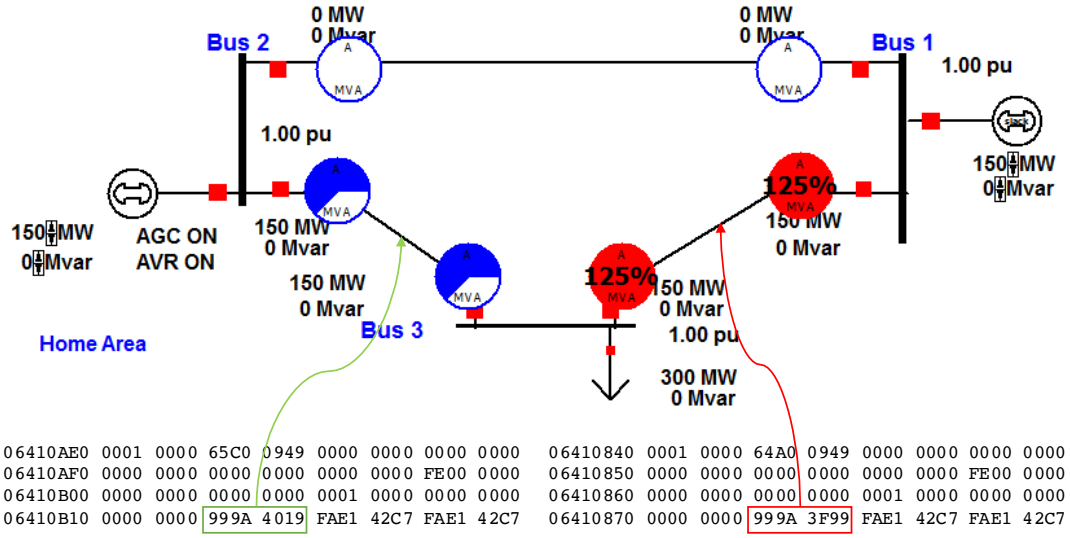
3.6.2 Case-study Demonstration

As a concrete example, we show how the state of underlying power system (the same model used in Section 4.6) gets affected once the memory corruption is completed (Figure 3.8[3]). Before the corruption (Figure 3.8a), the EMS GUI visualizes the safe state of power system operation, where the transmission lines are mostly fully utilized; however, no line rating (capacity constraints) are violated. The optimal attack generation algorithm computes the adversary-optimal values for the line ratings, and chooses to *i)* modify the $B1-B3$ transmission line to $120MW$ from $150MW$; and *ii)* modify the line rating for the $B2-B3$ transmission line to $240MW$ from $150MW$. While implementing the optimal attacker strategies that we obtain from the maximin solution, we need to translate the line rating values to higher values using basic power flow calculations. For example, for the implementation of optimal attack, we use $\widehat{u}_{13} = 120\ MVA$ and $\widehat{u}_{23} = 240\ MVA$. These values are higher than the values $\widehat{u}_{13} = 100$ and $\widehat{u}_{23} = 200$

[3]The pie charts on the transmission lines represent the used percentages of the line power flow capacities in that particular state.



(a) PowerWorld pre-attack power system state (safe).



(b) PowerWorld post-attack power system state (unsafe).

fbus	tbus	r	x	b	rateA	rateB	rateC	ratio	angle	status	angmin	angmax
1	3	0.0	0.05	0.0	150.0	9999.0	9999.0	0.0	0.0	1	-30.0	30.0
1	2	0.0	0.05	0.0	150.0	9999.0	9999.0	0.0	0.0	1	-30.0	30.0
2	3	0.0	0.05	0.0	150.0	9999.0	9999.0	0.0	0.0	1	-30.0	30.0

016B2AE0 0001 0000 0000 0000 2AC8 016B 0000 0000
016B2AF0 0000 0000 0000 3FF8 0000 0000 0000 0000
016B2B00 0000 0000 0000 3FF0 0000 0000 0000 0000
016B2B10 0000 0000 0000 0000 999A 9999 9999 3FA9
016B2B20 0000 0000 0000 0000 FFFF FFFF FFFF C033
016B2B30 0000 0000 0000 3FF0 0000 0000 0000 0000

016C0500 0003 0000 0000 0000 95B8 016B 0000 0000
016C0510 0000 0000 0000 3FF8 0000 0000 0000 0000
016C0520 0000 0000 0000 3FF0 0000 0000 0000 0000
016C0530 0000 0000 0000 0000 999A 9999 9999 3FA9
016C0540 0000 0000 0000 0000 FFFF FFFF FFFF C033
016C0550 0000 0000 0000 3FF0 0000 0000 0000 0000

(c) Powertools memory image of the sensitive parameters.

Figure 3.8: PowerWorld and Powertools controller software attack results as the result of targeted adversary-optimal line rating manipulation.

calculated by the bilevel optimization.

This increase in optimal line rating manipulations is necessary to account for the fact that the AC OPF implementation is constrained by the line rating bounds on apparent power flows (with both real and reactive power components) while the optimal attack generation procedure calculates manipulated line rating assuming that only real power flows are subject to line ratings. As the consequence, the power system enters an unsafe state after the OPF control algorithm uses the corrupted line ratings and hence produces wrong control outputs to the power generators; see Figure 3.8b. Optimal and physics-aware corruption of the sensitive values through a controller attack allows the intruders to maximize the physical impact on the power system operations without having to compromise a large number of sensors as required in false data injection attacks. We also performed the same memory data corruption attack on Powertools pow (2017) package. In this scenario, the attacker changed the line rating for two of the branches as shown in Figure 3.8c. Similar to the PowerWorld case, the exploit locates the sensitive parameters (line ratings) and modifies them during the program execution. As the result, the memory corruption impacted the power flow iterations of DC-OPF performed by the Powertools software that consumed the modified memory regions, and made it converge to a different wrong value. In terms of the attack implementation approach, the attacks against PowerWorld and powertools were identical.

3.7 Discussions and Potential Mitigation

Our attack and similar domain-specific memory data corruption attacks can be mitigated through several potential solutions: *i)* Protection of sensitive data: fine-grained data isolation mechanisms such as hardware supported Intel SGX can be leveraged to store and process sensitive data such as power system parameters within protection enclave regions. This protects sensitive data against access requests by other irrelevant instructions in the same memory space. A more fine-grained version of such memory-based data protection can distinguish between data that are often fixed during the operation (e.g., power system topological information) vs. regularly updated data regions (e.g., sensor measurements) to facilitate lower-overhead protection such as

read-only memory pages for the fixed data once they are loaded on memory initially. *ii)* Control command verification: controller output verification mechanisms such as an extended version of TSV McLaughlin et al. (2014) can be used to ensure the safety of the (maliciously) issued control commands by an infected control system software before they are allowed to reach the actuators. Monitoring of the control channel, however, does not ensure the correct functionality of the control system software. Instead it just ensures its outputs (even though corrupted) are within the safety margins of the physical plant. *iii)* Intrusion-tolerant replication: a more traditional approach is to use redundancy such as N-version programming by maintaining a redundant controller software that is different from the main one used. The replica controller can monitor the dynamic behavior of the physical plant (e.g., power system) as well as the main controller's output to the actuators. The replica can rerun the control algorithm to calculate and compare its calculated control outputs with those of the main controller. Hence, the main controller infection (misbehavior) can be identified if a mismatch is detected; *iv)* Algorithmic redundancy: Carefully designed algorithmic tools (e.g., attack-aware optimal dispatch) can provide safe operating regimes to limit the impact of successful attacks. Indeed, this is a topic of future research.

3.8 Related Work

We review the most related recent work on control system security. The existing solutions to protect the control networks' trusted computing base (TCB) are insufficient as software patches are often applied only months after release Pollet (2010), and new vulnerabilities are discovered on a regular basis Peterson (2012); Szekeres et al. (2013). The traditional perimeter-security tries to keep adversaries out of the protected control system entirely. Attempts include regulatory compliance approaches such as the NERC CIP requirements U.S. Department of Energy Office of Electricity Delivery and Energy Reliability (2015) and access control Formby et al. (2016). Despite the promise of information-security approaches, thirty years of precedence have shown the near impossibility of keeping adversaries out of critical systems Igure et al. (2006) and less than promising results for the prospect of addressing the security problem from the

perimeter Lewis (2006); Kuz'min and Sokolov (2012); Morris et al. (2009). Embedded controller software from most major vendors Kuz'min and Sokolov (2012); Valentine (2013) and popular human machine interfaces Morris et al. (2009) have been shown to have fundamental security flaws. Offline control verification solutions McLaughlin et al. (2014) implement formal methods using symbolic execution of the controller program to verify the safety of the code before it is let execute on the controller device. Not surprisingly, those methods face scalability problem, caused by state-space explosion.

One specific related line of research is proposed false data injection (FDI) attacks Liu et al. (2011); Tan et al. (2016); Wang et al. (2014) that have been explored over the past few years. FDI assumes compromised set of sensors and make them send corrupted measurements to electricity grid control centers to mislead the state estimation procedures. The authors propose a system *observability* Liu et al. (2011) analysis to determine the required minimal subset of compromised sensors to evade the electricity grid's bad data detection algorithms Lu and Zhang (2007). The power system stability has also been studied under corrupted real-time pricing signals Tan et al. (2013). As a fundamental domain-specific monitoring tool for cyber-physical platforms, state estimation is to fit sensor data to a system model and determine the current state Abur and Expósito (2004); Alsac et al. (1998). Existing real-world solutions to analyze power system stability Glover et al. (2011) run every few minutes Singh and Alvarado (1995). These solution do not consider the cyber-side controllers and/or adversarial settings Arrillaga and Smith (1998); Wood and Wollenberg (2012); hence they may miss malicious incidents such as the controller code execution attacks. Risk assessment techniques, e.g., contingency what-if analyses Sun and Overbye (2004) investigate potential power system failures speculatively. However, enumeration of all *possible* incidents is a combinatorial problem and does not scale up efficiently in practical settings Davis and Overbye (2011).

3.9 Concluding Remarks

We presented a domain-specific attack against the popular and widely-used power grid economic dispatch control algorithm. The attack searches the controller's live memory

for sensitive power grid parameters and modifies them maliciously. It replaces the legitimate values with the adversary-optimal values that are calculated considering the physical system dynamics. As validated on real-world controller implementations, the attack maximizes the physical damage to the power system.

Chapter 4

Tell Me More Than Just Assembly! Reversing Semantics of Embedded IoT and Industrial Control Software Binaries

4.1 Introduction

Cyber-physical systems (CPS) interconnect, control and monitor critical environments such as electrical power generation, transmission and distribution, chemical production, oil and gas refining and transport, and water treatment and distribution. In recent years, cyber-physical Internet-of-things (IoT) have received considerable attention due to security concerns originated by the trend to connect those critical platforms to the Internet Network and (ENISA). Critical infrastructures connected to and controlled by CPS substantiate these security concerns. Nevertheless, the number of CPS/IoT devices is projected to reach 20.4 billion in 2020, forming a global market valued $3 trillion TechNavio (2014). Nation-state CPS malware such as Stuxnet Falliere et al. (2010) against Iranian nuclear uranium enrichment facilities and BlackEnergy F-Secure Labs (2016) against the Ukranian train railway and electricity industries show that targeted attacks on critical infrastructures can evade traditional cybersecurity detection and cause catastrophic failures with substantive impact. The discoveries of Duqu Chien et al. (2011) and Havex Rrushi et al. (2015) show that such attacks are not isolated cases as they infected critical infrastructures in more than eight countries. Additionally, IoT devices have been attacked over the years Angrishi (2017). The Mirai botnet Antonakakis et al. (2017), composed primarily of embedded and IoT devices, took the Internet by storm in late 2016 when it overwhelmed several high-profile safety-critical targets with massive distributed denial-of-service (DDoS) attacks.

Control algorithms in cyber-physical IoT platforms act as functional guarantees for

the entire cyber-physical system Cardenas et al. (2008); Kwon et al. (2013); Mo et al. (2010). As indicated by the past attacks Falliere et al. (2010); F-Secure Labs (2016), adversaries are often attracted to vulnerabilities that directly affect the core embedded controller algorithm implementations. For instance, Stuxnet modified the set-points for feed-back control mechanisms within the variable frequency drives controlling the motors for uranium enrichment centrifuges Falliere et al. (2010); Hinkkanen (2013). This was implemented by compromising the programmable logic controllers (PLCs) communicating with the variable frequency drives. BlackEnergy F-Secure Labs (2016) modified the Ukraine's power system parameters in control algorithm implementations to cause a blackout. Similarly, Harvey Garcia and Zonouz (2017) implemented a firmware rootkit in which the malicious control algorithm code issues disruptive feedback-control actuation to the physical system. At a higher level of abstraction, another work Shelar et al. (2017) showed how the optimization control algorithm parameters in energy management systems can be exploited to dispatch malicious control commands to damage the electric power grid.

The recent exponential growth of major cyber-physical IoT attacks indicate the insufficiency of existing security analysis solutions to protect controller software in aforementioned cyber-physical platforms. A common feature in most of the past attacks has been the adversaries' focus on affecting the controller software behavior. As a result, the control algorithm implementations within the critical controller software binaries, e.g., the proportional-integral-derivative (PID) controller in drone firmware, have become the main battlefield for cyber-physical security. Attackers try to hijack the control flow and/or corrupt sensitive parameters within those algorithms to make the controller issue malicious actuation commands and cause large-scale physical damage.

On the protection side, however, the state-of-the-art reverse engineering and vulnerability assessment tools are unable to extract and leverage precise, domain-specific of low-level embedded binary modules. Therefore, they fail to reason about the impact of a particular vulnerability to the overall system, e.g., the drone's flight operation. Choi, *et al.* used system identification of a drone to generate *control invariants* based

on a control general control system template Choi et al. (2018). They then instrumented the controller binary to incorporate monitors at the end of the control loop to ensure the system is abiding to these invariants. However, this was a coarse-grained approach could only detect extreme deviations from the expected behavior and cannot pinpoint which areas of the binary need to be protected. A prerequisite for developing effective protection solutions would be a rich understanding of the low-level controller software segments that implement the core theoretical control algorithms. Such knowledge about the high-level algorithmic semantics of the controller software binary could then be used to identify and protect sensitive code and data segments of an algorithm in the binary, e.g., using CFI or Intel SGX. Development of the protection mechanisms or attacks based on the discovered control algorithm details in the software binaries is outside the scope of this paper. Our objective is to extract the high-level algorithmic semantic knowledge from closed-source (commercial) controller software binaries in cyber-physical IoT platforms.

Access to source code is often unavailable for third-party security analysis of commercial off-the-shelf (COTS) embedded software in cyber-physical control domains. This remarkably limits the use of existing source code-based solutions cla; Cadar et al. (2008b); dat. For binary executables, automatic reverse engineering of their semantics provides intuitions about the program's functionality and expedites security analysis with respect to discovering commonly known software bugs and vulnerabilities. Advanced disassembler and debugger tools such as IDA Pro Rescue (2006) and OllyDbg Yuschuk (2007) offer a variety of techniques to help elevate low-level machine codes to more abstract representations (e.g., assembly instructions), increasing the readability of a program for a user. For instance, such tools can identify any known library functions in the disassembled program and translate such function calls to their corresponding descriptive symbolic names. However, such static binary analysis tools mainly extract syntactical information and are not guaranteed to preserve/extract the semantics of the original program Schwartz et al. (2013).

Contributions. We present MISMO, a reverse engineering framework to extract algorithm-level semantics from stripped embedded software binary implementations of IoT and

cyber-physical control algorithms. MISMO utilizes dynamic binary analysis and comparison of mathematical expressions to recover a particular algorithm implementation's . MISMO performs dynamic binary analysis to locate the target subroutines of the executable that implements the control algorithm. The arithmetic operations of the execution paths are analyzed symbolically to build a binary-level abstract syntax tree (AST) for the corresponding output values. The generated AST subtrees are recursively compared to and matched with the algorithm-level AST subtrees of the control theoretic expressions. Consequently, our solution fills the semantic gap between the low-level binary executables and high-level algorithmic descriptions with regards to control and data flows. MISMO's ultimate output is to provide the security analysts with domain-specific information about the IoT/CPS binary by annotating individual disassembled instructions and memory addresses with the corresponding algorithm-level operation and mathematical parameters, respectively.

Our contributions are summarized as follows:

- We propose a domain-specific reverse engineering solution to extract high-level algorithmic control- and data-flow semantics from embedded binary executables in various cyber-physical IoT control applications.

- We introduce a semantic mapping using dynamic binary analysis and symbolic comparison of the mathematical and binary expressions to fill the semantic gap between high-level algorithm descriptions and low-level stripped binary segments.

- We implemented the proposed framework (MISMO) as an IDA Pro plug-in and evaluated it on 2,263 commercial embedded firmware from 6 various cyber-physical application domains. The plug-in transfers the collected semantics to enrich the disassembled code and data segments in order to expedite the reverse engineering process. We validated MISMO for various use-cases, and discovered a previously unknown Linux kernel bug that exists in all kernel versions since 3.13.

We evaluated MISMO on a wide-ranging set of real-world applications including drones, self-driving cars, smart homes, robotics, 3D printers, as well as the Linux kernel.

Potential use-cases. MISMO provides an enhanced reverse engineering solution with more informative extracted semantics about the IoT firmware executables. Hence, its use-cases include well-known software security analysis scenarios that involve binary reverse engineering. For instance, the extracted algorithm-level semantics of the executable code and data segments by MISMO can be used for *i)* binary vulnerability assessment (e.g., to determine whether any of the important control algorithm parameters in memory - identified by MISMO- can be corrupted by a buffer-overflow exploit as shown in Kim et al. (2018) against a drone firmware); *ii)* memory forensics analysis (e.g., black-box analysis after a plane crash - to leverage the reverse-engineered binary semantics by MISMO to discover controller state information such as sensor/actuation and detailed parameter values from the crash-time dumped controller memory files similar to Saltaformaggio et al. (2014b)); *iii)* sensitive code and data segment protection (e.g., to protect sensitive memory areas - identified by MISMO- where important control algorithm logic and parameters reside to prevent targeted attacks against controllers Garcia and Zonouz (2017). The protection can be possibly deployed via software-based encryption, dynamic memory value vetting or hardware-assisted solutions such as Intel SGX).

iv) correct algorithm implementation verification (e.g., to determine whether the controller firmware binary indeed implements the target control algorithm correctly - possible mismatches - bugs - can potentially enable attackers to drive the controller into unsafe states Starbuck and Farjoun (2009)); and *v)* binary-level software similarity measures (e.g., to detect possibly unauthorized reuse of commercial binary implementation of control algorithms that are protected by intellectual property regulations. The information extracted by MISMO about each executable would help for a better semantic comparison of binaries as opposed to purely-syntactical comparisons Xu et al. (2017)).

It is noteworthy that our focus in this paper is mainly to propose a solution (MISMO) that *extracts* knowledge from a given firmware binary file. In this paper, we explain MISMO's design and implementation in details, and evaluate its performance by comparing its outcomes (extracted semantics) on real-world firmware binaries with the ground

truth. However, individual development and demonstration of the aforementioned use-cases and how Mismo's outcomes can be leveraged for each one of them (e.g., memory dump forensics analysis using Mismo's outputs) involve research challenges that remain outside the scope of this paper.

The remainder of the paper is structured as follows. We provide an overview of the Mismo's threat model and architecture in Section 4.3. In Section 4.4, we discuss the design details for each component of the Mismo framework. We discuss the implementation of Mismo in Section 4.5 on a wide-range of applications. We present the evaluation of Mismo on multiple case studies in Section 4.6 followed by a review of related reverse engineering works in Section 4.7. We conclude the paper in Section 4.8.

4.2 Threat Model

In our threat model, the binary executable is not assumed to be malicious. It may, however, include vulnerabilities. We utilize Mismo in the context of IoT and industrial control applications to reverse engineer control algorithm implementations. We leverage the fact that IoT and ICS embedded control software developers rarely design a new theoretical control algorithm from scratch (see Section 4.6 for our empirical validation on 2,263 commercial binaries). Instead, they almost always pick and implement one (or more) out of a set of commonly used and known control algorithms that have been extensively analyzed theoretically and for practical deployments.

As examples, proportional-integral-derivate (PID) controllers are used in programmable logic controller (PLC) programs Sangeetha et al. (2012), Kalman-Filters for guidance, navigation and control of drones Gowda et al. (2016), and Pulse-width modulation (PWM) is used for robotics and 3D printer extruder motor control Raj et al. (2017). Hence, we assume that Mismo has access to a predefined set of popular control algorithms widely used in embedded cyber-physical systems (CPS) applications. This set is used to reverse engineer the semantics of a given stripped embedded binary executable. This is intuitively similar to existing signature databases used by disassemblers for library API identification, e.g., IDA Pro's FLIRT technology Eagle (2011).

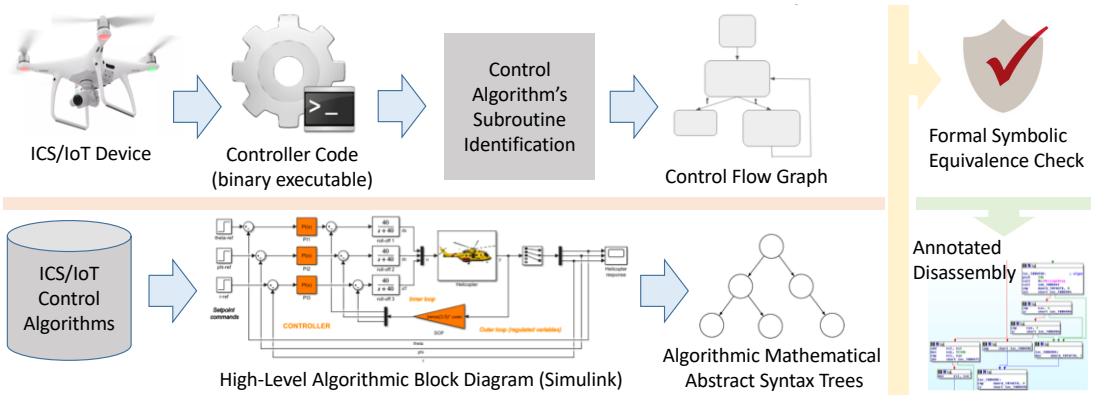


Figure 4.1: Overview of MISMO framework.

