June 2017

THE SCALABLE AND ACCOUNTABLE BINARY CODE SEARCH AND ITS APPLICATIONS

Qian Feng
Syracuse University

ABSTRACT

The past decade has been witnessing an explosion of various applications and devices. This big-data era challenges the existing security technologies: new analysis techniques should be scalable to handle "big data" scale codebase; They should be become smart and proactive by using the data to understand what the vulnerable points are and where they locate; effective protection will be provided for dissemination and analysis of the data involving sensitive information on an unprecedented scale.

In this dissertation, I argue that the code search techniques can boost existing security analysis techniques (vulnerability identification and memory analysis) in terms of scalability and accuracy. In order to demonstrate its benefits, I address two issues of code search by using the code analysis: scalability and accountability. I further demonstrate the benefit of code search by applying it for the scalable vulnerability identification [57] and the cross-version memory analysis problems [55, 56].

Firstly, I address the scalability problem of code search by learning "higher-level" semantic features from code [57]. Instead of conducting fine-grained testing on a single device or program, it becomes much more crucial to achieve the quick vulnerability scanning in devices or programs at a "big data" scale. However, discovering vulnerabilities in "big code" is like finding a needle in the haystack, even when dealing with known vulnerabilities. This new challenge demands a scalable code search approach. To this end, I leverage successful techniques from the image search in computer vision community and propose a novel code encoding method for scalable vulnerability search in binary code. The evaluation results show that this approach can achieve comparable or even better accuracy and efficiency than the baseline techniques.

Secondly, I tackle the accountability issues left in the vulnerability searching problem by designing vulnerability-oriented raw features [58]. The similar code does not always represent the similar vulnerability, so it requires that the feature engineering for the code

search should focus on semantic level features rather than syntactic ones. I propose to extract conditional formulas as higher-level semantic features from the raw binary code to conduct the code search. A conditional formula explicitly captures two cardinal factors of a vulnerability: 1) erroneous data dependencies and 2) missing or invalid condition checks. As a result, the binary code search on conditional formulas produces significantly higher accuracy and provides meaningful evidence for human analysts to further examine the search results. The evaluation results show that this approach can further improve the search accuracy of existing bug search techniques with very reasonable performance overhead.

Finally, I demonstrate the potential of the code search technique in the memory analysis field, and apply it to address their across-version issue in the memory forensic problem [55, 56]. The memory analysis techniques for COTS software usually rely on the so-called "data structure profiles" for their binaries. Construction of such profiles requires the expert knowledge about the internal working of a specified software version. However, it is still a cumbersome manual effort most of time. I propose to leverage the code search technique to enable a notion named "cross-version memory analysis", which can update a profile for new versions of a software by transferring the knowledge from the model that has already been trained on its old version. The evaluation results show that the code-search based approach advances the existing memory analysis methods by reducing the manual efforts while maintaining the reasonable accuracy. With the help of collaborators, I further developed two plugins to the Volatility memory forensic framework [2], and show that each of the two plugins can construct a localized profile to perform specified memory forensic tasks on the same memory dump, without the need of manual effort in creating the corresponding profile.

THE SCALABLE AND ACCOUNTABLE BINARY CODE SEARCH AND ITS APPLICATIONS

by

Qian Feng

B.S., Xian Jiaotong University, 2008

M.S., Xi'an Jiaotong University, 2011

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Syracuse University

May 2017

To my parents, and my boyfriend.

ACKNOWLEDGMENTS

The work presented in this thesis could not have been created without the encouragement and guidance provided by many others. I would