4.3 System Overview

Recent years have witnessed the rapid progress of IoT and embedded control systems technologies, with many of them already seeing wide adoption. The booming of IoT and embedded control ecosystem inevitably attracts cyber criminals, who aim at compromising IoT/ICS devices. The loose protection of these devices and pervasiveness of vulnerabilities Constantin (2016); Emm et al. (2015); Chen et al. (2018); Muench et al. (2018) in them actually present to the miscreants low-hanging fruits.

Towards securing these devices, reverse engineering of the semantics of the target embedded binaries is a crucial step in their vulnerability assessment and patching. MISMO focuses on reversing COTS embedded IoT/ICS stripped binaries, i.e., executables without any relocation information or symbols–except those necessary for dynamic linking.

The IoT and ICS devices execute embedded controller binaries in periodic *scan cycles*. In each scan cycle, the firmware processes the input values that consist of measurements from the sensors in the environment, e.g., accelerometer readings on a drone. The firmware implements a control algorithm and determines the next control commands based on the most recent sensor readings. It sends the calculated output values to the actuators that manipulate the environment, e.g., rotation speed of the motors on a drone. Then, the next scan cycle starts by reading the updated sensor measurements going through identical aforementioned procedures.

Subsequent scan cycles implement the control algorithms and are deployed as loops within the firmware binaries. The recent IoT/ICS attacks have shown that the control algorithm implementation submodules of the firmware binaries are often the most attractive target for adversaries Falliere et al. (2010); F-Secure Labs (2016), because their infection result in system-wide damages, e.g., a drone crash.

According to our experimentation with 2,263 commercial embedded firmware binaries, the embedded control algorithm submodules are often not branch-heavy, and hence can be analyzed using exhaustive static/dynamic analysis techniques. Consequently, the control algorithm is implemented using only a few arithmetic-heavy execution paths (deterministic algorithm logic). The execution paths across subsequent scan cycles that go through a predictable pattern covering all the paths during a short operation of the system, e.g., a short few seconds of drone flight. However, input data values from sensors differ significantly over time. Furthermore, in our experimentation with the control algorithm submodules, we never faced indirect call-sites, and were able to leverage existing tools to extract the binaries' syntactical information such as functions and control flow graphs.

Figure 5.1 shows Mismo's architecture. To reverse engineer the semantics of a given binary executable for a particular controller, Mismo tries to match the binary to each control algorithm in its database of popular commonly-used embedded control algorithms. Mismo finally picks the algorithm with the highest matching score and uses its description to annotate the binary's disassembled code/data regions with high-level algorithmic semantics and help the analysts to better understand the binary's functionalities.

Mismo starts with the control algorithm's high-level description flowchart. The high-level description can be either in pseudo-code or block-diagram format, e.g., a MATLAB Simulink diagram. Each possible operation flow of the algorithm is exercised parametrically. As a result, the mathematical expressions of the output values as a function of input parameters are calculated.

Mismo then analyzes the given stripped binary dynamically through inspection of its execution traces when used in practice, e.g., an execution trace of the firmware

controller during the drone's flight. Through examining the memory values and arithmetic instructions, the part of the execution trace (i.e., responsible subroutines) that implements the control algorithm is identified for further analysis.

MISMO implements a symbolic semantic-matching algorithm to map the high-level operations from the algorithmic flowcharts to low-level instructions and memory addresses in the executable's code and data segments. This mapping provides semantic meaning for each assembly instruction, e.g., the impact of each instruction with respect to the overall system as well as the critical memory associated with the control algorithm. Such semantic information can enable a finer-grained white-box analysis of the binary.

The first step of MISMO entails locating the function in the firmware that corresponds to the core controller algorithm implementation. It uses dynamic data flow analysis to locate such a function. The second step is traditional disassembly of the function and the recovery of its control flow graph (CFG). The framework analyzes the CFG given the control algorithm's high-level flowchart, and ranks the CFG's control flows based on how similar they "look" to the algorithm's operations. The third step involves the symbolic expression generation for the selected control-flow path's output variables (actuation commands) via symbolic execution. The fourth step compares the generated binary-level symbolic expression to the abstract syntax tree (AST) of the associated high-level algorithm operations. The final step is the semantic refinement of the previous results. The previous steps may not resolve all of the mappings between binary- and algorithm-level operations. MISMO uses a satisfiability modulo theories (SMT) solver to improve the ultimate mapping accuracy. The acquired semantic information will then be used to annotate the binary's disassembled code/data segments for more informed binary analyses.

4.4 Mismo Design

In this section, we will detail the aforementioned sequential components of the MISMO framework.

4.4.1 Locating the controller subroutines

A high-level control algorithm's expression cannot be directly compared to a particular embedded firmware implementation in assembly format automatically as there will most likely be a high level of variability in both the algorithm expression as well as the implementation. To start its analysis, Mismo first locates the binary subroutines that implement the core control algorithm. A stripped COTS binary for an embedded controller may include thousands of functions. Locating a particular algorithm function implementation without debugging information is not trivial.

Mismo utilizes dynamic data flow tracking within individual scan cycles to locate the control algorithm functions. Although Mismo does not have access to debugging information, domain-specific knowledge can provide some hints about input arguments that are directly related to the target control algorithm. Specifically, we mark the input variables that are populated by the sensor measurements as the data taint sources. The sinks are the output variables whose values are sent out to the actuators (i.e., the output module on the embedded controller device).

Assuming that the inputs to the embedded firmware binary are known, dynamic taint analysis can be used to establish which registers and areas of memory related to the program are affected by each sensor input value. Given the execution traces of the firmware, we also have access to all functions calls during each scan cycle. Mismo keeps track of the tainted data propagation from source to sink during the embedded controller execution. Out of the whole trace, Mismo identifies the subroutine(s) that implements the control algorithm and performs value-changing arithmetic (floating point) operations on the tainted data. As the result, irrelevant functions will be excluded from further analysis. For instance, Mismo ignores a function call that does not perform any arithmetic operations (value modifications) and only moves data around in the memory such as fetching data from the controller's sensor/actuation GPIO ports. Such a function may be required for low-level execution of the software, but does not contribute to high-level control algorithmic parameter value manipulation operations.

4.4.2 Control flow graph refinement

Once Mismo has established a candidate set of functions for the control algorithm, it focuses the following analyses on the identified few functions only. Mismo performs symbolic execution of the function binary code to generate concrete input test cases for each feasible execution path. However, generating test cases for all paths for each candidate function may be infeasible due to the well-known scalability issues with symbolic execution. Furthermore, some paths may not be relevant to the core of the associated control algorithm. For instance, in our experiments with commercial firmware samples, we observed a large portion of paths perform input validation or exception handling. These paths include almost no value-changing arithmetic instructions, which are used heavily by the relevant paths that implement the core control algorithm. We utilize the aforementioned feature (arithmetic operation density) to distinguish and prune such irrelevant paths.

To further narrow down symbolic execution's search space to relevant execution paths only, we obtain the control flow graphs of the identified candidate functions. Mismo performs static analysis of the CFG's execution paths and measures how likely each particular CFG path represents a control flow (sequence of operations) within the high-level control algorithm description. This similarity checking has to consider the semantic gap between the low-level binary execution paths and the high-level control algorithm descriptions. Mismo does so by utilizing the density and types of arithmetic operations as the core metrics. The ultimate identified set of relevant execution paths are represented as a smaller CFG, where the set of vertices and edges is a subset of those in the original function CFG.

Intuitively, the number of branches in the high-level control algorithm flowchart (also referred to as CFG in this chapter) has to be equal to the number of branches in the low-level function binary's pruned CFG. In reality, however, a CFG of a control algorithm implementation typically has a greater number of branches than the high-level CFG (flowchart) of the associated algorithm expression due to modifications unique to the environment in which the algorithm was implemented. For instance, there may

be more comparison instructions in the binary if there are limitations on the values or different variable assignments for different contexts. That being said, if the two CFG's do indeed have the same number of branches, it will be an indication of high similarity. However, a comparison solely based on the number of branches is clearly not sufficient in most cases.

MISMO leverages domain-specific features particular to embedded IoT/ICS firmware implementations. During the design process, we noticed that control algorithm implementations typically use floating point operations as the sensor readings are noisy numeric values. Additionally, the implementations often have a significant amount of error checking on the input parameter values without performing arithmetic value-changing operations. The common comparison instructions include '`VCMP.F32`', '`VCMP.F64`', '`VCMPE.F32`', and '`VCMPE.F64`'. This leads to several CFG paths that bypass the algorithm's core set of instructions and directly jump to the final returning basic block. MISMO uses a lower bound threshold of the number of arithmetic instructions that are required to implement the control algorithm. Any CFG path that contains a smaller number of arithmetic operations is excluded from MISMO's following analyses.

The aforementioned threshold varies significantly for different control algorithms based on their size and level of complexity. We calculate the threshold value for each control algorithm by generating a parametric expression of the control algorithm's output based on its inputs. As a trivial example, if the control algorithm involves calculating a weighted average of the two sensor readings s_1 and s_2, the calculated parametric expression would be $\frac{2*s_1+s_2}{3}$. However, an algorithm may calculate the output as $\frac{s_1+s_1+s_2}{3}$, i.e., without multiplication. Hence, we use a SMT solver to simplify all arithmetic operations down to a canonical minimum-sized form, i.e., $\frac{2*s_1+s_2}{3}$ in the example above. The threshold is calculated as the number of the arithmetic operations in the simplified expression. Using the threshold, MISMO guarantees that a CFG path that bypasses the core set of arithmetic operations will not be considered.

4.4.3 Symbolic controller abstraction

Once the candidate function CFG is pruned, we can perform more exhaustive analyses on the resulting (much smaller) refined CFG. Mismo implements symbolic execution to compute symbolic expressions for the function's output variables that would be sent to the actuators during the firmware normal execution. Typically, the output values calculated by the control algorithm function are written to the controller's actuator-connected (often GPIO) ports by other subroutines in the firmware.

Mismo needs to first locate the control algorithm function's output variable to calculate and report its symbolic value. This is needed, because we use the symbolic value of the controller function output variable to compare and match with the high-level control algorithm flowchart's parametric output. The outcomes of this analysis will enable us to map low-level controller function code/data segments to high-level control algorithm logic/parameters (discussed later in Section 4.4.4).

Identifying the relevant output variables of the controller function is not straightforward as a function may return the output value in a variety of ways, e.g., the output variable may be a reference parameter, a global variable, or a value that is directly returned by the function. We utilize Mismo's dynamic data flow analysis (Section 4.4.1) of the controller firmware's execution trace to identify the controller function's output variable. Specifically, Mismo focuses on the controller algorithm function execution trace. It determines the memory address or register that stores the function's calculated values right after the last value-changing arithmetic instruction. Mismo marks the determined memory address or register as the function's output.

Mismo's objective is to perform symbolic execution of the refined function CFG and calculate the symbolic value of the output variable. The value should represent the symbolic value of the actuation command (output) based on the symbolic sensor readings (inputs). Mismo needs to identify relevant inputs to the controller function. This is needed to enable symbolic execution, which requires to identify a set of input variables to be associated with symbols.

Trivial solutions such as marking the function arguments as inputs would not work

in IoT/ICS binaries for the following three reasons. First, the controller functions sometimes leverage other sources as inputs such as global variables. Second, some of the function inputs, even though necessary for the binary's execution, are not relevant to the core control algorithm. Hence, our objective is to identify only the input variables that affect the values of our identified output variables. Third, function arguments may be data types or structures that do not have a fixed size, which is needed for scalable generation of symbolic values. Finally, controller function arguments often leverage pointers to sensor reading data blocks (implemented as structures or objects in the firmware) that cannot be marked as symbolic input directly for symbolic execution; instead the corresponding pointed memory regions should be labeled symbolic.

To identify the relevant inputs to the controller function, we use a slightly modified backward slicing analysis. In our experiments, conventional backward dynamic taint analysis led to over-tainting and was not helpful for accurate identification of the function inputs. More specifically, dynamic backward taint analysis of the binary often resulted in many additional unnecessary memory variables, as tainted, that were irrelevant to the core mathematical embedded ICS/IoT control algorithm, e.g., a file descriptor pointer used for event logging.

We want to identify the the relevant numeric inputs to the controller function that correspond to the input parameters in the control algorithm's high-level mathematical flowchart. Mismo uses a slightly modified backward slicing that exploits a domain-specific fact the IoT/ICS control algorithms mainly deal with numerical sensor inputs and variables (e.g., double or float data types as opposed to strings and characters). Mismo's backward analysis considers memory taints coupled with their data values. Starting from the output variable, Mismo discovers all associated inputs that contribute to the output value numerically through arithmetic instructions, e.g., `VADD.F64` and `VMUL.F64`. In other words, the inputs whose values have been used arithmetically to calculate the output values are selected only and marked as symbolic. The remaining controller function inputs are represented with the concrete values in the Mismo's symbolic execution of the function.

Mismo executes the controller function symbolically, and calculates the symbolic

expressions of the identified controller function output variables. By design, all the non-concrete entries of the expression originate from sensor readings and the control algorithm parameters.

The calculated symbolic expression may 'look' different across different implementations (IoT devices) of the same theoretical control algorithm (e.g., implemented as $\texttt{cntrlr}(i)\{\texttt{return } 2 \times i;\}$ or without multiplication as $\texttt{cntrlr}(i)\{\texttt{return } i + i;\}$). Mismo utilizes SMT solvers to simplify and turn the calculated symbolic expression into its minimal and unique canonical form that is mathematically equivalent to the original expression. This canonical symbolic expression is essentially the arithmetic summary of the controller function in the IoT/ICS device firmware. The next subsection will use this expression to map its operations with the control algorithm's high-level flowchart logic.

4.4.4 Abstract syntax tree mapping

To map the low-level firmware binary's code/data segments with the high-level control algorithm semantics, we compare and match their abstract syntax tree (AST) representations. Mismo computes the AST representations of the high-level control algorithm (referred to as the *high-level AST*) as well as the aforementioned canonical symbolic expression of the controller function (referred to as the *low-level AST*).

Mismo's objective is to map the individual nodes of the low-level AST to their counterparts in the high-level AST. Mismo accomplishes this in two steps. During the first step (which is more lightweight), it tries to map the nodes/subtrees of two ASTs based on the tree structure and using a recursive graph-theoretic isomorphism check. Consequently, some nodes from the low-level AST may each be mapped to more than one node in the high-level AST. To resolve such cases, Mismo utilizes formal satisfiability checking to precisely find the unique maps via comparing the subtree contents and their arithmetic representations from the two ASTs.

Mismo first checks whether the two AST roots have the same arithmetic opcode. If the root opcode is the same, the roots' degree (the number of children nodes) is compared. If both have the same degree, every child pair from the two ASTs is investigated.

For each given pair of children nodes, the following cases may arise: *i)* if they store the same arithmetic opcodes, their corresponding subtrees are compared recursively; *ii)* if they store different opcodes, their subtrees are not investigated further; *iii)* if one node stores an arithmetic opcode, while the other node stores a symbol or a concrete value, the case is discarded; *iv)* if they store equal concrete values, their subtrees are compared recursively; *v)* if they store different concrete values, the case is discarded; *vi)* if they store two symbols, or a symbol and concrete value, the subtrees are compared recursively.

The aforementioned comparison mainly exploits the structure of the AST trees and compares the ASTs based on individual node contents. If the recursive procedure above can uniquely map the two ASTs, MISMO reports the node-node mappings. Otherwise, it performs a more in-depth analysis to resolve the one-to-many node mappings. The analysis uses formal SMT satisfiability checks and compares ASTs based on the whole subtree contents recursively. We will see an example of the one-to-many mapping in Section 4.5.

More specifically, to resolve the potential non-unique mappings, we perform a formal symbolic equivalence check between subtrees of the two ASTs. Given the two ASTs for high-level algorithm expression ($\texttt{AST}_{\texttt{high-level}}$) and low-level binary symbolic output expression ($\texttt{AST}_{\texttt{low-level}}$), we construct a conjunctive logical predicate for each candidate mapping. The predicate is in the form $P \coloneqq [(\texttt{AST}_{\texttt{high-level}} = \texttt{AST}_{\texttt{low-level}}) \wedge (\texttt{AST}^1_{\texttt{high-level}} = \texttt{AST}^1_{\texttt{low-level}}) \wedge \cdots \wedge (\texttt{AST}^n_{\texttt{high-level}} = \texttt{AST}^n_{\texttt{low-level}})]$, in the form, where $\texttt{AST}^i_{\texttt{high-level}}$ and $\texttt{AST}^i_{\texttt{low-level}}$ represent subtrees of the two abstract syntax trees for high-level algorithm and binary implementation, respectively. If the predicate is proved to be infeasible by the SMT solver, MISMO rejects the mapping and investigates the remaining candidate mappings until the correct one is identified.

4.5 Implementation and Case-Study

This section describes the implementation details of our prototype solution. MISMO utilizes the QEMU emulator Bellard (2005) and S2E Chipounov et al. (2011) for its

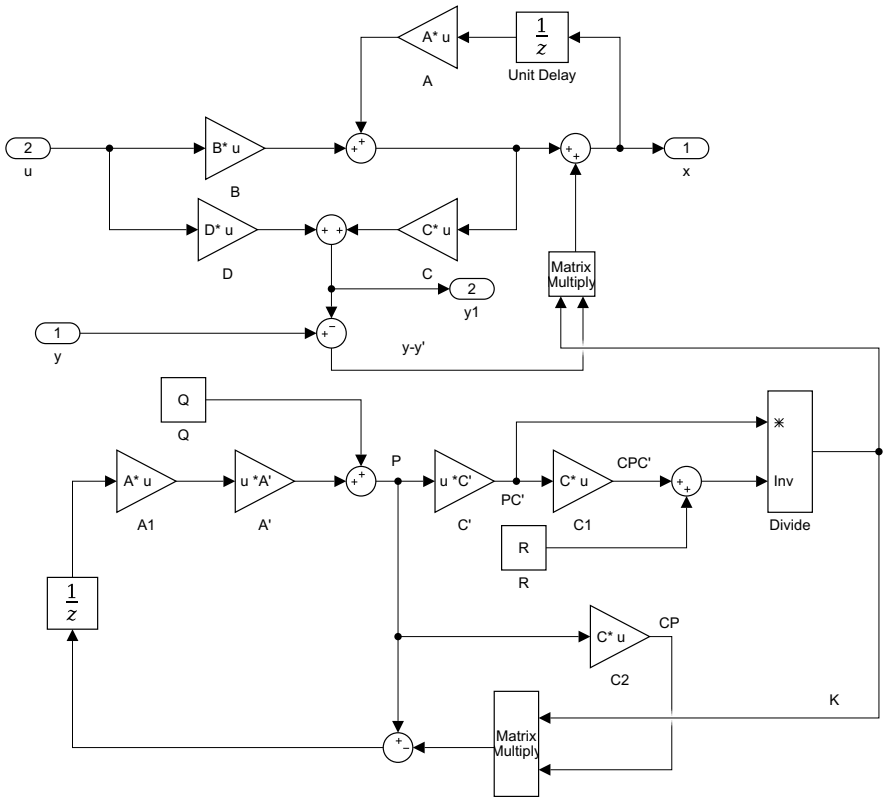


Figure 4.2: High-level block diagram of a sample embedded CPS control algorithm (Kalman filter). MISMO will map algorithmic logic and parameters of the diagram to their corresponding binary-level control flows and memory variables, respectively.

dynamic analysis and symbolic execution procedures, respectively. The ultimate prototype is implemented as an IDA Pro plugin that integrates MISMO's results (extracted high-level algorithmic semantics) into the IDA Pro user interface.

To facilitate the understanding of our implementation details, we will provide an ongoing simple application on a popular widely-used algorithm in industrial control: the Kalman filter Chen et al. (2017). The Kalman filter is an optimal estimation algorithm commonly used in embedded control systems to estimate unknown variables of various physical processes kal. For instance, drones typically use Kalman filters for the runtime location state estimation based on the sensor measurements. For this case-study, we used the stripped binary executable of the commercial 3DR Solo drone controller that utilizes a Kalman filter. See Figure 4.2 for a high-level Simulink Simulink (2005) block diagram of the algorithm. The first step of the algorithmic semantic-matching reverse engineering process will be to analyze the binary file in order to determine which

subroutine corresponds to the concrete implementation of the Kalman filter. MISMO performs dynamic taint analysis to determine the number of arithmetic instructions that are used by each subroutine in the binary program. The results are presented in Figure 4.3. There are three functions (`sub_10288`, `sub_94F8`, and `sub_8EC4`) that have been tainted by MISMO's dynamic analysis of the firmware execution paths. Therefore, these three functions will be considered as our candidate functions for the implementation of the control algorithm.

We further refine the candidates using IDA Pro's binary call-site analysis. The analysis shows that the subroutines `sub_10288` and `sub_94F8` both call the subroutine `sub_8EC4`. If the three functions have the same number of arithmetic/logic instructions, MISMO will prune the calling functions. Therefore, MISMO considers `sub_8EC4` as the main implementation of the Kalman filter algorithm. MISMO assumes that there is only one implementation for a particular algorithm for an execution path. This assumption will be reinforced in Section 4.6.

If we had chosen to use traditional dynamic taint analysis–without integrating the domain knowledge with respect to the arithmetic instructions–17 functions would have been identified as candidate implementations for the Kalman filter algorithm. Even after removing the repeated function calls from the candidate set and using a similar binary call-site analysis, the analysis would still result in 6 candidate functions for the Kalman filter implementation. Although this is a simple example, this demonstrates the usefulness of MISMO's taint analysis in the context of embedded control system algorithms.

As mentioned previously, even though we have identified the candidate function for the Kalman filter algorithm implementation, we still cannot directly compare the associated assembly code to the high-level algorithm expressions. We first need to generate the symbolic expression for the function implementation in order to compare it to the associated symbolic AST of the high-level embedded CPS control algorithm. The first step in the generating the symbolic expression is to choose a single candidate execution path in the function's CFG. MISMO obtains the binary program's CFG using a Python plugin script for IDA Pro. We then implement the aforementioned CFG

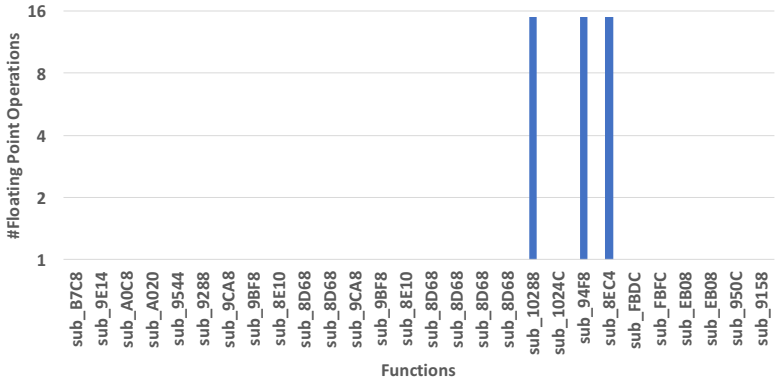


Figure 4.3: The (correctly) identified controller functions within the popular Kalman filter implementation. The remaining subroutines (with zero value-changing floating point operations) are not considered for later analyses.

selector to choose the most fruitful control flow path based on the lower bound of the number of the algorithm's arithmetic operations.

Figure 4.4 provides an example of a selected path. The blocks with gray background compose one control flow path, which will be used for further analysis. In the first branch, Mismo selects the `true` branch as the `false` branch overwrites the value `[R7]`, which is the location where the the final arithmetic result is stored and compared with zero. Put in other words, the selection of the false branch would result in a path where the number of the arithmetic operations fall below those of the high-level control algorithm. Once Mismo selects one control flow path, it will use dynamic symbolic execution to generate the final symbolic expression for the algorithm. To identify the output value of the function implementation, Mismo needs to locate the memory addresses or registers associated with the program that stores the output value of the function. To that end, Mismo first identifies the final arithmetic instruction to be the final instruction that updates the function's output value.

Starting from the output values that are sent to the actuators by the firmware, Mismo traverses the its execution trace backward, and locates the last arithmetic instruction within the candidate control algorithm subroutine that produces the target output value. Figure 4.6 shows the last set of instructions as well as the analysis of the candidate subroutine's instructions, including the last arithmetic operation. Mismo marks the first memory address or register after the identified arithmetic instruction

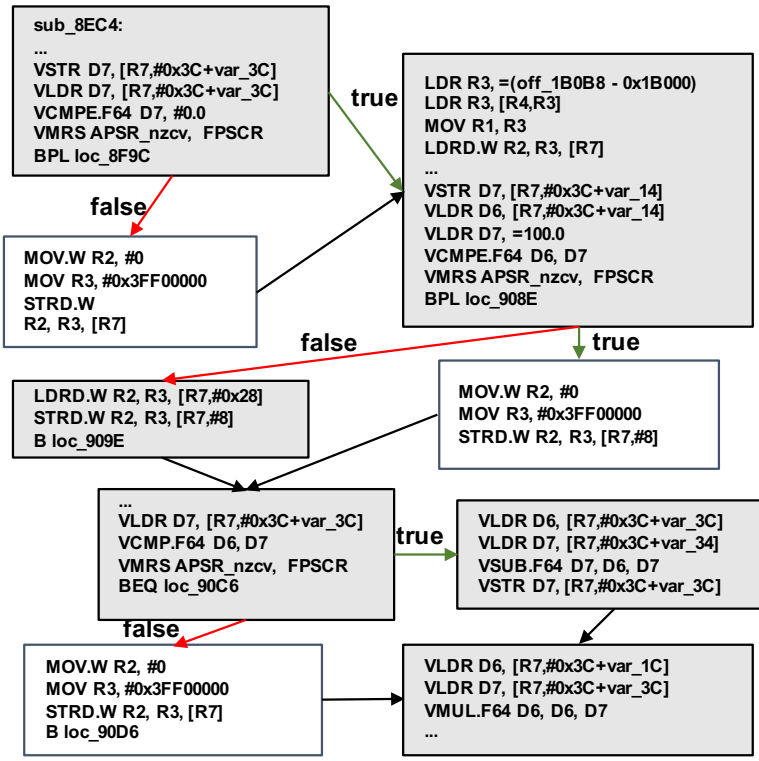
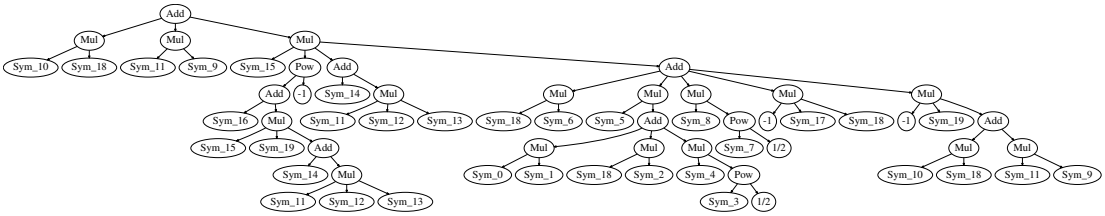
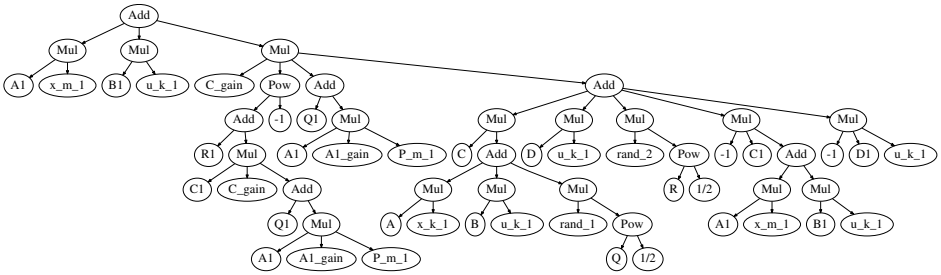


Figure 4.4: The CFG of function sub_8EC4 and the top candidate of paths selected by Mismo selector.



(a) Abstract syntax tree for the symbolic output value of the binary executable subroutine (w/o semantics, e.g., parameter names).



(b) Abstract syntax tree for the symbolic output value of the Kalman filter mathematical algorithm (w/ semantics).

Figure 4.5: Mapping executable- to algorithm-derived ASTs to enrich the disassembled code/data views with semantic information.

Symbolic	Sym_0	Sym_1	Sym_12	Sym_13
A	✓	✓		
x_k_1	✓	✓		
A1_gain			✓	✓
P_m_1			✓	✓

Table 4.1: Result of Comparing Two ASTs (from algorithm-level block diagram and implemented binary executable)

that stores the output value as the candidate output variable.

For our example, the output value of the binary program was `6.162498634045976`. For the instruction `0x90E6: VADD.F64 D7, D6, D7`, register `D7` holds the value `0x4018a6660abb7d12`, which is corresponding to the aforementioned output result `6.162498634045976`. MISMO concludes that the instruction `0x90E6: VADD.F64 D7, D6, D7` is the last instruction to generate the final output value and register `D7` holds the final output result. MISMO can now generate the symbolic expression for register `D7` when executing the instruction `0x90E6: VADD.F64 D7, D6, D7`.

We already know that the register `D7` will hold the final symbolic output. To generate the symbolic expression for the entire algorithm implementation, MISMO needs to also symbolize the the associated input values. It is not sufficient–or even necessarily correct–to symbolize just the function's parameters. Therefore, we implemented backward slicing to backtrace all related input variables. In the case of the Kalman filter implementation, the symbolic inputs are generated from global variables since there are no function parameters.

MISMO found four pointers with different offsets that are used as inputs to the function. For a more accurate location of the associated input values, we performed dynamic backward slicing to determine all relevant inputs for the isolated control flow path specifically. For the Kalman filter application, we found 20 relevant input variables that are all offset values from four pointers: `unk_1B258`, `unk_1B5E8`, `unk_1B640` and `unk_1B620`.

Once the output value and the associated input variables are located and marked as symbolic, MISMO uses symbolic execution to generate the symbolic output expression for the algorithm implementation. The symbolic expression for the Kalman filter candidate function is as follows: ((((`Sym_15` $*$ (((`Sym_12` $*$ `Sym_11`) $*$ `Sym_13`) $+$

Sym_14))/((Sym_19∗(Sym_15∗(((Sym_12∗Sym_11)∗Sym_13)+Sym_14)))+Sym_16))∗ (((√(Sym_7) ∗ Sym_8) + ((Sym_5 ∗ ((√(*Sym*_3) ∗ Sym_4) + ((Sym_0 ∗ Sym_1) + (Sym_18 ∗ Sym_2)))) + (Sym_18 ∗ Sym_6))) − ((Sym_19 ∗ ((Sym_18 ∗ Sym_10) + (Sym_11 ∗ Sym_9))) + (Sym_17 ∗ Sym_18)))) + ((Sym_18 ∗ Sym_10) + (Sym_11 ∗ Sym_9))).

Before comparing the symbolic expression of the low-level binary implementation with the associated algorithm's symbolic expression, MISMO will further simplify the symbolic expression using the Z3 SMT solver Subramanian et al. (2017). Once Z3 has simplified the symbolic expression, MISMO then uses SymPy Certik et al. (2008) to generate the AST for the implementation's symbolic expression as well as the associated algorithm. Figure 4.5 shows the symbolic expression AST for the binary's candidate subroutine as well as the AST for the associated high-level control algorithm expression that was generated using the block diagram of the Kalman filter.

Once both ASTs are generated, MISMO compares the two ASTs and determines if there is a mapping between the variables of the implementation's symbolic expression and the algorithm's symbolic expression. Table 4.1 shows the results of comparing both ASTs of the Kalman filter. Out of the 20 candidate symbolic input variables, MISMO's initial filters cannot conclude a one-to-one mapping for 4 of them. The symbols `Sym_0`, `Sym_1`, `SymVar_12`, and `SymVar_13` have two candidates each. So, further refinement is necessary to recover the semantic algorithm-level meaning for these symbolic variables.

We use Z3 satisfiability checking to improve the one-to-one node mapping between the ASTs. For the drone's Kalman filter subroutine, MISMO was able to correctly narrow down the candidates of the symbols to their exact corresponding mappings. MISMO finally ensures a one-to-one mapping between symbolic variables of the binary executable and semantic variables (mathematical parameters of the algorithm). It then uses static analysis to propagate the symbolic variable's semantic meaning throughout the binary code and data segments using an IDA Pro plugin. MISMO adds the discovered semantic information to the IDA Pro view of the subroutine using our custom plugin. This allows an analyst to incorporate MISMO's analysis into the binary's

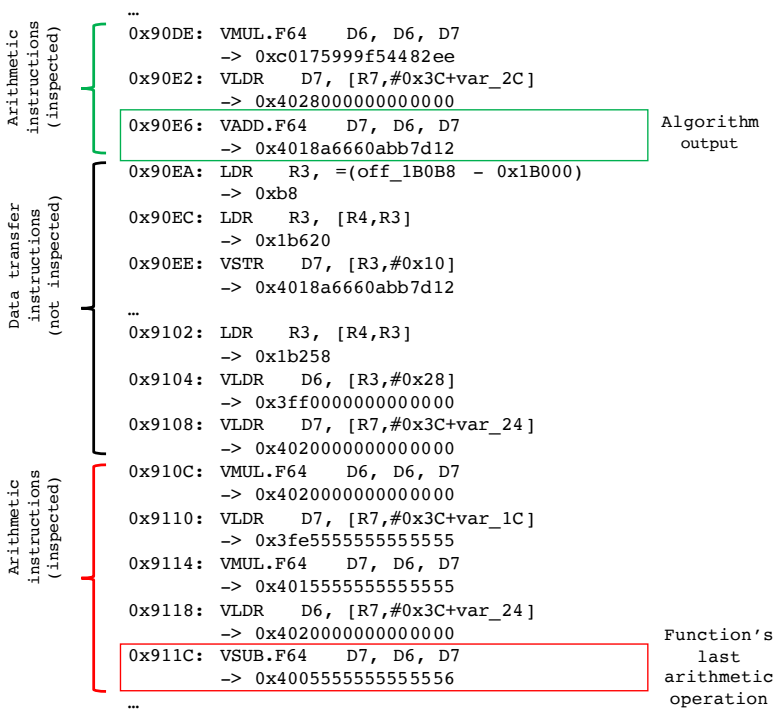


Figure 4.6: Last set of instructions, including the last arithmetic instructions, for the Kalman filter candidate function as well as the associated data flow analysis.

reverse engineering project and can display critical information that is relevant for security analyses and applications, e.g., fine-grained sensitive data protection.

Figure 4.7 shows the IDA Pro view of the Kalman filter binary annotated with the extracted semantic information. The left side of the figure shows the standard IDA Pro view, while the right section displays the extracted semantic information. This expedites the reverse engineering process by providing semantic meaning to every instruction, register, and area of memory of the program. Furthermore, we can develop intuitions about the associated global information of the program. For example, in the case of the Kalman filter, MISMO discovered that there are at least four data structures.

4.6 Evaluations

For the experiments, our main focus is to validate MISMO's main functionality (extracting semantic information from embedded firmware binaries). To that end, we evaluated MISMO on a set of real-world embedded applications, including drones, self-driving automobiles, smart home devices, robotics, 3D printers, as well as the embedded kernel

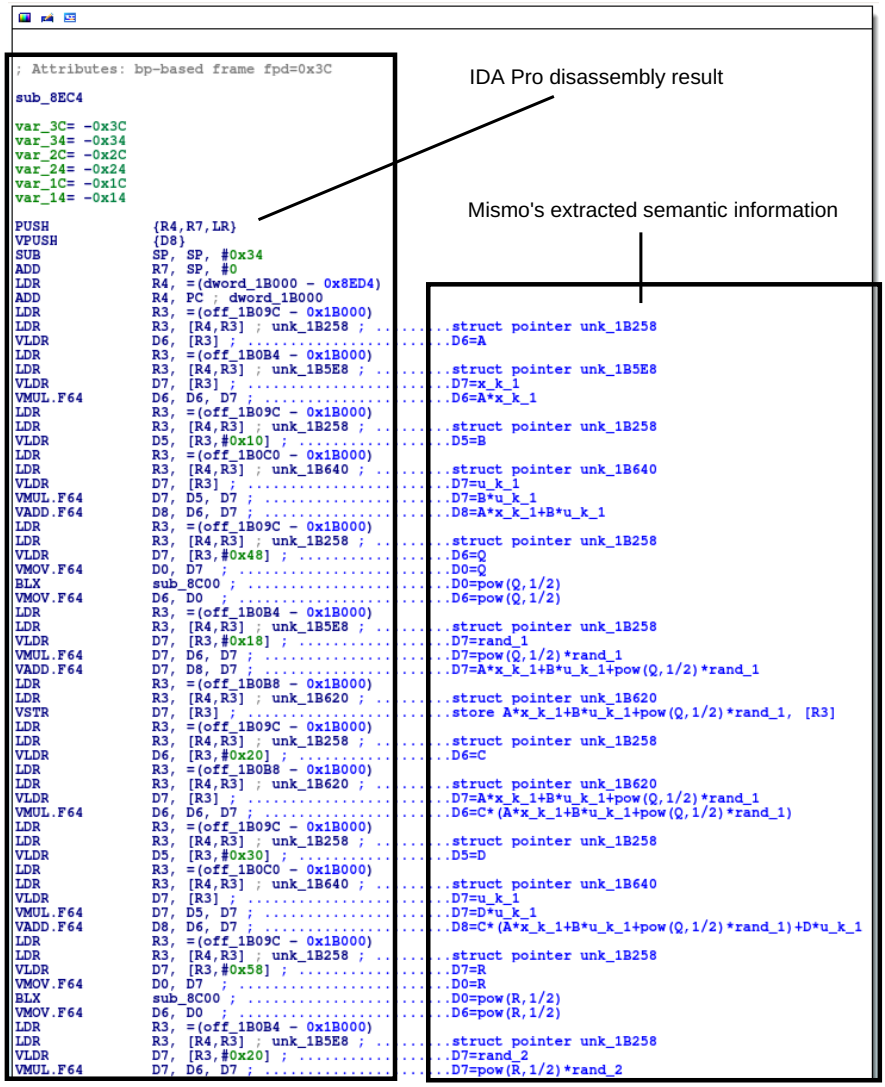


Figure 4.7: Mismo provides semantically rich information for the Kalman filter binary program within the IDA Pro view.

Category	Vendor	Control Algorithm					#firmwares	Data Accuracy (%)	Code Accuracy (%)
		BB	KF	PF	PID	PWM			
Drone	Bitcraze		✓		✓	✓	38	100.00	96.40
	Ardupilot		✓		✓	✓	168	78.57	86.96
	DJI		✓		✓	✓	66	100.00	93.69
	3D Robotics		✓		✓	✓	327	78.57	86.96
	Cleanflight				✓		48	71.43	50.26
	Fluoreon				✓		1	77.78	48.70
	Eachine				✓		1	77.78	48.70
	Paparazzi				✓		53	77.78	86.14
	Cheerson		✓		✓	✓	169	84.29	91.56
Automotive	Baidu		✓		✓		2	100.00	93.67
	PolySync				✓		3	100.00	97.01
	Microsoft				✓		1	100.00	100.00
	Tier IV		✓		✓		11	100.00	89.47
	Udaticy		✓	✓	✓		2	100.00	97.14
3D Printer	LulzBot	✓					22	90.91	92.86
	Makerbot				✓		19	88.89	63.81
	Repetie	✓			✓	✓	6	100.00	82.96
	Printrbot	✓			✓		22	90.91	92.86
	BCN3D	✓			✓		15	81.82	50.26
	Robo3D				✓		1	90.91	92.86
	Teacup	✓			✓	✓	1	100.00	93.24
	Solidoodle				✓		2	90.91	92.86
Robotics	ROS		✓	✓	✓	✓	62	88.89	94.20
	Robotiq				✓		1	100.00	98.64
	LinuxCNC				✓		145	53.85	43.34
	Drake		✓		✓		8	85.71	87.38
Smart Home	SmartPID				✓		2	100.00	100.00
	Particle				✓		87	100.00	96.81
	MBED				✓	✓	147	100.00	100.00
Linux Kernel	Linux Kernel				✓		833	100.00	100.00
Total/Average	**30**						**2,263**	**89.82**	**84.96**

Table 4.2: Embedded IoT/CPS firmware vendors and the corresponding identified control algorithms (BB: Bang-Bang, KF: Kalman Filter, PF: Particle Filter, PID: Proportional-Integral-Derivative, PWM: Pulse Width Modulation)

controller (see Section 4.6.1).

To the best of our knowledge, Mismo is the first solution to provide high-level semantic information about IoT controller binaries, hence we cannot compare Mismo's core capability with any prior work. However, one potential use-case enabled by Mismo is semantic comparison of the embedded controller software binaries. Prior work has developed solutions to calculate the similarities between any pair of functions based on their syntactical features (e.g., number of basic blocks, instructions, etc.). Mismo can perform more in-depth semantic similarity calculations in a specific application domain, i.e., embedded IoT controllers. As a part of our evaluation (see Section 4.6.2), we compare how accurately Mismo distinguish similar function binaries with the solutions by the prior work, namely BinDiff Dullien and Rolles (2005), BinJuice Lakhotia et al. (2013) and Blex Egele et al. (2014).

Additionally, we briefly demonstrate Mismo's capabilities in a few use-cases in the contexts of data type recovery, binary decompilation, as well fine-grained sensitive data

protection (see Section 4.6.3).

MISMO assumes the application's source is unavailable (as in third-party commercial software analysis settings). However, for our accuracy results, we use the source code as the ground truth to evaluate the correctness of MISMO's outputs (as discussed in details later).

4.6.1 Real-world Embedded/IoT Firmware

We first evaluated MISMO on ten representative real-world applications listed in Table 4.3. For evaluation, we collect the ground truth by compiling binaries with debugging information to validate MISMO's findings after it finishes its analyses. It is important to note that MISMO does not utilize the ground truth during its analysis - it is only used for evaluation purposes. In each case, our goal was to locate the control algorithm implementation function and extract the corresponding high-level algorithmic semantics from the function's binary code and data.

Individual application binaries consisted of several control flows that did not pertain to the core of the target control algorithm. These control flows typically correspond to input validation, error checking, etc. as discussed in Section 4.5. MISMO chose the control flow paths that represented the candidate function for each case. Although some of the applications' CFGs were immediately pruned to a single path, a few of the applications still had multiple candidates left, e.g., `PX4fmu` (drone controller) had four possible candidates. `PX4fmu` was found to have four different control modes, each having a slightly different algorithm implementation.

Figure 4.8 shows the number of symbolic input variables for each application's target function as well as the source type of each symbolic input variable. For example, there are 14 total symbolic inputs for `3DRsolo`. 12 out of the 14 symbolic inputs stem from global variables, while the other 2 are from function parameters. In that case, the function parameters were trivial to resolve as they were not pointer data types. Similarly, `PolySync` has 8 symbolic inputs, all of which stem from function parameters: 3 non-pointer values and 5 pointer values. Such data type recovery provides useful information to help with the reverse engineering and binary analysis procedures.

Classification	Application	Details
Drone	Crazyflie	Crazyflie Nano Quadcopter Firmware
	Ardupilot	Ardupilot is the most advanced autopilot software
	Px4fmu	PX4 Pro Drone Autopilot Firmware
	3DRsolo	3drobotics Ardupilot Solo
Automotive	WoDCar	Microsoft The Self Driving RC Car
	PolySync	The Car Control Project for an autonomous driving vehicle.
Smart Home	SmartPID	Smart temperature and process controller: heating or cooling
Linux kernel	Tmon	A Monitoring and Testing Tool for Linux kernel thermal subsystem
Robotics	ROS	Ros_arduino_bridge is a ROS driver and base controller for Arduino microcontrollers
3D Printer	Marlin	Marlin 3D Printer Firmware for RepRap 3D printers

Table 4.3: Embedded applications with control algorithm implementations.

The accuracy of Mismo is defined as the portion of the inputs that has been semantically explained correctly, i.e., the inputs have been mapped to their corresponding mathematical algorithm-level parameters. There are cases during the semantic-matching process where two binary-level variable's arithmetic result match one semantic (algorithm-level) parameter. In these cases, Mismo associates the values in program memory with the semantic parameter's tag.

Of course, there are also cases where the symbolized inputs cannot be resolved (mapped to their algorithm-level counterparts). For example, Mismo was not able to associate two of the algorithm's semantic variables with any of the symbolized inputs of the `3DRsolo`. This is due to the fact that there are too many symbolized variables that confuses Mismo's automated semantic discovery process. For instance, to compute a `derivative` value, the implementation introduces three extra auxiliary variables.

Figure 4.9 shows Mismo's accuracy for the applications from the data and code semantics discovery, respectively. Once Mismo adds the semantic information for the binary-level input variables, Mismo will propagate the semantics for each instruction– as was previously shown in Figure 4.7. The accuracy is then calculated based on the portion of the instructions that has been resolved and correctly annotated in the chosen execution path.

Figure 4.10 shows the relative runtime of Mismo for individual applications. On average, the total time required by Mismo to complete its reverse engineering took less than 2s for each application. Most of the time was due to the symbolic expression generation through symbolic execution of the candidate execution paths.

To ensure Mismo is generalizable, we further evaluated Mismo on 2,263 firmware

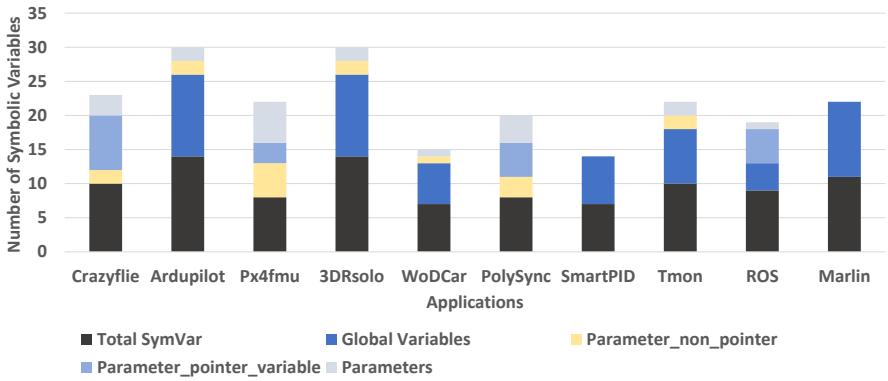


Figure 4.8: Number of symbolic variables and the data sources.

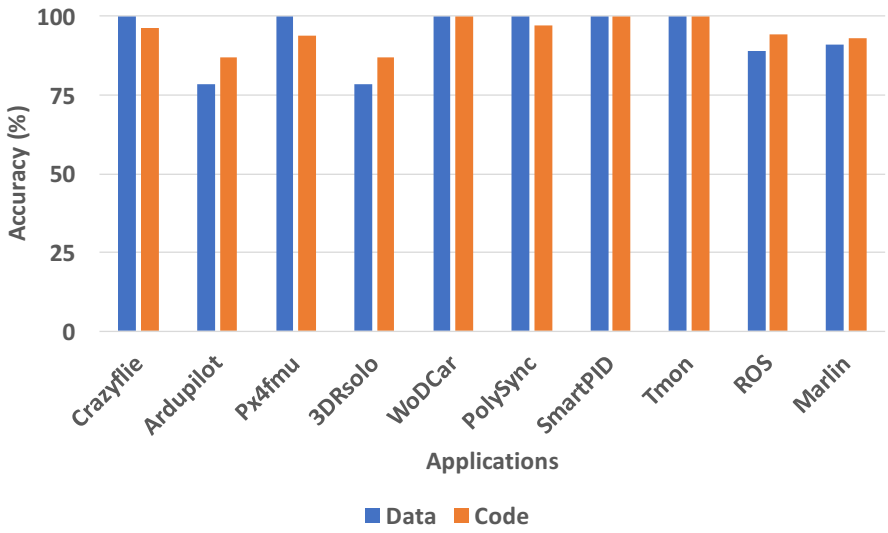


Figure 4.9: Accuracy of data and code semantics discovery.

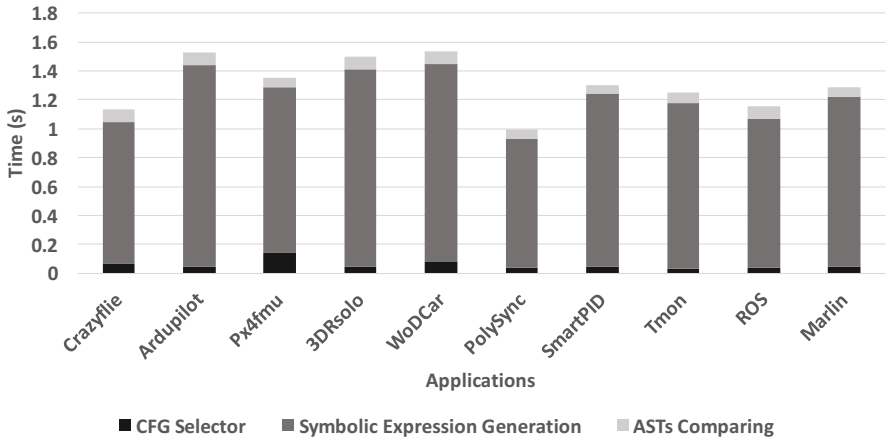


Figure 4.10: MISMO analysis time on 10 real world applications.

	Kalman Filter					Particle Filter					Proportional-Integral-Derivative					Pulse-Width Modulation				
	BD	BL	BJ	Mismo	G	BD	BL	BJ	Mismo	G	BD	BL	BJ	Mismo	G	BD	BL	BJ	Mismo	G
Crazyflie				✓	✓									✓	✓				✓	✓
Ardupilot				✓	✓									✓	✓				✓	✓
Px4fmu				✓	✓									✓	✓				✓	✓
3DRsolo				✓	✓									✓	✓				✓	✓
WoDCar														✓	✓					
PolySync														✓	✓					
SmartPID														✓	✓					
Tmon														✓	✓					
ROS				✓	✓				✓	✓				✓	✓				✓	✓
Marlin														✓	✓					

Table 4.4: Comparison among BinDiff (BD), Blex (BL), BinJuice (BJ) and Mismo. G indicates the ground truth.

binaries from over 30 different vendors. The results of our analyses are shown in Table 4.2. The 2,263 firmware images consisted of several different control algorithms, the most popular being PID, Kalman filter, and pulse-width modulation (PWM) implementations. Mismo's accuracy for data discovery was an average of 89.82%, and 84.96% for code semantics discovery. The average false positive rate was 2.86% that is promising for real-world use-cases.

4.6.2 Comparison with Prior Work

As discussed earlier, other existing solutions, by design, cannot achieve Mismo's main objective, i.e., to extract high-level algorithmic semantics from the low-level binary implementations. Instead, they mainly focus on similarity checking between two low-level binary implementations. However, we compare Mismo with current state-of-the-art binary similarity checking solutions (BinDiff Dullien and Rolles (2005), BinJuice Lakhotia et al. (2013) and Blex Egele et al. (2014)) on several commercial embedded controllers. We used Mismo to identify the control algorithms used in those binaries. For BinDiff, BinJuice and Blex, we used binary implementations of the full set of popular control algorithms. We compared them against individual commercial executables. The tool utilized the associated binary implementations to perform pattern-matching.

The results of the comparisons are shown in Table 4.4. Existing solutions for binary comparison of the complied binary set could not find any matching function for any of the applications. In contrast, Mismo was able to provide accurate semantic reverse engineering for all 10 commercial controller firmware packages.

4.6.3 Selected Mismo Use-Cases

Data Type Recovery and Decompilation

Snowman sno is a well-known machine code to C/C++ decompiler. We show how Mismo can recover more precise semantics such as data type information and high-level semantics for a control algorithm implementation. In Table 4.5, there is one data structure `PID` which has been used by the function implementation of `PolySync`. Snowman does not reverse engineer any data structures other than `int` and `double` data types. However, as shown in the source code, there should be one data structure whose members are of type `double`. Mismo extracts one data structure and the data type of each field. Mismo failed to recover the last field due to the fact that it was never used by the function.

Source Code	Snowman Reversed Result	Mismo Reversed Result

`typedef struct`
`{`
`double windup_guard;`
`double proportional_gain;`
`double integral_gain;`
`double derivative_gain;`
`double prev_input;`
`double int_error;`
`double control;`
`double prev_steer_angle;`
`} PID;`
	`signed int v6;`
	`double v19;`
	`double v20;`
	`int v21;`
		`struct`
		`{`
		`0x00: double SymVar;`
		`0x08: double Kp;`
		`0x10: double Ki;`
		`0x18: double Kd;`
		`0x20: double prev_md;`
		`0x28: double integral;`
		`0x30: double output;`
		`}`

`diff = ((input - pid->`
`prev_input)/dt);`
	`_R3 = v21;`
	`__asm`
	`{`
	`VLDR D7, [R3,#0x20]`
	`VLDR D6, [R7,#0x4C+v44]`
	`VSUB.F64 D6, D6, D7`
	`VLDR D7, [R7,#0x4C+v4C]`
	`VDIV.F64 D7, D6, D7`
	`VSTR D7, [R7,#0x4C+v24]`
	`}`
		`reg_D6 = md_value`
		`- previous_md;`
		`reg_D7 = reg_D6/dt;`

Table 4.5: Comparing the reverse engineering results between Snowman and Mismo.

Furthermore, the decompilation precision of both tools was evaluated on a binary translation of a single line of source code. As shown in the Table 4.5, Snowman does not provide much of useful semantic information that can simplify the binary reverse engineering process. Mismo recovers much more semantic information that is very similar to the semantic level source code.

Bug Discovery

The semantic information recovered by MISMO can be useful for finding bugs in a closed-source binary file. We present our exciting finding of a concrete case in which we found a bug in the PID algorithm implementation of the Linux kernel from version 3.13 to the present by analyzing the semantic information recovered by MISMO. We found a similar bug in the controller implementation of the Android Things kernel for IoT devices by Google. The bug lies in the computation of the values of `i_term` and `d_term` parameters in the PID control algorithm. MISMO reported a mismatch between the kernel's PID implementation and the high-level PID algorithm. Upon our investigation, we identified the incorrect implementation and buggy code statement within Linux kernel.

Figure 4.11 shows the part of program code[1] of PID implementation and adopted algorithm expression that is the PID Type-C algorithm. With source code's help, it will be easy to figure out the inconsistency between and algorithm and code implementation. However, for MISMO's context, we only have the stripped binary code and PID algorithm candidates. MISMO was able to discover the above-mentioned inconsistency.

Initially, MISMO located the control algorithm implementation function out of the 44 functions in the Linux kernel `tmon` module, which implements different control algorithms. MISMO refined the control flow graph by pruning the irrelevant basic blocs. MISMO symbolicly executed the remaining execution path in the CFG and generated the corresponding symbolic expression in the form of an abstract syntax tree (see Figure 4.12). Finally, MISMO compared Figure 4.12 with abstract syntax tree of different types of known PID algorithms (see Figure 4.13). The inconsistency occurs in Figure 4.12, where there are three nodes with `Sym_2` label under the three subtrees each rooted at a `Mul` node. However, based on the algorithm graphs (Figure 4.13), there should be no same variable under the three `Mul` subtrees. As the result MISMO's constraint satisfaction returns no possible concretization of the symbolic values for a match between the

[1]Our discussion is at the source code level for readability, whereas our design and implementation assume only the binary.

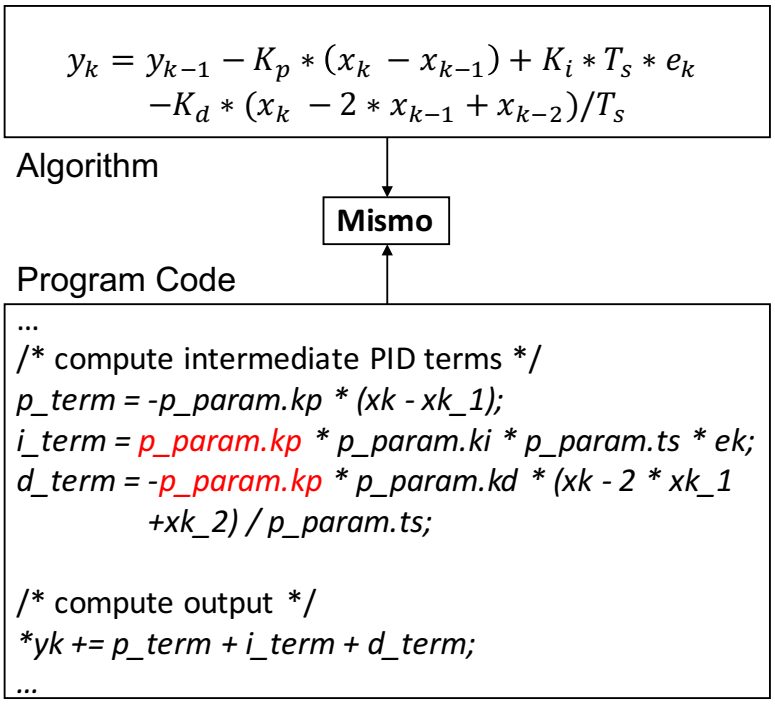


Figure 4.11: MISMO detects the bug in Linux Kernel.

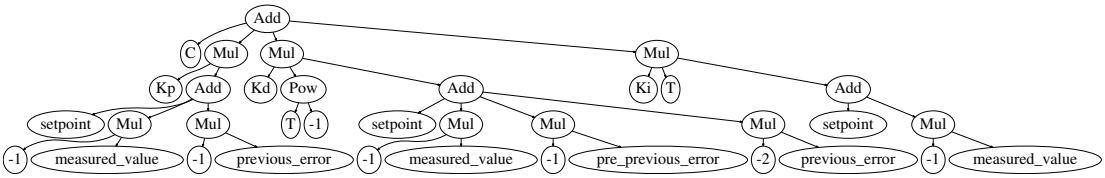
Figure 4.12: Abstract syntax tree for the symbolic output value of the Linux kernel PID implementation (w/o semantics, e.g., parameter names).

graph in Figure 4.12 and any of the graphs in Figure 4.13. Upon our manual inspection as the result, we discovered the bug in the PID algorithm implementation of the Linux kernel in Android Things framework for IoT devices.

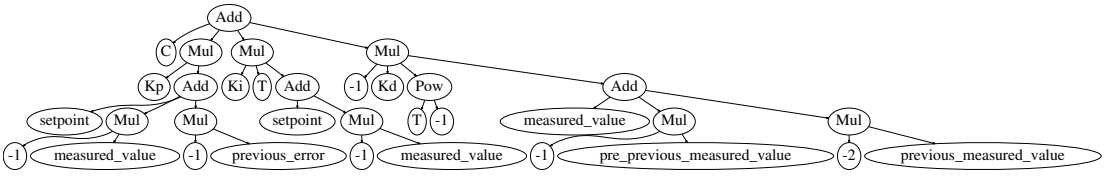
The associated expression was found to have a different expression than the actual Type-C control algorithm. The `i_term` and `d_term` values were found to have an extra K_p value. Figure 4.14 conveys the difference between the correct Type-C algorithm and the implementation that contains this bug. This bug could be crucial considering that the Linux kernel is widely used in many real-time embedded cyber-physical systems.

Fine-Grained Sensitive Data Protection

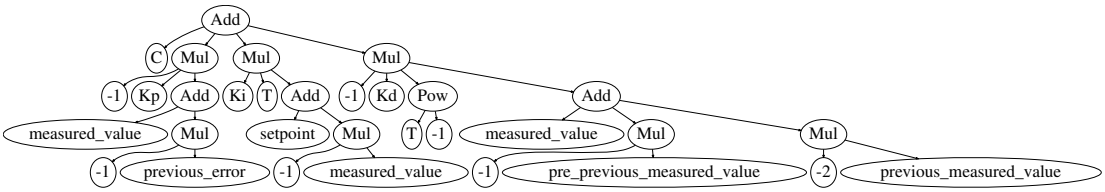
The semantic information provided by MISMO can also be used for fine-grained sensitive data protection. The first step is to determine which data is sensitive enough



(a) Abstract syntax tree for the PID type A algorithm (w/ semantics)



(b) Abstract syntax tree for the PID type B algorithm (w/ semantics)



(c) Abstract syntax tree for the PID type C algorithm (w/ semantics)

Figure 4.13: Abstract syntax tree for PID algorithm

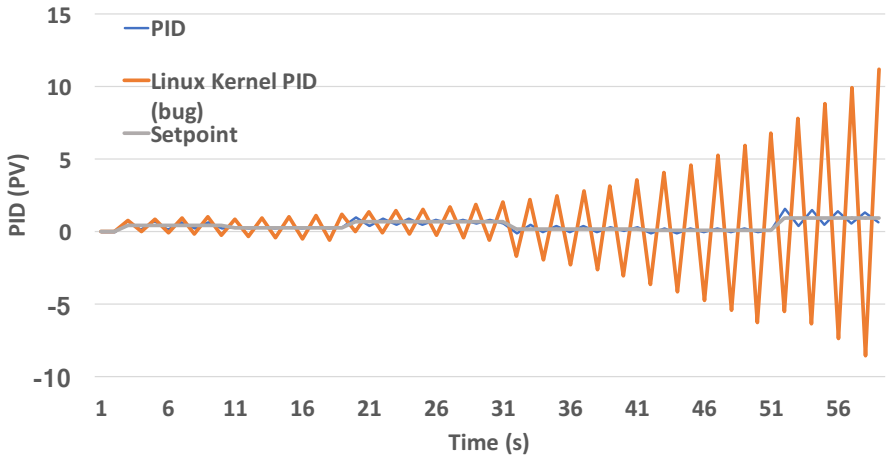


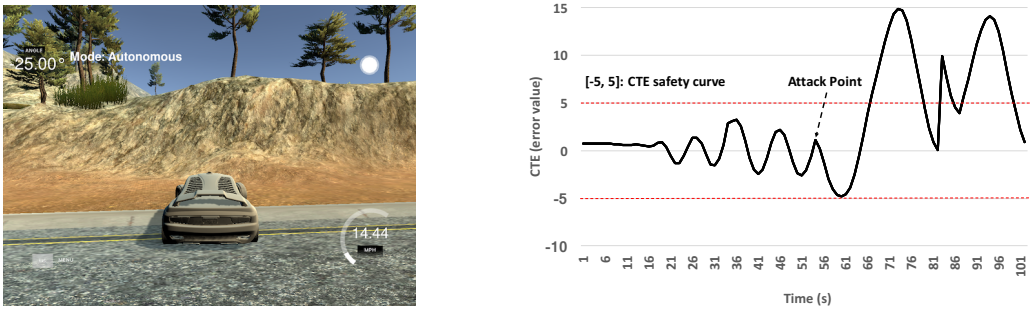
Figure 4.14: Comparing controller output between correct implementation and buggy Linux Kernel implementation.

to require extra security protection. Traditionally, sensitive data has referred to sensitive *information*, e.g., passwords, credit card numbers, and health records. MISMO can identify sensitive data in the controller's live memory that is important for securing safety-critical cyber-physical systems.

To demonstrate such fine-grained data protection, we use one example to show how MISMO can identify a control algorithm's sensitive data, as well as the implications of compromising such sensitive data.

For an autonomous car steering, the parameters' value integrity is crucial for secure operation of the underlying control algorithm and ensuring stable control of the vehicle's actuators. As a case in point, the values of the different control gains (K_p for proportional, K_i for integration, and K_d for derivative) as well as the associated code are statically defined and do not change throughout the car's operation.

Assume an attacker has access to the binary associated with the controller (e.g., the firmware downloaded from an online repository used for firmware updates). The attacker may modify the parameter value offline or during its execution at runtime (e.g., using a data corrupting return-oriented programming exploit) in order to induce an unsafe state such as a car crash. The commercial autonomous controller example in our experiments limits the controller output (normalized steering wheel degree) to the range [-1, 1]. One possible attack vector would be to modify and expand the value range to also include unsafe values. Additionally, an attacker can utilize MISMO to figure out the location of K_p and modify its value. We implemented the latter attack.



(a) Car crash visualization using the autonomous controller.

(b) Modified gain parameter of the controller causes the crash.

Figure 4.15: Car crash through sensitive controller parameter identification and corruption.

We developed an exploit to target the control parameter (steering wheel degree) value in order to keep the controller computation output out of range. Figure 4.15a shows the consequence of the attack. After a few seconds, the car keeps traveling in a circle. The figure shows that the car proceeds to veer off the road. We also traced the change of cross track error (CTE) value, which is the lateral distance between the car and the reference trajectory (see Figure 4.15b).

To protect the controller software against such targeted memory data attacks, Mismo determines the low-level binary variables, register or memory address, represents the critical high-level control gains in the binary code. To prevent the attacker from modifying these critical control gains, one can take one or more of the several possible countermeasures such as dynamic monitoring the value on these memory location, or isolation of these critical control gains by memory isolation Kim et al. (2018) and binary rewriting Kim et al. (2017).

4.7 Related Work

Binary reverse engineering.

The existing advanced tools, e.g., IDA Pro Hex-Rays, Boomerang Emmerik and Waddington (2004) and OllyDbg oll offer a variety of techniques to help elevate low-level machine code to higher level assembly instructions. Snowman sno can provide native code to C/C++ decompilation. However, these tools do not provide an automated means of deriving the high-level algorithmic semantics, leaving the analyst with the responsibility of finding semantic and domain-specific information. Phoenix Schwartz et al. (2013) provides semantic-preserving structural decompilation analysis. TOP Zeng et al. (2013) reconstructs program source code from execution traces. Unlike decompilation that statically transforms a piece of binary code, TOP dynamically translates it with more runtime information and generates reusable software components. However, TOP cannot ensure semantic recovery for controller algorithms as it cannot fill the semantic gap between the recovered source code and the corresponding abstract algorithmic concepts, i.e., parameters and mathematical operations. Binary reverse engineering

techniques Cozzie et al. (2008); Lin et al. (2010); Lee et al. (2011); Slowinska et al. (2011); Sun et al. (2016) can extract data types from binaries accurately. The recent work Lin et al. (2010); Sun et al. (2016) recover only the information that comes from the semantics of system call parameters. Although type information is useful in binary reverse engineering, it still does not provide high-level algorithm-level semantics about the binary code and data segments.

Low-level similarity checking.

A plethora of CFG-based code similarity algorithms have been previously proposed Kruegel et al. (2005); Sokolsky et al. (2006); Vujošević-Janičić et al. (2013), but there has been no work on comparing low-level binary instructions to high-level mathematical algorithmic expressions. Differential testing introduces an effective way of discovering low-level differences between independent implementations with similar intended functionality. BinDiff Dullien and Rolles (2005); Flake (2004) has been widely used and has established a reputation of being the industry standard for binary diffing. BinDiff starts by recovering the control flow graphs (CFGs) of two binaries and uses heuristic to normalize and match the vertices from the two graphs. BinJuice Lakhotia et al. (2013) extracts syntactic equations from basic blocks to measure similarity between two basic blocks. Blex Egele et al. (2014) uses a dynamic approach for binary code search. The search depends on the two similar codes having syntactically similar low-level execution behavior. Xu , *et al.* Xu et al. (2017) leverage neural networks to find similarities among executables based on graph-theoretic analysis of their control flow graphs.

For our problem, one major issue with the aforementioned solutions is that most controller software have custom implementations of the associated algorithms. Hence, directly comparing a compiled implementation of an algorithm with a separate implementation will be fruitless across different implementations.

Embedded firmware analysis.

Avatar Zaddach et al. (2014); Muench et al. (2018) provides a framework to support firmware dynamic analysis. It integrates target hardware and an emulator based on

the selective symbolic execution to provide reverse engineering and vulnerability detection. FIE Davidson et al. (2013) presents a platform to detect memory safety issues in firmware on the MSP430 family of micro-controllers. FIE implements a symbolic execution engine based on KLEE Cadar et al. (2008b) and requires source code access. Costin , *et al.* Costin et al. (2014) propose large-scale firmware analysis and mainly focus on simple static analysis techniques only. Firmalice Shoshitaishvili et al. (2015) develops a static binary analysis framework for firmware analysis. It requires manual annotations, and hence its use for complex control algorithm implementations may be limited in practice. The aforementioned solutions aim at vulnerability assessment of the firmware binaries through low-level binary analysis without the knowledge about the high-level algorithmic semantics. Therefore, they miss potential algorithmic weaknesses in the implemented controllers.

4.8 Conclusions

We presented Mismo, a general framework to extract semantic information of an embedded firmware binaries with respect to its associated high-level control algorithm. We evaluated Mismo on 2,263 commercial firmware binaries by 30 industry vendors from 6 real-world IoT/ICS application domains. We were able to extract their semantic information with respect to the algorithm. We utilized Mismo to discover a zero-day vulnerability in the most recent Linux Kernel, and provided fine-grained protection of sensitive data on a self-driving automobile application.

Chapter 5

I Know What You Didn't Do Last Vulnerability! Firmware Analysis via Deep Learning for Known Security Vulnerabilities

5.1 Introduction

Every year a new record is set for the number of known vulnerabilities. The number of vulnerabilities registered in the Common Vulnerabilities and Exposures (CVE) was approximately 4,600 in 2010, grew to approximately 6,500 in 2016, and more than doubled in 2017 with over 14700 vulnerabilities Mell et al. (2006). The vendors of the associated vulnerable software are responsible for issuing a patch for each vulnerability in a timely fashion. However, it has been shown in the past that vendors may not be honest when it comes to reporting whether a vulnerability has been patched or not, especially in the context of mobile and IoT devices.

Because of the increasing ubiquity of mobile and IoT devices in our daily lives, associated vulnerabilities raise concerns of privacy, security, and even safety. Gartner forecasts that 20.4 billion IoT devices will be in use worldwide by 2020 **?**. Currently, patch management for both IoT and mobile devices is a challenge for heterogeneous ecosystems. A 2018 Federal Trade Commission report COMMISSION mentioned that although an ecosystems diversity provides extensive consumer choice, it also contributes to security update complexity and inconsistency. For instance, if Android releases a patch for a vulnerability, all variants of Android need to be updated accordingly. However, vendors often release new versions of software with *hidden patch gaps*–vulnerabilities that were supposed to have been previously patched.

A study showed that 80.4% of vendor-issued firmware is released with multiple

known vulnerabilities, and many recently released firmware updates contain vulnerabilities in third-party libraries that have been known for over eight years Cui et al. (2013). Duo labs found that only 25 percent of mobile devices operated on a recent patch level in 2016 Labs (a). A large study of Android phones Labs (b) found that some Android vendors regularly miss patches, leaving parts of the ecosystem exposed to the underlying risks. As such, a means of identifying vulnerabilities for a given binary image is critical for identifying hidden patch gaps.

Known vulnerability discovery via deep learning. In this paper, we first focus on known vulnerability detection to address the hidden patch gap for heterogeneous mobile and IoT devices. Recently, researchers have started to tackle the cross-platform binary similarity checking to detect known vulnerabilities Pewny et al. (2015); Feng et al. (2016); Eschweiler et al. (2016); Xu et al. (2017). These efforts propose to extract various robust, platform-independent features directly from binary code for each node in the control flow graph that represents a function. Other approaches have focused on binary similarity detection where a graph matching algorithm is used to check whether two functions' control flow graph representations are similar Pewny et al. (2015); Feng et al. (2016); Eschweiler et al. (2016). However, deep learning approaches have proven to be the most promising for known vulnerability detection.

Prior works have shown that deep learning approaches can be used for binary analysis to detect vulnerabilitiesShin et al. (2015); Xu et al. (2017); Chua et al. (2017). The most recent approach Chua et al. (2017) has a performance of 0.971 Area Under the Curve (AUC). However, despite this performance, assuming the target binary has around 3000+ functions, we found that we would narrow down the set of candidate vulnerable functions to about 90 candidate functions (89 of which represent false positives) for the binary. In this work, they assume access to the symbol tables and, as such, are able to further narrow down the candidate functions. However, for stripped commercial-off-the-shelf (COTS) binaries, their solution could only provide a very large (mostly false positives) set of candidate functions. As such, further measures are necessary to prune the candidate functions to identify and report only the true positives (the functions with actual vulnerabilities).

Pruning Candidate functions via dynamic analysis. In order to prune the set of candidate functions, we integrate deep learning-based approaches with dynamic analysis to incorporate dynamic features of candidate functions. The dynamic analysis provides a much richer feature space than prior works that focused only on heuristic or static features of basic blocks and functions Feng et al. (2016); Xu et al. (2017). Due to scalability concerns, these works engineered their solutions for speed at the expense of accuracy. However, we show that integration of dynamic features can not only speed up the vulnerability function matching process, but can also provide high accuracy while removing false positives.

This initial framework allows us to develop a new training and dataset generation method that uses a default policy to pretrain a task-independent graph embedding network. We then use this method to generate a large-scale dataset using binary functions compiled from the same source code but for different platforms and compiler optimization levels. We then built a vulnerability database that includes 1382 vulnerabilities for mobile/IoT firmware.

However, the ultimate goal of our solution is not to only find similar vulnerability functions. The final goal is to ensure whether the vulnerability is still in the target firmware, or it has been patched.

Missing patch detection. Prior work has already developed precise patch presence tests Zhang and Qian (2018). However, this solution only works with access to the source code for both the vulnerable and patched function source code. Also, because this solution relies on binary similarity-based approaches to locate target functions, it suffers from the aforementioned high false positive rate for candidate functions. Our solution works directly with stripped COTS binary and does not require access to the source code while significantly pruning false positives.

In this paper, we present BinSec: a framework that integrates deep-learning for binary similarity-checking with dynamic analysis to discover known vulnerabilities as well as to test for patch presence. Our evaluations demonstrates that BinSec outperforms the state-of-the-art approaches by large margins with respect to both accuracy

and efficiency. Our deep learning model can achieve 96% accuracy for vulnerability detection. Our dynamic analysis engine can further reduce the false positive and can rank the correct matches among the top 3 results 100% of the time. For vulnerability and patch differential analysis, BinSec can correctly distinguish whether a vulnerability has been patched. Finally, BinSec can achieve an average 96% accuracy to locate the function and distinguish whether the function has been patched. There was only one wrong detection out of 25 CVE evaluation (the vulnerability was correctly discovered but it was reported as unpatched while being patched indeed).

Contributions. We summarize our contributions as follows:

- We propose an efficient firmware vulnerability assessment framework that leverages deep learning and dynamic binary analysis techniques to achieve high accuracy and performance in known vulnerability discoveries in stripped firmware binaries without source code access.
- We propose a fine-grained binary comparison algorithm to distinguish accurately between patched and unpatched versions of the same functions binaries. Our solution works cross-platform currently supporting ARM and X86 architectures. The selected relevant features for the comparison enables our solution to pinpoint the unpatched functions with very low false positive rates.
- We evaluate BinSec on 25 CVE vulnerabilities, 100 different Android firmware libraries across 4 different architectures. Our results are very promising for practical deployment in real settings. With most of BinSec's prototype being fully automated, its dynamic analysis module correctly identified and threw away the false positives from the deep learning classification outcomes. The results were later processed, and the unpatched functions were separated from the functions with already-patched vulnerabilities.

5.2 Overview

We first introduce the vulnerability function similarity problem and challenges in 5.2.1 and then present an overview of our solution in 5.2.2.

5.2.1 Problem Setting and Challenges

In this paper we consider the problem of searching for known vulnerabilities in stripped COTS mobile/IoT binaries. We assume that we do not have access to the source code. We also assume that the binary is not packed or obfuscated and that the binary is compiled from a high-level procedural programming language, i.e., a language that has the notion of functions. While handling packed code is important, it poses unique challenges which are out of scope for this paper. Considering these assumptions, we identify the following challenges that arise in the domain of mobile/IoT platforms.

Heterogeneous binary compilation. Mobile/IoT platforms typically consist of heterogeneous distributions of hardware that may share common software vulnerabilities. As such, we explicitly consider cases where different cross-platform compilations with different levels of optimization produce different binary programs from identical source code. This way, we can develop a query function may come from different hardware architectures (e.g., x86 and ARM) and software platforms (e.g., Windows, Linux and MacOS).

Copious amount of candidate vulnerable functions. To illustrate the scale of the number of candidate vulnerable functions, we analyzed the firmware of Android Things 1.0 and IOS 12.0.1. For Android Things 1.0, we found 379 different libraries which included 440,532 functions, while IOS 12.0.1 contained 198 different libraries with 93,714 functions. Although prior works have shown that deep learning-based methods can be used to identify a set of vulnerable candidate functions with high precision Xu et al. (2017), these techniques rely on symbol tables–which are not available on stripped COTS binaries–to prune candidate functions. As such, there remains a challenge to prune candidate vulnerable functions for stripped COTS binaries.

Differentiating between patched or vulnerable code. Vulnerable functions may not be very distinguishable from their patched versions as a patch may be as little as changing a single line of code. Prior works have also used deep learning to detect whether or not vulnerable code has been patched Zhang and Qian (2018). However, this solution relies on access to the source code for both the vulnerable code as well as

the patched code. In practice, we often do not have access to the source code of binary functions.

Given these motivating challenges, we now present an overview of the BinSec framework.

5.2.2 Overview

An overview of the BinSec framework is presented in Figure 5.1. Our solution is implemented in four steps: (1) deep learning is used to train the vulnerability detector; (2) the vulnerability detector is used to analyze the target mobile/IoT firmware; (3) the identified vulnerable components are run for verification of the existence of a vulnerability; and (4) based on extracted static and dynamic features of vulnerable and patched functions, we identify whether the candidate vulnerability function has been patched.

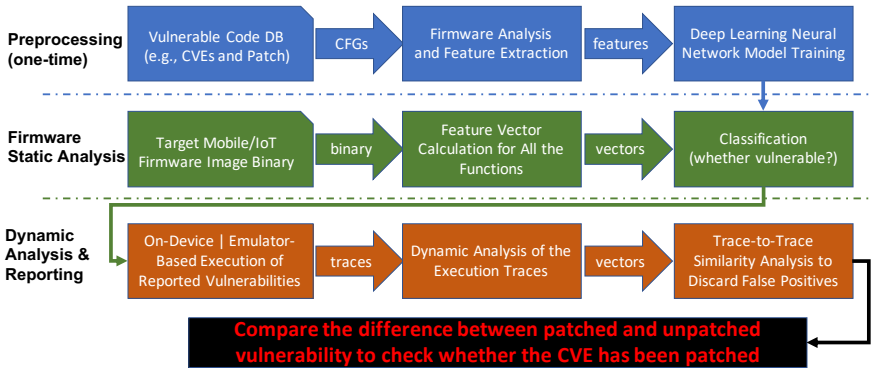


Figure 5.1: BinSec vulnerability and patch search workflow.

BinSec's objective is to compare the functions within firmware binaries to the set of known CVE vulnerabilities as well as any associated patches. BinSec outputs the vulnerable points (functions) within the target firmware image and the corresponding CVE numbers. To compare two binary functions at runtime, BinSec combines static and dynamic programming language analysis techniques along with deep learning methods from AI and machine learning.

BinSec starts with lightweight static analysis to convert each function within a binary to a machine learning feature vector. BinSec then leverages a previously trained

deep neural network model to determine if the two functions (one from the firmware binary and the other one from the CVE database) are similar, i.e., coming from the same source code with possibly different compilation flags. If the two functions are detected to be similar, BinSec performs a more in-depth dynamic analysis to ensure the report by the static analysis is not a false positive and indeed indicates a matching function pair.

To perform dynamic analysis, BinSec leverages runtime DLL binary injection and remote debugging solutions to run the CVE vulnerable function binary as well as the target firmware function binary on identical input values (e.g., function arguments or global variables) within the corresponding mobile/IoT embedded system platform. BinSec captures the execution traces of the two function binary executions and extracts dynamic features such as number/type of executed instructions, number/type of user-defined and library function calls, amount of stack/heap data read/writes, etc for each execution trace.

Using the extracted features, BinSec calculates a similarity measure between the two functions and determines whether the report by the static analysis was indeed correct. If so, the target function within the firmware is reported to be vulnerable along with the corresponding CVE number. It is noteworthy that BinSec's analysis is performed without any source code access and hence its deployment does not rely on cooperation of the firmware vendors.

Since we don't really know whether the reported function is patched, BinSec will first compare the difference based on their static features and restart the whole process based on the patched version of the vulnerable function. BinSec then uses the differential engine to analyze the static/dynamic features as well as the similarity score to decide whether the function has been patched.

5.3 Design

In this section we present the design of the BinSec framework that explores any given mobile/IoT firmware binary executable and discovers and reports vulnerable points

Feature Name	Feature Description
no_c	number of constants value
no_s	number of strings
no_inst	number of instruction of the function
l	size of local variables in bytes
f	function flags
no_i	number of import functions
no_ox	number of code references from this function
no_cx	number of function calls from this function
z	the size of function
min_i_b	the minimal number of instruction for basic block
max_i_b	the maximal number of instruction for basic block
avg_i_b	the average number of instruction for basic block
std_i_b	the standard deviation of number of instruction for basic block
min_s_b	the minimal size of basic block
max_s_b	the maximal size of basic block
avg_s_b	the average size of basic block
std_s_b	the standard deviation of size of basic block
num_bb	the number of basic block for each function
num_edge	the number of edge of among basic blocks for each function
cyclomatic_complexity	the complexity of the function
fcb_normal	normal block
fcb_indjump	block ends with indirect jump
fcb_ret	return block
fcb_cndret	conditional return block
fcb_noret	noreturn block
fcb_enoret	external noreturn block (does not belong to the function)
fcb_extern	external normal block
fcb_error	block passes execution past the function end
min_call_b	the minimal number of call instruction of each basic block
max_call_b	the maximal number of call instruction of each basic block
avg_call_b	the average number of call instruction of each basic block
std_call_b	the standard deviation of call instruction of basic block
sum_call_b	the total number of call instruction of the function
min_arith_b	the minimal number of arithmetic instruction of each basic block
max_arith_b	the maximal number of arithmetic instruction of each basic block
avg_arith_b	the average number of arithmetic instruction of each basic block
std_arith_b	the standard deviation of arithmetic instruction of each basic block
sum_arith_b	the total number of arithmetic instruction of the function
min_arith_fp_b	the minimal number of arithmetic FP instruction of each basic block
max_arith_fp_b	the maximal number of arithmetic FP instruction of each basic block
avg_arith_fp_b	the average number of arithmetic FP instruction of each basic block
std_arith_fp_b	the standard deviation number of arithmetic FP instruction of each basic block
sum_arith_fp_b	the total number of arithmetic FP instruction of the function
min_betweeness_cent	the minimal number of betweeness centrality
max_betweeness_cent	the maximal number of betweeness centrality
avg_betweeness_cent	the average number of betweeness centrality
std_betweeness_cent	the standard deviation number of betweeness centrality
betweeness_cent_zero	how many node the betweeness centrality is zero

Table 5.1: Function features used in BinSec.

in the binary code/data segments of the firmware without access to its source code. Beyond the similar vulnerable code discovery, BinSec can also accurately test the security patch presence in the target firmware binary.

5.3.1 Known Vulnerability Discovery via Deep Learning

Comparing with the previous bipartite graph matching bin and dynamic similarity testing Egele et al. (2014), deep learning approaches Xu et al. (2017) can achieve significantly better accuracy and efficiency for known vulnerability discovery. This is due to the fact that deep learning approaches can evaluate graphical representations

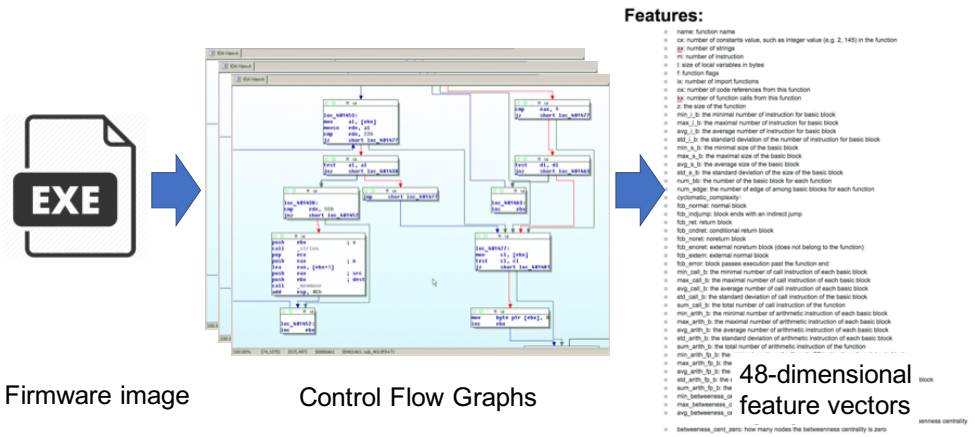


Figure 5.2: Detailed BinSec Architecture for static analysis of mobile/IoT firmware.

of binaries as a whole and also can automatically learn relationships without manually defined rules. BinSec uses a deep learning approach as a first step to generate a list of candidate vulnerable functions on the order of seconds. However, in order to accommodate our prior assumptions, we need to first need to build a training dataset that extracts function features to train a deep learning model.

Feature extractor. In order to extract function features, BinSec first analyzes functions in assembly format. Marking the correct boundary, scope and range of each assembly routine is usually the first problem to solve. Also, distinguishing between code and data is equally important. The input for BinSec's neural network model is the function feature vector that is extracted from the disassembled binary code of the target function. To obtain this feature vector, the first step is to identify the function boundaries. Function boundary identification with minimal reliance of instruction set semantics is an independent problem of interest. Previous approaches range from traditional machine learning techniques Bao et al. (2014) to neural networks Shin et al. (2015) to applying function interface verification Qiao and Sekar (2016). In this work, we assume that these steps are handled by the disassembler using a robust heuristic technique. A disassembler can provide the control flow graph (CFG) of a binary–a common feature used in vulnerability detection.

Figure 5.2 shows the procedure for BinSec's function feature extraction. BinSec utilizes the CFG with different basic block-level attributes as the features to model the

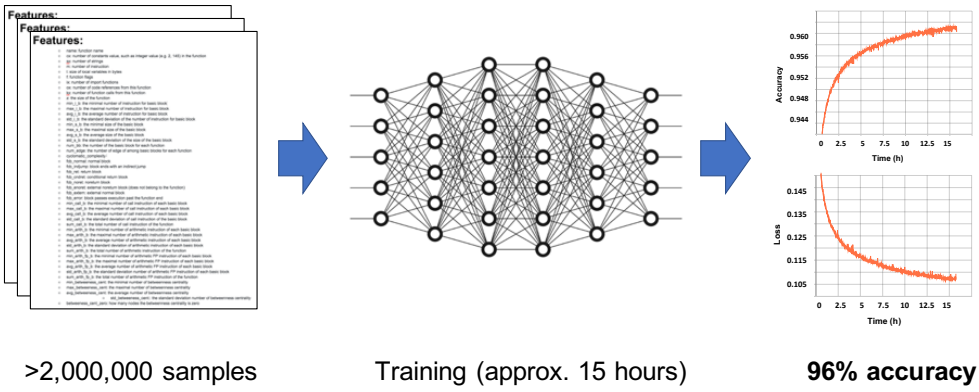


Figure 5.3: Training the neural networks for automated firmware vulnerability assessment of mobile/IoT firmware.

function in our problem. For each function, BinSec can extract function-level, basic block-level and inter-block-level information. Table 5.1 shows the extracted interesting 48 features from each function for generating a feature vector. Feature extraction is one very important step for later building neural network model. It is very important to design flexible and efficient architectures for data collection because of how much it can speed up the process of building the deep learning model. BinSec keeps the feature extraction rich (48 features), efficient (automated feature extraction) and scalable (multi-architecture support).

Training the deep learning model. For BinSec's deep learning model, we adapt a sequential model that is composed of linear stack of layers. For our case, we adapt the sequential model with 6 layers. Figure 5.3 depicts an actual example process of training the model with a 6-layer network. We first specify the input for each layer. The first layer in our sequential model needs to receive information about its input shape. The model is training using the extracted function features in our dataset built from 2,108 binaries with different architectures. The training took approximately 15 hours and resulted in a model with 96% accuracy.

5.3.2 Pruning candidate functions via dynamic analysis

We now discuss how we can use dynamic analysis to further prune the candidates produced from the previous deep learning stage. The goal is to determine whether

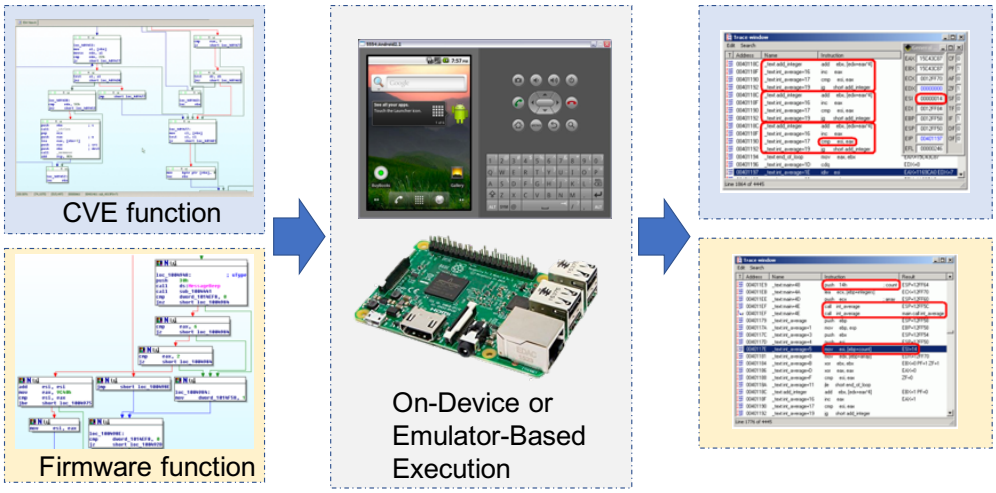


Figure 5.4: Concrete execution of potentially vulnerable code segments on embedded devices for BinSec's detailed dynamic analysis.

the reported pair of matching functions from the previous stage are indeed a match. Using dynamic analysis, we can execute functions of the two binaries functions with the same inputs and compare observed behaviors and features for similarity. Obviously, two functions may be compiled with different flags and, hence, the execution traces of the two functions on the same input data may differ drastically if investigated syntactically only. Hence, our analysis will consider the semantic similarity of the execution traces in terms of the ultimate effect on the memory after the two functions finish their execution on the identical input values.

As such, we extract features of the execution traces and compare the feature vectors of the two traces as the result of two function executions on the same input values. If the observed features are similar across different generated inputs, we gain confidence that they are semantically similar. We are essentially implementing a dynamic equivalence testing system to evaluate the two potential similar functions. Our current system observes different features from an execution.

There are a few challenges to apply the dynamic analysis for actual execution. The key challenges reside in the preparation of the execution environment as well as the simultaneous monitoring of the execution. Furthermore, because we are working in a heterogeneous mobile/IoT ecosystem, concretely running binary code to obtain execution traces is not trivial, especially since "valid" values are required for correct

function execution. We first discuss the preparation of the inputs that feed into the dynamic analysis engine.

Inputs to the dynamic analysis engine. A key challenge to designing BinSec's dynamic analysis engine is preparing the associated inputs. The dynamic analysis engine takes in two inputs: the program binary, **F**, and the *execution environment* of **F**. The program binary contains the target function, `f`. In order to concretely execute the target function for dynamic analysis, we need to figure out *how* to execute the target function. For instance, we need to provide concrete and valid input values. Furthermore, actually running the function typically requires loading and executing the entire program binary. However, given a program binary, one cannot just instruct the operating system to start execution at a particular address. As such, we need a means of providing an *execution environment* that can efficiently encapsulate the required execution state.

BinSec uses fuzzing to generate different inputs for target functions to boost coverage of the associated CFG. For each execution of a target function, BinSec exports a compact representation of a function-level executable, i.e., a compact binary representation of the file that can be executed dynamically using runtime DLL binary injection, as well as the associated inputs that triggered that execution. This allows the dynamic analysis execution engine to efficiently execute the target function. This implies that BinSec will use a fixed execution environment.

Execution environment selection. Before BinSec begins to instrument the target function execution, BinSec uses one fixed execution environment to perform execution on a large number of candidate functions. Once an execution environment is chosen, there are several possible outcomes after we start to run a target function, `f`. For example, the candidate `f` may terminate, the candidate `f` may trigger a system exception, or the candidate `f` may go into an infinite loop. If the candidate `f` triggers a system exception, we will remove the candidate function from candidate set. Once this execution environment is selected, the target function execution can be instrumented.

Target function instrumentation. The output of the dynamic analysis engine for a function `f` in a fixed execution environment is a feature vector v(f, env) of length N.

Index	Feature Name	Feature Description
1	binary_defined_fun_call_num	number of binary-defined function calls during execution
2	min_stack_depth	the minimal stack depth during execution
3	max_stack_depth	the maximal stack depth during execution
4	avg_stack_depth	the average stack depth during execution
5	std_stack_depth	the standard deviation stack depth during execution
6	instruction_num	number of executed instruction
7	unique_instruction_num	number of executed unique instruction
8	call_instruction_num	number of call instruction
9	arithmetic_instruction_num	number of arithmetic instruction
10	branch_instruction_num	number of branch instruction
11	load_instruction_num	number of load instruction
12	store_instruction_num	number of store instruction
13	max_branch_frequency	the maximal number of frequency of the executed same branch instruction
14	max_arith_frequency	the maximal number of frequency of the executed same arithmetic instruction
15	mem_heap_access	number of accessing heap memory space
16	mem_stack_access	number of accessing stack memory space
17	mem_lib_access	number of accessing library memory space
18	mem_anon_access	number of accessing anonymous mapping memory space
19	mem_others_access	number of accessing others part memory space
20	library_call_num	number of library function calls during execution
21	syscall_num	number of system calls during execution
22	time_exe	the time of execution

Table 5.2: Dynamic features used in BinSec.

In order to generate the feature vector, BinSec traces the function execution. For the actual dynamic analysis, a wealth of systems are available such as debuggers, emulators, and virtual machines. However, because of the heterogeneity of mobile/IoT firmware architectures and platforms, BinSec utilizes an instrumentation tool that supports a variety of architectures and platforms accordingly.

Once BinSec can instrument individual functions and record trace information during execution, we can extract particular features that capture a variety of instruction information (e.g., number of instructions), system level information (e.g., memory accesses), as well as higher level attributes such as function and system calls. Table 5.2 shows the initial set of features we initially considered and eventually proved to be useful for establishing function binary similarity. However, this feature list is not comprehensive and can easily be extended.

For each execution, the dynamic engine will generate a set of observations for each feature, e.g., in the above case there will be 22 sets of observations. Once all instructions for a function `f` have been covered, BinSec combines the observations into a single vector, e.g., ($\texttt{f}_{\texttt{input\_1}}$). The same process is repeated for different inputs for the same function to produce ($\texttt{f}_{\texttt{input\_2}}$),($\texttt{f}_{\texttt{input\_3}}$)),...,($\texttt{f}_{\texttt{input\_N}}$). Now that we have the capability of extracting both static and dynamic features of a target function, we need to design an algorithm for calculating function similarity for a given pair of functions

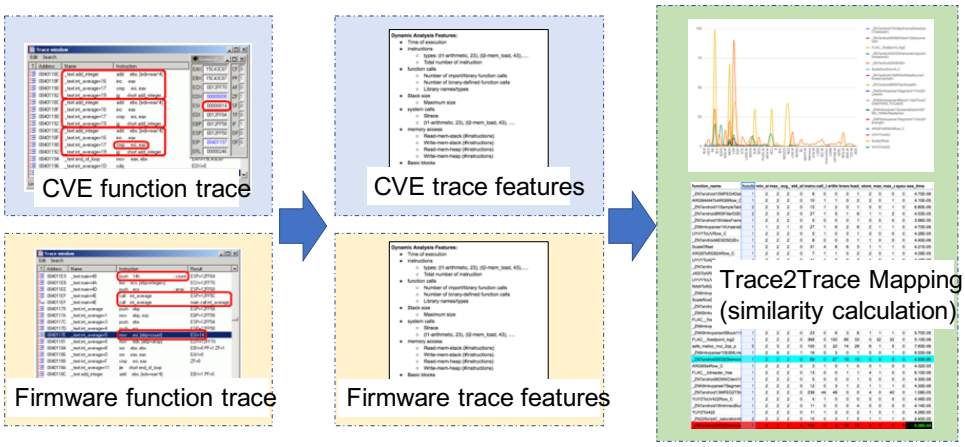


Figure 5.5: Detailed BinSec architecture for dynamic analysis of the mobile/IoT firmware that produces the true positive report and discards false positives.

and their extracted feature sets.

5.3.3 Calculating Function Similarity

For each function pair, (f,g), BinSec computes a similarity measure based on the *distance* between the two functions. Distance has been used in data mining contexts with dimensions representing features of the objects. In particular, BinSec uses the Euclidean distance, Manhattan distance, and Minkowski distances as our similarity measures based on each functions feature vector. Different behaviors result in slightly different values of the corresponding coordinates in the feature vectors. We now explore each distance measure in detail.

The **Euclidean distance** metric is a distance measure between two points or vectors in a two-dimensional or multidimensional (Euclidean) space based on Pythagoras' theorem. The distance is calculated by taking the square root of the sum of the squared pair-wise distances of every dimension, i.e.,

$$d_{euclidean} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}. \tag{5.1}$$

The **Manhattan distance** (sometimes referred to as the Taxicab distance) metric is similar to the Euclidean distance, but instead of calculating the shortest diagonal

path between two points, it calculates the distance based on gridlines. The Manhattan distance was named after the block-like layout of the streets in Manhattan. The equation is as follows,

$$d_{manhattan} = \sum_{i=1}^{n} |x_i - y_i|. \tag{5.2}$$

The **Minkowski distance** is a generalized form of the Euclidean distance (if p=2) and ofthe Manhattan distance (if p=1), i.e.,

$$d_{minkowski} = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}. \tag{5.3}$$

We feed the feature vector of each function into all three metrics and use these as our similarity measures. This is the final component for the identification of known vulnerabilities. We now design the final component that allows us to perform patch presence detection.

5.3.4 Patch Detection

We noticed that a patch typically introduces few changes to a vulnerable function. However, these minor changes can still have a significant impact to make the pre- and post-patch functions dissimilar[1]. Based on this notion, BinSec uses a *differential engine* to collect both static and dynamic similarity measures in order to determine if a vulnerable function has been patched.

Given a vulnerable function f_v, a patched function f_p, and a target function f_t, the differential engine will first generate three values: the *semantic static features* of f_v, f_p, and f_t, and the *dynamic similarity scores* of f_v vs. f_t and f_p vs. f_t, as well as the *differential signature* between f_v and f_p,. The semantic static features are the same aforementioned 48 different quantified features and the dynamic similarity scores are the aforementioned function similarity metrics. The differential signatures are an additional metric that compares the CFG structures, i.e., the CFG topology, of two

[1]This intuition is confirmed in Section 5.5.

functions as well as semantic information, e.g., function parameters, local variables, and library function calls. We now detail the implementation of BinSec's design.

5.4 Implementation and Case-Study

We implemented the BinSec framework on Ubuntu 18.04 in its AMD64 flavor. Our experiments are conducted on a server equipped with one Intel Xeon E51650v4 CPU running at 128 GB memory, 2TB SSD, and 4 NVIDIA 1080 Ti GPU. During both training and evaluation, 4 GPU cards were used. As in the design, BinSec consists of four main components: a feature extractor, a deep learning model, a dynamic analysis engine, and a differential analysis engine for patch detection.

5.4.1 Feature Extractor

The input for the feature extractor is the disassembled binary code of the target function. We assume the availability and the correctness of function boundaries by building on top of IDA Pro Hex-Rays, a commercial disassembler tools used for extracting binary program features. As such, we implemented the feature extractor as a plugin for IDA Pro. We developed two versions of the plugin: a GUI-version and command line-version (for automation). Since BinSec works on cross-platform binaries, the plugin can support different architectures (x86, amd64, ARM 32/64 bit and MIPS) for feature extraction.

5.4.2 Deep Learning Model

We implement the neural network model based on Keras Chollet et al. (2015) and TensorFlow Abadi et al. (2016). We use TensorBoard ten to visualize the whole training procedure.

5.4.3 Dynamic Analysis Engine

As was mentioned in the design section, the key challenges for dynamic analysis are the preparation of the inputs for the engine as well as the instrumentation of target

functions for tracing dynamic information.

Input preparation. As was mentioned in Section 5.3.2, BinSec needs to efficiently prepare the execution environment. To perform dynamic analysis without having to load the entire binary, we utilize DLL injection to execute compact execution binaries that correspond to a single target function. In particular, we use the Linux function dlopen() to load the dynamic shared object binary file which returns an opaque "handle" for the loaded object. This handle is employed with other useful functions in the dlopen API, such as dlsym. Using dlsym, we can directly find the exported functions based on the exported function's name. We can then execute the targeted function.

Of course, a library binary will contain a large amount of different functions, some of which are non-exported functions. As such, we must find a way to export these functions for further analysis. BinSec uses the LIEF Quarkslab (2017-2018) project to export functions into executable binaries. Such a transformation allows BinSec to instrument a candidate function that was found at a given address by using dlopen/dlsym. Thus, any candidate function can be exported and executed without running the whole binary. This approach has excellent reliability and efficiency since we can focus on targeted function without having to spawn the entire binary. Furthermore, we use LibFuzzer Infrastructure (2017) to fuzz candidate functions and generate different input sets for the execution environment.

Instrumentation. Because we are targeting heterogeneous mobile/IoT ecosystems, we choose to implement the same instrumentation of BinSec on two dynamic instrumentation frameworks: IDA Pro and GDB. For example, we implement a plugin based on GDB and GDBServer for Android and Android Things platforms, and we implement a plugin based on IDA Pro and debugserver for IOS platforms.

5.4.4 Case Study

To facilitate the understanding of our implementation details, we will provide an ongoing example to show how we can locate a known CVE vulnerability and how we can ensure whether the vulnerability has been patched or not patched in one IoT device firmware (Android Things). Android Things is an Android-based embedded operating

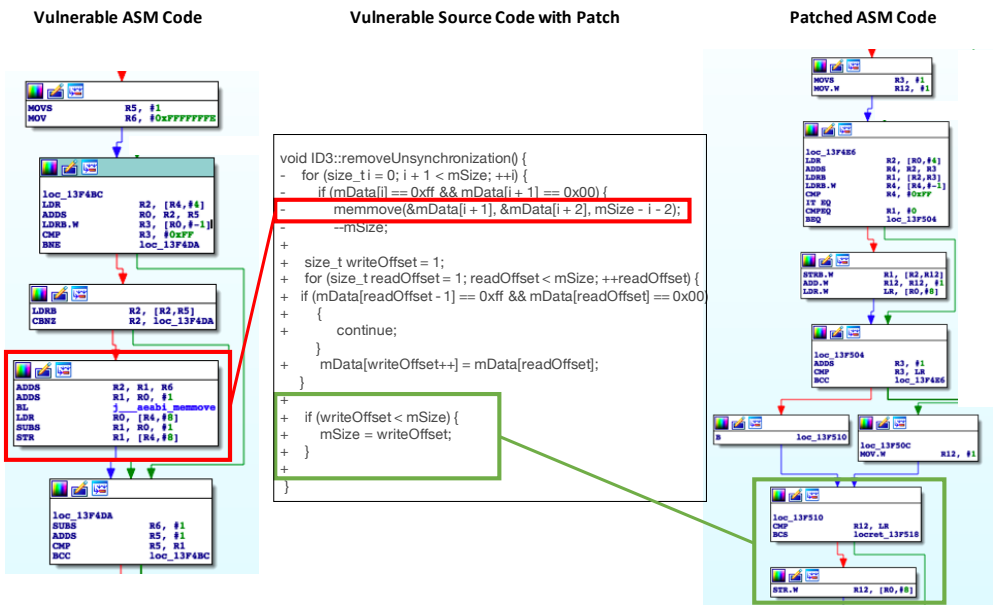


Figure 5.6: Vulnerable code with the associated patch of CVE-2018-9412.

system platform by Google.

Known CVE vulnerability function discovery. We chose one CVE vulnerability, `CVE-2018-9412`, from Android Security Bulletins Project. This is a DoS vulnerability in the function `removeUnsynchronization` of the library `libstagefright`. In order to simplify the case study, we generated these binaries directly from the source codes of both the vulnerable and patched `libstagefright` libraries. We compiled both versions using Clang with optimization level O0.Although BinSec never uses the source code for it's analysis, Figure 5.6 shows the source code and assembly code of the patched `CVE-2018-9412` for illustration. We elaborate on the components of this figure in the following subsection.

Generating a training dataset. We compiled 100 Android libraries from their source code using Clang. The compiler is set to emit code in x86, amd64, ARM 32bit and ARM 64bit with optimization levels O0, O1, O2, O3, Oz, Ofast. In total, we obtain 2108[2] library binary files. We provide more details in Section 5.5.

Feature Extraction. We use our feature extraction plugin to extract the features

[2]Some compiler opitimization levels didn't work for certain instances.

on top of IDA Pro. Once we get the raw features, BinSec will refine the raw features to generate the feature vector. BinSec extracted all function features from `libstagefright.so` and identified a total of 5,646 functions and generated 5,646 function feature vectors.

Detection by deep learning. Once the features are extracted, we use the training model for detection. We also use the vulnerable and patched functions as a baseline. Our model identified 252 candidate functions which are based on vulnerable function's feature vector while generating 971 candidate functions based on the patched function's feature vector. We also compared the feature vectors of the vulnerable and patched functions to check whether they are similar and found them to be dissimilar–meaning the patched version has significantly different features than the vulnerable version. Looking at the source code in Figure 5.6, one can intuit that the patched version is significantly different. For instance, the patch removed the `memmove` function and added one more `if` condition for value checking. Similarly, one can observe the difference in the number of basic blocks at the assembly level.

Dynamic analysis engine. Not only are the numbers of candidate vulnerable functions (252) and patched functions (971) from the last step very large, but the candidate functions are also very similar. As such, it would be difficult to locate the target vulnerability function by manual inspection. We therefore use the dynamic analysis engine to generate dynamic information for each function. We first use libfuzzer to generate the different inputs for the vulnerability function `removeUnsynchronization`. We tested that these inputs worked witfor both the vulnerable and patched functions. As before, we use the input to test each candidate function and remove any functions that crashed. Using the input-function validation, we obtain 38 candidate functions for the vulnerable function and 327 candidate functions for the patched function. For these candidate functions, BinSec's dynamic analysis engine will generate the dynamic information. For instrumentation in Android Things, we use gdbserver to collect the dynamic features on the Android Things devcie. Table 5.3 shows the final dynamic feature vector of a candidate vulnerable function. In the next subsection, analyze why `candidate_29` is the vulnerable function.

Candidate	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_10	F_11	F_12	F_13	F_14	F_15	F_16	F_17	F_18	F_19	F_20	F_21	F_22
candidate_1	1	2	2	2	0	12	12	0	0	0	4	0	0	0	0	0	0	3	0	0	0	4.25E-05
candidate_2	2	2	3	2	0	12	12	1	1	0	2	1	0	1	0	0	1	2	0	0	0	4.93E-05
candidate_3	1	1	2	1	0	29	24	1	5	2	8	2	1	2	0	4	2	2	2	0	0	4.20E-06
candidate_4	1	2	2	2	0	8	8	0	0	0	1	2	0	0	0	0	0	3	0	0	0	4.70E-06
candidate_5	1	2	2	2	0	10	9	1	1	0	2	2	0	1	0	0	0	0	0	0	0	4.10E-05
candidate_6	2	2	3	2	0	12	12	1	2	0	1	0	0	1	0	0	1	0	0	0	0	6.80E-06
candidate_7	2	2	3	2	0	27	25	1	5	1	9	1	1	2	0	1	3	4	1	0	0	4.53E-05
candidate_8	1	2	2	2	0	5	5	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3.96E-05
candidate_9	1	1	2	1	0	27	25	1	6	2	9	2	1	1	0	7	1	1	2	0	0	4.70E-06
candidate_10	1	2	2	2	0	5	5	1	0	0	1	0	0	0	0	0	0	0	0	0	0	4.28E-05
candidate_11	1	2	2	2	0	8	8	0	0	0	1	1	0	0	0	0	0	2	0	0	0	4.40E-06
candidate_12	1	2	2	2	0	31	31	4	8	6	3	1	1	1	0	3	0	0	0	0	0	4.21E-05
candidate_13	1	2	2	2	0	7	6	1	1	0	2	2	0	1	0	0	0	0	0	0	0	4.09E-05
candidate_14	1	2	2	2	0	11	11	1	2	0	2	1	0	1	0	3	0	0	0	0	0	4.15E-05
candidate_15	1	2	2	2	0	8	7	0	0	0	1	1	0	0	0	0	0	2	0	0	0	4.40E-06
candidate_16	1	2	2	2	0	6	5	1	0	0	1	2	0	0	0	0	0	0	0	0	0	4.25E-05
candidate_17	1	2	2	2	0	4	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4.19E-05
candidate_18	1	2	2	2	0	4	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4.20E-05
candidate_19	1	2	2	2	0	8	8	0	0	0	2	0	0	0	0	0	0	2	0	0	0	4.01E-05
candidate_20	1	2	6	4	2	6	5	1	1	1	0	0	1	1	0	0	0	0	0	0	0	4.40E-05
candidate_21	1	2	2	2	0	12	12	0	1	1	1	1	1	1	0	0	0	2	0	0	0	4.20E-06
candidate_22	1	2	2	2	0	30	28	1	8	1	10	1	1	1	0	2	1	4	4	0	0	5.40E-06
candidate_23	1	2	2	2	0	5	5	0	0	1	0	0	1	0	0	0	0	0	0	0	0	4.00E-06
candidate_24	1	1	2	1	0	30	26	1	6	2	9	2	1	2	0	5	2	2	2	0	0	4.40E-06
candidate_25	1	2	2	2	0	23	21	0	6	3	8	1	1	1	0	2	2	2	2	0	0	5.70E-06
candidate_26	1	2	2	2	0	368	28	0	100	66	33	0	32	32	0	0	0	0	33	0	0	5.10E-06
candidate_27	5	2	3	2	0	100	88	3	22	14	29	6	1	5	0	4	6	11	1	0	0	7.60E-06
candidate_28	1	2	6	2	1	16	16	0	3	0	1	0	0	1	0	0	0	1	0	0	0	6.00E-06
candidate_29	1	2	2	2	0	89	17	0	27	19	19	0	9	9	0	0	0	10	0	1	0	4.33E-07
candidate_30	1	2	2	2	0	3	3	1	0	1	0	0	1	0	0	0	0	0	0	0	0	4.32E-05
candidate_31	1	2	2	2	0	13	13	0	0	1	1	4	1	0	0	0	0	5	0	0	0	6.10E-06
candidate_32	1	2	2	2	0	5	5	0	0	0	1	0	0	0	0	0	0	1	0	0	0	4.30E-06
candidate_33	1	2	2	2	0	12	12	0	3	1	2	1	1	1	0	2	0	0	0	0	0	4.30E-06
candidate_34	1	2	2	2	0	238	17	44	48	0	0	4	0	40	0	0	0	0	4	0	0	1.08E-05
candidate_35	1	2	2	2	0	4	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4.46E-05
candidate_36	1	2	2	2	0	11	11	0	0	0	4	0	0	0	0	0	0	4	0	0	0	4.14E-05
candidate_37	1	2	2	2	0	11	11	1	2	0	2	1	0	1	0	3	0	0	0	0	0	4.26E-05
candidate_38	2	2	2	2	0	15	15	0	3	1	5	1	1	1	0	1	2	2	1	0	0	4.40E-05
Vulnerable function	1	2	2	2	0	122	21	0	9	18	19	0	9	9	0	0	0	10	0	1	0	4.13E-05

Table 5.3: The dynamic feature vector profiling for candidate functions of the vulnerable version of `removeUnsynchronization` in the library `libstagefright.so`. F_1,...,F_21 represents different dynamic features 1 to 21 showed in Table 5.2. In the last row, the vulnerable function is from our vulnerability database.

Calculating Function Similarity. We use the three aforementioned three similarity metrics to calculate the function similarity. The results for the vulnerable function are listed in Table 5.4 and the results for the patched functions are listed Table 5.5. For the vulnerable function results, we see that `candidate_29` is the top ranked candidate across all three metrics, i.e., according to the rule of Euclidean distance (as well as the other two distances), if this distance is small, there will be a high degree of similarity. We can also see a significant difference between the top candidate and second candidate (`candidate_27`). As such, we conclude that `candidate_29` is the vulnerable function.

Diving deeper into the results in Table 5.3, we can observe why the distance between the dynamic features is so small. The two highlighted rows indicate `candidate_29` and the ground truth vulnerable function. Referring back to Table 5.2, we know that *F_13* represents the max frequency for the same branch instruction and *F_14* represents the max frequency for the same arithmetic instruction. We can that `candidate_29` is the only candidate function that has the same frequency numbers as the vulnerable function. It is important to note that this analysis was only enabled by dynamic analysis–static analysis would not have been able to identify these dynamic features.

For the patched case, Table 5.5 only shows the results for the top 40 candidate functions due to page limitations. In this case `candidate_102` on average is the top ranked candidate despite being the incorrect function. However, we can see that **candidate_29** is ranked in a very close second, while there is a significant difference with the third candidate. Intuitively, we can narrow down the candidate functions to the top two and can assume that `candidate_29` is likely to be the associated candidate vulnerable function. However, at this point we cannot tell whether the function is patched.

Candidate	EUD	MAD	MID	Ground truth
candidate_29	**37.8**	**56.0**	**34.7**	**_ZN7android3ID323removeUnsynchronizationEv**
candidate_27	73.9	154.0	68.1	safe_malloc_mul_2op_p
candidate_12	95.1	164.0	91.4	ScaleOffset
candidate_22	95.2	157.0	92.3	_ZNK9mkvparser7Segment11DoneParsingEv
candidate_24	95.2	163.0	92.3	_ZN9mkvparser15UnserializeUIntEPNS_10IMkvReaderExx
candidate_3	96.2	163.0	93.3	_ZN9mkvparser6ReadIDEPNS_10IMkvReaderExRl
candidate_7	98.0	160.0	95.3	_ZN7android8RSFilterD2Ev
candidate_9	98.4	168.0	95.3	_ZN9mkvparser14UnserializeIntEPNS_10IMkvReaderExxRx
candidate_25	101.8	159.0	99.2	_ZNK9mkvparser5Block11GetTimeCodeEPKNS_7ClusterE
candidate_28	110.4	184.0	106.4	_ZN9mkvparser10EBMLHeader4InitEv
candidate_38	110.5	180.0	107.3	_ZN22ScriptC_saturationARGBD2Ev
candidate_31	113.2	187.0	109.4	FLAC__bitreader_free
candidate_1	114.1	186.0	110.3	_ZN7android8RSFilter5resetEv
candidate_33	114.1	188.0	110.4	_ZNK9mkvparser7Segment11FindClusterEx
candidate_21	114.2	187.0	110.4	_ZN7android13MPEG2TSWriter10SourceInfo26incrementContinuityCounterEv
candidate_2	114.3	192.0	110.4	_ZN7android3ID3D2Ev
candidate_6	114.6	193.0	110.4	_ZN7android11SampleTable22CompositionDeltaLookupC2Ev
candidate_36	115.1	187.0	111.3	_ZN7android19IntrinsicBlurFilter5resetEv
candidate_14	115.5	195.0	111.4	UYVYToI422
candidate_37	115.5	195.0	111.4	YUY2ToI422
candidate_5	116.7	197.0	112.4	ARGB4444ToARGBRow_C
candidate_19	118.7	197.0	114.4	_ZN9mkvparser8SeekHeadD2Ev
candidate_4	118.8	199.0	114.4	_ZN7android15MPEG4DataSource10clearCacheEv
candidate_11	118.8	199.0	114.4	_ZN7android4ESDSD2Ev
candidate_15	118.9	200.0	114.4	_ZN7android10MidiEngine14releaseBuffersEv
candidate_13	119.9	203.0	115.4	ARGBToRGB24Row_C
candidate_20	121.2	211.0	116.5	ScaleRowDown4_C
candidate_16	121.3	208.0	116.5	J400ToARGBRow_C
candidate_32	122.1	205.0	117.5	_ZN7android9OMXClient10disconnectEv
candidate_8	122.2	206.0	117.5	_ZN7android19VideoFrameScheduler7releaseEv
candidate_23	122.2	205.0	117.5	FLAC__fixed_restore_signal
candidate_10	122.2	207.0	117.5	UYVYToUVRow_C
candidate_18	123.5	210.0	118.5	RAWToRGB24Row_C
candidate_17	123.5	210.0	118.5	UYVYToUV422Row_C
candidate_35	123.5	210.0	118.5	YUY2ToUV422Row_C
candidate_30	124.3	210.0	119.5	ARGBSetRow_C
candidate_34	137.1	298.0	120.5	_ZN7android13MPEG2TSWriter12initCrcTableEv
candidate_26	271.3	495.0	251.0	FLAC__fixedpoint_log2

Table 5.4: Calculating Function Similarity in BinSec for CVE-2018-9412 in Android Things based on **vulnerable** function (EUD: Euclidean distance, MAD: Manhattan distance, MID: Minkowski distance)

Differential Analysis Engine. According the previous steps, we can consider **candidate_29** is the target function. But it is still not clear whether it is patched. We collect the differential signatures (`j___aeabi_memmove`, `if` condition), semantic static features (`j___aeabi_memmove`), and dynamic similarity scores (`37.8` V.S. `69.8`). Based on these metrics, the differential analysis engine concludes the target function is still vulnerable and not patched.

Candidate	EUD	MAD	MID	Ground truth
candidate_102	44.7	133.0	32.8	CanonicalFourCC
candidate_29	**68.9**	**113.0**	**65.6**	**_ZN7android3ID323removeUnsynchronizationEv**
candidate_52	95.8	191.0	91.4	_ZN7android11MPEG4Writer13writeLatitudeEi
candidate_76	96.8	192.0	92.4	_ZN7android11MPEG4Writer14writeLongitudeEi
candidate_85	105.6	218.0	96.7	_ZN7android21ElementaryStreamQueueC2ENS0_4ModeEj
candidate_93	110.7	277.0	86.8	__divdf3
candidate_101	114.8	234.0	106.3	_ZN7android10MediaMuxerC2EiNS0_12OutputFormatE
candidate_40	118.6	241.0	109.5	ARGBToARGB4444Row_C
candidate_66	121.8	249.0	113.2	CopyPlane
candidate_111	126.3	255.0	116.7	_ZN7android10WebmWriter16estimateCuesSizeEi
candidate_92	127.3	240.0	119.3	_ZN7android10WebmWriter9numTracksEv
candidate_106	128.2	246.0	119.6	__floatdidf
candidate_54	129.6	229.0	123.0	_ZN9mkvparser10VideoTrackC2EPNS_7SegmentExx
candidate_116	130.5	230.0	124.0	_ZN9mkvparser10AudioTrackC2EPNS_7SegmentExx
candidate_95	130.6	250.0	123.0	_ZN7android23StagefrightMediaScanner15extractAlbumArtEi
candidate_65	135.3	254.0	127.2	ScaleRowDown34_Any_NEON
candidate_78	135.5	253.0	127.3	ScaleRowDown34_0_Box_Any_NEON
candidate_105	136.0	256.0	127.4	_ZN7android19SampleConverterBase10targetSizeEj
candidate_49	136.9	255.0	129.1	InitCpuFlags
candidate_83	139.1	263.0	129.7	ARGBSepiaRow_NEON
candidate_88	139.5	254.0	132.8	_ZN7android12CameraSource19stopCameraRecordingEv
candidate_60	141.1	260.0	133.9	_ZN7android9AMRWriter5resetEv
candidate_70	141.2	260.0	132.5	__aeabi_dadd
candidate_41	141.4	262.0	133.2	_ZN7android10WebmWriter5pauseEv
candidate_48	141.5	256.0	133.3	__floatdisf
candidate_103	142.9	263.0	135.9	_ZN7android6ACodec13FlushingState28changeStateIfWeOwnAllBuffersEv
candidate_87	143.5	266.0	136.0	FLAC__bitreader_skip_byte_block_aligned_no_crc
candidate_108	143.9	265.0	136.1	_ZN7android6ACodec10BufferInfo14checkReadFenceEPKc
candidate_47	144.1	272.0	135.3	CopyPlane_16
candidate_84	146.2	269.0	138.1	ARGBGrayRow_C
candidate_38	147.8	271.0	140.0	_ZN22ScriptC_saturationARGBD2Ev
candidate_57	148.4	272.0	140.2	_ZNK7android17MyVorbisExtractor13approxBitrateEv
candidate_75	148.7	274.0	141.0	ARGBUnattenuateRow_C
candidate_89	148.8	275.0	140.3	_ZN7android19SampleConverterBase10sourceSizeEj
candidate_55	148.9	274.0	140.3	__aeabi_fadd
candidate_115	149.7	276.0	141.2	_ZN9mkvparser8Chapters4Atom5ClearEv
candidate_53	150.5	285.0	141.4	ARGBComputeCumulativeSum
candidate_114	151.6	280.0	143.2	_ZN7android11AudioSource31waitOutstandingEncodingFrames_lEv
candidate_99	151.9	281.0	143.2	_ZN7android14MPEG4Extractor11countTracksEv
candidate_42	152.3	286.0	143.3	ARGBSepia

Table 5.5: Calculating Function Similarity in BinSec for CVE-2018-9412 in Android Things based on **patched** function

5.5 Evaluation

In this section, we evaluate BinSec with respect to its search accuracy and computation efficiency. In particular, we evaluate the accuracy of our deep learning model, dynamic analysis engine and differential analysis engine using a dataset containing ground truth.

5.5.1 Data preparation

In our evaluation, we collected three datasets: 1) Dataset I for training the deep learning model and evaluating the accuracy of the deep learning model; 2) Dataset II for collecting known CVE vulnerabilities and to build our vulnerability database. 3) Dataset III for evaluating the accuracy and performance of the deep learning model, dynamic analysis engine and differential analysis engine for real world mobile/IoT firmware;

Dataset I: This dataset is used for neural network training and baseline comparison. It consists of binaries compiled from source code, providing us with the ground truth. We consider two functions compiled from the same source code function are similar, and dissimilar if they are from different functions. In particular, we compile 100 Android libraries from their source code (version android-8.1.0_r36) using Clang. We exported 24 different binaries for each Android library by setting the compiler to emit code in x86, AMD64, ARM 32bit, and ARM 64bit ISA with optimization levels O0, O1, O2, O3, Oz, Ofast. However, not every library could be compiled with the six optimization levels, e.g., libbrillo, libbacktrace, libtextclassifier, and libmediaplayerservice. In total, we obtain 2108 library binary files containing 2,037,772 function feature samples. For this Dataset, we compiled all binaries with a debug flag to establish ground truth based on the symbol names. For our problem setting, we strip all binaries before processing them with BinSec.

Dataset II: Since we perform vulnerability assessement, we generated a vulnerability database which includes the static feature vectors and dynamic feature vectors for vulnerable versions and patched versions of functions. The vulnerable function dataset comes from Android Security Bulletins Project. We collect the vulnerabilities from 07/2016 to 11/2018. In total, there are 2,076 vulnerabilities, including 1,351 high

vulnerabilities and 381 critical vulnerabilities.

Dataset III: To evaluate BinSec, we collected different firmware images which included different versions of Android, Android Things, and IOS. In particular, we select two firmware images from Android Things 1.0 and Google Pixel 2 XL (Android 8.0) as our targets. For vulnerability detection, we considered the vulnerabilities which were patched in 2018 and focus on version 8.0 and 8.1. Finally, we choose 25 different CVE vulnerabilities from our database to evaluate our solution on Android Things and Google Pixel 2 XL.

5.5.2 Training details

Our deep learning model is first trained using Dataset I. We adapt the sequential model with 6 layers. We first specify the inputs for each layer. The first layer in our sequential model needs to receive information about its input shape. In our case, it is 96. We split Dataset I into three disjoint subsets of functions for training (1,222,663), validation (407,554), and testing (407,555), respectively.

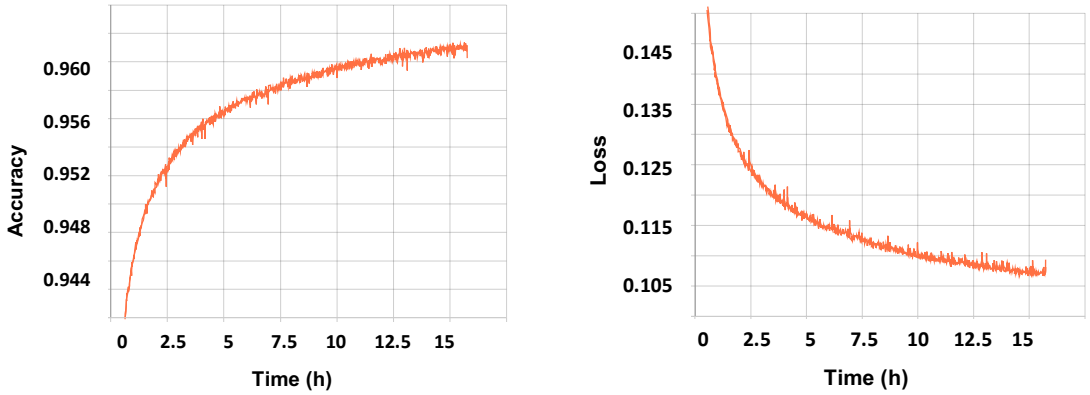
5.5.3 Testing devices

We evaluate BinSec in two different devices: Android Things and Google Pixel 2 XL. For Android Things, we use Android Things 1.0–which includes a 05/2018 security patch as well as a previous security patch. For Google Pixel 2 XL, its system version is Android 8.0 and it includes a 07/2017 security patch as well as a previous security patch.

5.5.4 Accuracy

In this section, we evaluate the accuracy of BinSec's deep learning model, dynamic analysis engine, as well as its patch detection.

Deep learning model. Figure 5.7a and Figure 5.7b show the accuracy and loss when we train our deep learning model for ˜15 hours. The accuracy can reach 96%.



(a) Accuracy for training the model

(b) Loss for training the model

Figure 5.7: Deep learning training result.

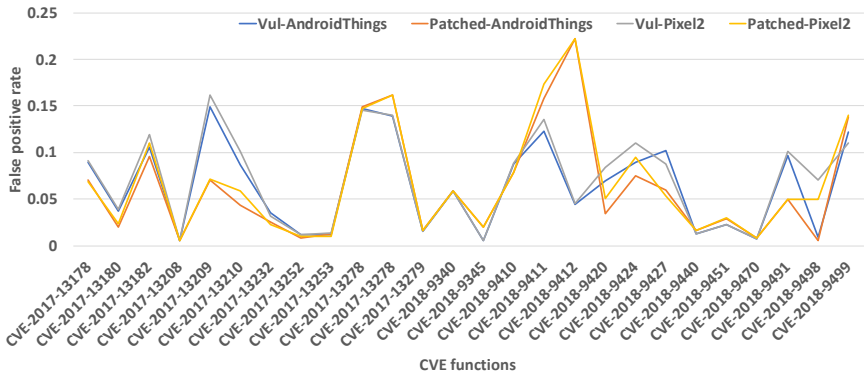


Figure 5.8: False positive rate on Android Things and Google Pixel 2 XL with vulnerable and patched versions.

Since Zhang and Qian (2018) needs the previous similarity checking solutions to locate the target function when the symbol table is not available, the target function may be missed if the vulnerable version and patched version are not similar. To validate this notion, we use our deep learning model to check the similarity between the vulnerable and patched version of the same function for 25 CVEs. We found that there are four CVEs (CVE-2018-9345, CVE-2017-13209, CVE-2018-9499 and CVE-2018-9412) whose vulnerable and patch versions are not similar based on the deep learning model. For example, if CVE-2018-9345 had been patched, the solution in Zhang and Qian (2018) will miss the target function based on vulnerable function features and, thus, may use the wrong function to detect whether it is patched.

We use the training model to detect 25 CVEs in Android Things and Google Pixel

2 XL. The average detection accuracy is more than 93%. Figure 5.8 shows the false positive rate when we test vulnerable and patched versions in the two devices' firmware images. It is interesting that the false positive rate of the vulnerable and patched versions of CVE-2017-13209 and CVE-2018-9412 on the two devices are obviously different, which is reflective of their dissimilar result. Furthermore, we notice that because CVE-2017-13209 has been patched, the false positive rate of patched version is lower than vulnerable version. Similarly, CVE-2018-9412 has not been patched and Figure 5.8 shows false positive rate of patched version is higher than vulnerable version. However, Table 5.6 shows the vulnerable version function gets 0 true positives and 1 false negative in Android Things for CVE-2017-13209. This is due to the fact that CVE-2017-13209 has been patched in Android Things. Therefore, when BinSec uses the vulnerable function, the deep learning may miss the correct target function. Intuitively, it makes sense that a known vulnerability discovery may miss a patched function as a vulnerability.

Dynamic analysis engine. The goal of the dynamic analysis engine is to prune the set of candidate functions. In Table 5.6 and Table 5.7, the results for the dynamic analysis engine includes only the **Execution** and **Ranking** metrics. Because we do not want to reduce the number of functions we perform dynamic feature profiling for, we use the concrete input of vulnerable functions to validate the candidate functions. As long as the candidate functions can survive the input validation, BinSec will do dynamic feature profiling for the final candidate function. For example, after deep learning, CVE-2018-9412 still has 252 candidate functions. For dynamic analysis, BinSec arranges different inputs of vulnerable function to validate the candidate functions. After validation, only 38 candidate functions remain that require dynamic feature profiling–which is a much more reasonable number. Finally, BinSec calculates the function similarity score. Table 5.6 and Table 5.7 show that BinSec can rank the target function in the top 3 candidates 100% of the time. The target function is only missed for CVE-2017-13209 since the deep learning model already misses the target function.

Patch detection. According to the differential signature, semantic static features, and the results from Table 5.6 and Table 5.7, BinSec generates the final results in

CVE	Deep Learning Classification						Dynamic Analysis Engine		Time(s)	
	TP	TN	FP	FN	Total	FP(%)	Execution	Ranking	DP	DA
CVE-2018-9451	1	1155	27	0	1183	2.28%	5	1	2.26	187.97
CVE-2018-9340	1	1113	69	0	1183	5.83%	6	1	2.14	197.56
CVE-2017-13232	1	951	35	0	987	3.55%	5	2	3.13	147.97
CVE-2018-9345	1	354	2	0	357	0.56%	1	1	2.72	41.13
CVE-2018-9420	1	107	8	0	116	6.90%	1	1	1.53	39.59
CVE-2017-13210	1	105	10	0	116	8.62%	2	1	1.57	73.18
CVE-2018-9470	1	1421	11	0	1433	0.77%	4	1	6.85	148.37
CVE-2017-13209	0	867	152	1	1020	14.90%	9	N/A	5.25	286.34
CVE-2018-9411	1	894	125	0	1020	12.25%	8	1	5.23	256.58
CVE-2017-13252	1	609	7	0	617	1.13%	7	2	3.35	227.15
CVE-2017-13253	1	609	7	0	617	1.13%	5	2	3.39	167.97
CVE-2018-9499	1	541	75	0	617	12.16%	6	1	2.56	210.35
CVE-2018-9424	1	561	55	0	617	8.91%	7	1	3.02	219.45
CVE-2018-9491	1	421	45	0	467	9.64%	3	1	1.19	108.78
CVE-2017-13278	1	2164	373	0	2538	14.70%	20	2	1.93	602.35
CVE-2018-9410	1	595	57	0	653	8.73%	22	1	2.76	671.46
CVE-2017-13208	1	178	1	0	180	0.56%	1	1	1.23	39.32
CVE-2018-9498	1	13598	130	0	13729	0.95%	7	1	5.90	227.15
CVE-2017-13279	1	723	11	0	735	1.50%	6	1	3.40	224.56
CVE-2018-9440	1	725	9	0	735	1.22%	4	1	2.06	156.52
CVE-2018-9427	1	1060	120	0	1181	10.16%	9	1	4.61	296.31
CVE-2017-13178	1	540	53	0	594	8.92%	15	1	2.01	473.89
CVE-2017-13180	1	571	22	0	594	3.70%	5	2	1.23	157.97
CVE-2018-9412	1	5393	252	0	5646	4.46%	38	1	3.54	1124.53
CVE-2017-13182	1	5050	595	0	5646	10.54%	72	3	3.16	2128.16
Average						6.16%			3.04	336.5844

Table 5.6: The accuracy for deep learning and dynamic execution for Android Things based on vulnerable function. Dp: Deep learning; DA: Dynanmic analysis

CVE	Deep Learning Classification						Dynamic Analysis Engine		Time(s)	
	TP	TN	FP	FN	Total	FP(%)	Execution	Ranking	DP	DA
CVE-2018-9451	1	1148	34	0	1183	2.87%	8	2	2.29	246.25
CVE-2018-9340	1	1113	69	0	1183	5.83%	6	1	2.07	197.56
CVE-2017-13232	1	961	25	0	987	2.53%	5	1	3.20	177.97
CVE-2018-9345	1	349	7	0	357	1.96%	4	3	1.66	148.37
CVE-2018-9420	1	111	4	0	116	3.45%	1	1	1.50	59.59
CVE-2017-13210	1	110	5	0	116	4.31%	2	1	1.63	91.19
CVE-2018-9470	1	1420	12	0	1433	0.84%	4	1	5.93	160.46
CVE-2017-13209	1	947	72	0	1020	7.06%	7	1	4.07	207.15
CVE-2018-9411	1	858	161	0	1020	15.78%	10	2	4.24	301.23
CVE-2017-13252	1	611	5	0	617	0.81%	6	1	2.33	230.56
CVE-2017-13253	1	608	8	0	617	1.30%	5	2	2.67	165.51
CVE-2018-9499	1	531	85	0	617	13.78%	9	3	2.57	287.65
CVE-2018-9424	1	570	46	0	617	7.46%	5	1	2.01	156.32
CVE-2018-9491	1	443	23	0	467	4.93%	1	1	2.20	45.93
CVE-2017-13278	1	2159	378	0	2538	14.89%	19	1	1.90	587.86
CVE-2018-9410	1	601	51	0	653	7.81%	21	1	2.83	651.45
CVE-2017-13208	1	178	1	0	180	0.56%	1	1	1.08	35.53
CVE-2018-9498	1	13647	81	0	13729	0.59%	6	1	4.89	243.3
CVE-2017-13279	1	722	12	0	735	1.63%	6	1	3.48	236.78
CVE-2018-9440	1	722	12	0	735	1.63%	5	2	4.84	175.52
CVE-2018-9427	1	1110	70	0	1181	5.93%	2	2	4.74	99.18
CVE-2017-13178	1	551	42	0	594	7.07%	13	1	2.86	390.89
CVE-2017-13180	1	581	12	0	594	2.02%	2	1	2.17	71.48
CVE-2018-9412	1	4391	971	0	5646	17.20%	327	2	3.52	8676.91
CVE-2017-13182	1	5103	542	0	5646	9.60%	42	1	3.15	1249.96
Average						5.67%			2.95	595.784

Table 5.7: The accuracy for deep learning and dynamic execution for Android Things based on patched function. Dp: Deep learning; DA: Dyanmic analysis

CVE	BinSec Result Patched (?)	Ground Truth Patched (?)
CVE-2018-9451	0	0
CVE-2018-9340	0	0
CVE-2017-13232	✓	✓
CVE-2018-9345	0	0
CVE-2018-9420	0	0
CVE-2017-13210	✓	✓
CVE-2018-9470	✓	0
CVE-2017-13209	✓	✓
CVE-2018-9411	0	0
CVE-2017-13252	✓	✓
CVE-2017-13253	✓	✓
CVE-2018-9499	0	0
CVE-2018-9424	0	0
CVE-2018-9491	0	0
CVE-2017-13278	✓	✓
CVE-2018-9410	0	0
CVE-2017-13208	✓	✓
CVE-2018-9498	0	0
CVE-2017-13279	✓	✓
CVE-2018-9440	0	0
CVE-2018-9427	0	0
CVE-2017-13178	0	0
CVE-2017-13180	✓	✓
CVE-2018-9412	0	0
CVE-2017-13182	✓	✓

Table 5.8: The final patch detection results for BinSec in Android Things

Table 5.8. There is only one missed classification for the patched version of CVE-2018-9470. The reason this classification was missed was due to the fact that that the only difference between vulnerable and patched version is one integer–which is a very minute and difficult patch to detect.

5.5.5 Efficiency

We evaluate the efficiency of BinSec for deep learning and dynamic analysis. For deep learning, it took around 15 hours to train the deep learning model and uses on average 3 seconds to finish the detection according to Table 5.6 and Table 5.7. For dynamic analysis, the costly time comes from two sources: the candidate function input validation and the dynamic feature profiling–which requires heavy instrumentation of the execution for each function. Based on our evaluation, the average time cost is around 466 seconds.

5.6 Related Work

In this section, we briefly survey the related work. We focus on approaches using code similarity for known vulnerability without source code. Other approaches for finding unknown vulnerability Avgerinos et al. (2011); Cha et al. (2015); Stephens et al. (2016); Chen et al. (2016) and source code based Kamiya et al. (2002); Jiang et al. (2007); Huo et al. (2016); Huo and Li (2017); Li et al. (2018) will not be discussed in this section. We divide the related work to traditional and machine learning based solutions.

The problem of testing whether two pieces of syntactically-different code are semantically identical has received much attention by previous researchers. A lot of traditional approaches based on the matching algorithm for the CFGs of functions have been proposed. Bindiff bin is based on the syntax of code for node matching. At a high-level, BinDiff starts by recovering the control flow graphs of the two binaries and then attempts to use a heuristic to normalize and match the vertices from the graphs. For Pewny et al. (2014), each vertex of a CFG is represented with an expression tree. Similarity among vertices is computed by using the edit distance between the corresponding expression trees. Regarding the literature of cross-platform binary similarity. Eschweiler et al. (2016) proposes a graph-based methodology. It used a matching algorithm on the CFGs of functions. The idea is to transform the binary code in an intermediate representation. For such a representation, the semantics of each CFG vertex is computed by using a sampling of the code executions using random inputs. Feng et al. (2016); Xu et al. (2017) extract feature representations from the control flow graphs and encodes them into graph embeddings to speed up the matching process. Ng and Prakash (2013); Khoo et al. (2013) are both based on static analysis techniques.

Comparing with static analyses, dynamic analysis is another approach to detect function similarity. Egele et al. (2014) proposed a dynamic equivalence testing primitive that achieves complete coverage by overriding the intended program logic. It collects the side effects of functions during execution under a controlled randomized environment. Two functions can be similar if their side effects are similar. However, it needs to

execute each function with many different inputs. Under such a huge search space, it will take a long time to finish the similarity checking.

Deep learning-based graph embedding approaches have also been used to do binary similarity checking. Xu et al. (2017) are currently the state-of-the-art work on cross-platform vulnerability search. Their approach looks for the same affected functions in the complete collection of functions in a target binary. However, when the search space is huge, it still leaves a large set of candidate functions. BinSec can integrate dynamic analysis to prune the candidate functions and reduce the false positives.

Zhang and Qian (2018) proposed a unique position that leverages the source-level information to answer a more specific question: whether the specific affected function is patched in the target binary. However, it needs the source code support as well as the aforementioned similarity checking solutions to help it to locate the target function. BinSec uses deep learning and dynamic analysis to locate the target function and perform accurate patch detection.

5.7 Discussion and Conclusion

In this paper, we presented BinSec, a vulnerability assessment tool which leverages deep learning and dynamic analysis to do cross-platform binary code similarity detection to identify known vulnerabilities with high accuracy. BinSec then uses a differential engine to distinguish between vulnerable functions and patched functions. We evaluated BinSec on 25 existing CVE vulnerability functions for the Google Pixel 2 smartphone and Android Things IoT firmware images while emulating a heterogeneous ecosystem, i.e., we compiled the firmware images for multiple architectures and platforms with different compiler optimization levels. Our deep learning model identifies vulnerabilities with an accuracy of over 93%, i.e., higher than the state-of-the-art.

We also demonstrated how dynamic analysis of the vulnerability functions in a controlled environment can be used to significantly reduce the number of candidate functions and, thus, the number of false positives. BinSec identifies the correct matches (candidate functions) among the top 3 ranked outcomes 100% of the time. Finally,

we evaluate BinSec's differential engine that distinguishes between functions that are vulnerable and functions that are patched on the same dataset with the same level of accuracy.

Chapter 6

Conclusion

The promising results motivate scalable and robust solutions for exploring semantic reverse engineering as well as how to use the semantic reverse engineering results to provide better protection for systems. This dissertation narrowed the gap between practical and theoretical approaches in semantic reverse engineering of stripped off-the-shelf software binaries.

To explore data structures in a dynamic memory space, the dissertation presented ReViver, a hybrid data structure reverse engineering solution that took the memory image for a selected running process on the user's machine, and determined its semantic data structure layout without the need for execution traces before the memory capture point. The dissertation provided the design of how ReViver performed the forensics analysis of the captured memory dump, in order to finally generate the exact memory data structure layout.

According to data structure instances in live memory, the dissertation showed and evaluated a domain-specific attack against the popular and widely-used power grid economic dispatch control algorithm. The presented attack searched the controller's live memory for sensitive power grid parameters and modified them maliciously. In the end, the dissertation clearly showed how to replace the legitimate values with the adversary-optimal values that were calculated considering the physical system dynamics.

As control system has been used widely, it has been one urgent topic to understand the control algorithm in control system. The dissertation presented Mismo, a general framework to extract semantic information of an embedded firmware binaries with respect to its associated high-level control algorithm. The work had been evaluated on 2,263 commercial firmware binaries by 30 industry vendors from 6 real-world IoT/ICS

application domains. The dissertation also showed how to use Mismo to discover a zero-day vulnerability in the most recent Linux Kernel, and provided fine-grained protection of sensitive data on a self-driving automobile application.

Finally, this dissertation presented BinSec to provide a more practical approach to integrate semantic information for vulnerability detection. BinSec leveraged deep learning and dynamic analysis to do cross-platform binary code similarity detection to identify known vulnerabilities with high accuracy. Then it used a differential engine to distinguish between vulnerable functions and patched functions. In the end, the dissertation showed that BinSec had been evaluated on 25 existing CVE vulnerability functions for the Google Pixel 2 smart phone and Android Things IoT firmware images while emulating a heterogeneous ecosystem. The accuracy of identifying vulnerabilities is much higher than the state-of-the-art.

Bibliography

Bindiff; available at https://www.zynamics.com/software.html.

Clang static analyzer; available at http://clang-analyzer.llvm.org/.

Dataflowsanitizer; available at http://clang.llvm.org/docs/DataFlowSanitizerDesign.html.

A simulink model to learn the kalman filter for gaussian processes; available at http://www.mathworks.com.

Ollydbg; available at http://www.ollydbg.de.

Snowman: A native code to c/c++ decompiler; available at http://derevenets.com/.

Tensorboard; available at https://www.tensorflow.org/guide/summaries_and_tensorboard.

Powertools; available at http://hhijazi.github.io/PowerTools/, 2017.

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

A. Abur and A. Expósito. *Power System State Estimation: Theory and Implementation*. Marcel Dekker, 2004. ISBN 9780824755706.

O. Alsac, N. Vempati, B. Stott, and A. Monticelli. Generalized state estimation. *IEEE Trans. on Power Systems*, 13(3):1069–1075, 1998.

K. Angrishi. Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets. *arXiv preprint arXiv:1702.03681*, 2017.

J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, pages 1092–1110, 2017.

A. R. Arasteh and M. Debbabi. Forensic memory analysis: From stack and code to execution history. *digital investigation*, 4:114–125, 2007.

J. Arrillaga and B. Smith. *AC-DC Power Systems Analysis*. The Institution of Electrical Engineers, 1998. ISBN 9780852969342.

M. Assante. Confirmation of a Coordinated Attack on the Ukrainian Power Grid. SANS Industrial Control Systems Security Blog, 2016.

T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. 2011.

T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. USENIX, 2014.

F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

C. Betz, 2015. Memparser: Memparser Analysis Tool; available at `http://www.dfrws.org/2005/challenge/memparser.shtml`.

D. Bienstock. *Electrical transmission system cascades and vulnerability - an operations research viewpoint*, volume 22 of *MOS-SIAM Series on Optimization*. SIAM, 2016.

D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Computer aided verification*, pages 463–469. Springer, 2011.

C. Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Report from the Digital Forensic Research Workshop (DFRWS)*, 2006.

C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008a. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008b.

M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

A. A. Cardenas, S. Amin, and S. Sastry. Secure control: Towards survivable cyber-physical systems. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*, pages 495–500. IEEE, 2008.

A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *digital investigation*, 5:S65–S75, 2008.

O. Certik et al. Sympy python library for symbolic mathematics, 2008.

S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741. IEEE, 2015.

B. Chen, X. Liu, H. Zhao, and J. C. Príncipe. Maximum correntropy kalman filter. *Automatica*, 76:70–77, 2017.

D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. 2018.

S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security*, volume 5, 2005.

E. Chien, L. OMurchu, and N. Falliere. W32.Duqu - The precursor to the next Stuxnet. Technical report, Symantic Security Response, 2011.

E. J. Chikofsky, J. H. Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

H. Choi, W.-C. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 801–816. ACM, 2018.

F. Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7(8), 2015.

Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium, Security*, volume 17, 2017.

F. T. COMMISSION. Mobile security updates: Understanding the issues; available at `https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf`.

L. Constantin. Hackers found 47 new vulnerabilities in 23 iot devices at def con. *CSO Website*, 2016.

A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, pages 95–110, 2014.

A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *OSDI*, volume 8, pages 255–266, 2008.

A. Cui, M. Costello, and S. Stolfo. When firmware modifications attack: A case study of embedded exploitation. 2013.

D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

C. M. Davis and T. J. Overbye. Multiple element contingency screening. *Power Systems, IEEE Transactions on*, 26(3):1294–1301, 2011.

Department of Energy. Improving Efficiency with Dynamic Line Ratings; available at `https://www.smartgrid.gov/files/NYPA_Improving-Efficiency-Dynamic-Line-Ratings.pdf`, 2016a.

Department of Energy. Dynamic Line Rating Systems for Transmission Lines; available at `https://www.smartgrid.gov/files/SGDP_Transmission_DLR_Topical_Report_04-25-14_FINAL.pdf`, 2016b.

B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Usenix Security*, pages 303–317, 2014.

D. Emm, M. Garnaeva, A. Ivanov, D. Makrushin, and R. Unuchek. It threat evolution in q2 2015. *Kaspersky Lab*, 2015.

M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 27–36. IEEE, 2004.

S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

F-Secure Labs. BLACKENERGY and QUEDAGH: The convergence of crimeware and APT attacks, 2016.

N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, Oct. 2010.

Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

H. Flake. Structural comparison of executable objects. *DIMVA 2004, July 6-7, Dortmund, Germany*, 2004.

D. Formby, P. Srinivasan, A. Leonard, J. Rogers, and R. Beyah. Who's in control of your control system? device fingerprinting for cyber-physical systems. In *NDSS*, 2016.

L. Garcia and S. A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. 2017.

J. Glover, M. Sarma, and T. Overbye. *Power System Analysis and Design*. Cengage Learning, 2011. ISBN 9781111425777. URL `http://books.google.com/books?id=U77A2C37QesC`.

M. Gowda, J. Manweiler, A. Dhekne, R. R. Choudhury, and J. D. Weisz. Tracking drone orientation with multiple gps receivers. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 280–293. ACM, 2016.

Hex-Rays. Ida pro disassembler; available at `http://wiki.ros.org/ros_arduino_bridge`.

A. P. M. Hinkkanen. *Protecting an Industrial AC Drive Application against Cyber Sabotage*. PhD thesis, Aalto University, 2013.

H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks.

H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

X. Huo and M. Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1909–1915. AAAI Press, 2017.

X. Huo, M. Li, and Z.-H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, pages 1606–1612, 2016.

V. M. Igure, S. A. Laughter, and R. D. Williams. Security issues in scada networks. *Computers & Security*, 25(7):498–506, 2006.

L. C. Infrastructure. libfuzzer: a library for coverage-guided fuzz testing, 2017.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.

C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu. Revarm: A platform-agnostic arm binary rewriter for security applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 412–424. ACM, 2017.

C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2005.

P. Krumins. Gnu coreutils cheat sheet. *good coders code, great reuse*, 2012.

E. V. Kuz'min and V. A. Sokolov. On construction and verification of plc-programs. *Modelirovanie i Analiz Informatsionnykh Sistem [Modeling and Analysis of Information Systems]*, 19(4):25–36, 2012.

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

C. Kwon, W. Liu, and I. Hwang. Security analysis for cyber-physical systems against stealthy deception attacks. In *American Control Conference (ACC), 2013*, pages 3344–3349. IEEE, 2013.

D. Labs. Thirty percent of android devices susceptible to 24 critical vulnerabilities; available at `https://duo.com/decipher/thirty-percent-of-android-devices-susceptible-to-24-critical-vulnerabilitie` a.

S. R. Labs. The android ecosystem contains a hidden patch gap; available at `https://srlabs.de/bites/android_patch_gap/`. b.

A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2013.

J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.

T. G. Lewis. *Critical infrastructure protection in homeland security: defending a networked nation*. John Wiley & Sons, 2006.

Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. 2010.

Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. ISOC Network and Distributed System Security Symposium*, 2012.

Y. Liu, P. Ning, and M. K. Reiter. False data injection attacks against state estimation in electric power grids. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):13, 2011.

Z. Lu and Z. Zhang. Bad data identification based on measurement replace and standard residual detection. *Automation of Electric Power Systems*, 13:011, 2007.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

S. McLaughlin, S. Zonouz, D. Pohly, and P. McDaniel. A trusted safety verifier for controller code. In *NDSS*, 2014.

P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6), 2006.

Y. Mo, E. Garone, A. Casavola, and B. Sinopoli. False data injection attacks against state estimation in wireless sensor networks. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 5967–5972. IEEE, 2010.

T. H. Morris, A. K. Srivastava, B. Reaves, K. Pavurapu, S. Abdelwahed, R. Vaughn, W. McGrew, and Y. Dandass. Engineering future cyber-physical energy systems: Challenges, research needs, and roadmap. In *North American Power Symposium (NAPS)*, pages 1–6. IEEE, 2009.

P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *USENIX Annual Technical Conference, FREENIX Track*, pages 23–32, 2005.

M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. 2018.

E. Network and I. S. A. (ENISA). Protecting industrial control systems recommendations for Europe and Member States. `https://www.enisa.europa.eu/`, 2011.

B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 492–501. IEEE, 2013.

D. G. Peterson. Project Basecamp at S4. `http://www.digitalbond.com/2012/01/19/project-basecamp-at-s4/`, January 2012.

N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006.

J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415. ACM, 2014.

J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. In *ACM SIGPLAN Notices*, volume 42, pages 39–46. ACM, 2007.

J. Pollet. Electricity for Free? The Dirty Underbelly of SCADA and Smart Meters. In *Black Hat USA*, 2010.

A. O. S. Project. Android security bulletins; available at `https://source.android.com/security/bulletin`.

R. Qiao and R. Sekar. Effective function recovery for cots binaries using interface verification. Technical report, Technical report, Secure Systems Lab, Stony Brook University, 2016.

Quarkslab. Lief: library for instrumenting executable files; available at `https://lief.quarkslab.com/`, 2017-2018.

M. D. Raj, I. Gogul, M. Thangaraja, and V. S. Kumar. Static gesture recognition based precise positioning of 5-dof robotic arm using fpga. In *Trends in Industrial Measurement and Automation (TIMA), 2017*, pages 1–6. IEEE, 2017.

G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 1999.

F. Rashid. Ics-cert: Response to cyber incidents against critical infrastructure jumped 52 percent in 2012. *Security Week*, 10, 2013.

D. Rescue. Ida pro disassembler, 2006.

J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, and J. Dabell. A quantitative evaluation of the target selection of havex ics malware plugin. In *Industrial Control System Security (ICSS) Workshop*, 2015.

B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 255–269. USENIX Association, 2014a.

B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *USENIX Security Symposium*, pages 255–269, 2014b.

B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Guitar: Piecing together android app guis from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015a.

B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 146–157. ACM, 2015b.

H. Sandberg, S. Amin, and K. H. Johansson. Cyberphysical security in networked control systems: An introduction to the issue. *IEEE Control Systems*, 35(1):20–23, Feb 2015. ISSN 1066-033X. doi: 10.1109/MCS.2014.2364708.

A. L. Sangeetha, B. Naveenkumar, A. B. Ganesh, and N. Bharathi. Experimental validation of pid based cascade control system through scada–plc–opc and internet architectures. *Measurement*, 45(4):643–649, 2012.

A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *digital investigation*, 3:10–16, 2006.

E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, volume 16, 2013.

D. Shelar, P. Sun, S. Amin, and S. Zonouz. Compromising security of economic dispatch in power system operations. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 531–542. IEEE, 2017.

E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, pages 611–626, 2015.

Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

M. Simulink. Simulation and model-based design, 2005.

H. Singh and F. Alvarado. Network topology determination using least absolute value state estimation. *Power Systems, IEEE Transactions on*, 10(3):1159–1165, 1995.

A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

O. Sokolsky, S. Kannan, and I. Lee. Simulation-based graph similarity. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 426–440. Springer, 2006.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

W. Starbuck and M. Farjoun. *Organization at the limit: Lessons from the Columbia disaster*. John Wiley & Sons, 2009.

N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

S. Subramanian, M. Berzish, V. Ganesh, and O. Tripp. A solver for a theory of string and bit-vectors. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 124–126. IEEE Press, 2017.

P. Sun, R. Han, M. Zhang, and S. Zonouz. Trace-free memory data structure forensics via past inference and future speculations. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 570–582. ACM, 2016.

Y. Sun and T. J. Overbye. Visualizations for power system contingency analysis data. *IEEE Trans. on Power Systems*, 19(4):1859–66, 2004.

L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

R. Tan, V. Badrinath Krishna, D. K. Yau, and Z. Kalbarczyk. Impact of integrity attacks on real-time pricing in smart grids. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 439–450. ACM, 2013.

R. Tan, H. H. Nguyen, E. Y. Foo, X. Dong, D. K. Yau, Z. Kalbarczyk, R. K. Iyer, and H. B. Gooi. Optimal false data injection attack against automatic generation control in power grids. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–10, 2016.

TechNavio. Global industrial control systems (ics) security market 2014-2018. `https://www.marketresearchreports.com/technavio/global-industrial-control-systems-ics-security-market%C2%A02014-2018`, 2014.

D. Urbina, Y. Gu, J. Caballero, and Z. Lin. Sigpath: A memory graph based approach for program data introspection and modification. In *Computer Security-ESORICS 2014*, pages 237–256. Springer, 2014.

U.S. Department of Energy Office of Electricity Delivery and Energy Reliability. North american electric reliability corporation critical infrastructure protection (nerc-cip), 2015.

S. E. Valentine. *PLC code vulnerabilities through SCADA systems.* PhD thesis, University of South Carolina, 2013.

M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak. Software verification and graph similarity for automated evaluation of students assignments. *Information and Software Technology*, 55(6):1004–1016, 2013.

R. J. Walls, E. G. Learned-Miller, and B. N. Levine. Forensic triage for mobile phones with dec0de. In *USENIX Security Symposium*, 2011.

A. Walters. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

Y. Wang, Z. Xu, J. Zhang, L. Xu, H. Wang, and G. Gu. Srid: State relation based intrusion detection for false data injection attacks in scada. In *European Symposium on Research in Computer Security*, pages 401–418. Springer, 2014.

A. J. Wood and B. F. Wollenberg. *Power generation, operation, and control.* John Wiley & Sons, 2012.

X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

O. Yuschuk. Ollydbg, 2007.

J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334. ACM, 2010.

B. Zeng and Y. An. Solving bilevel mixed integer program by reformulations and decomposition. 2014.

J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 487–498. ACM, 2013.

H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 887–902, 2018.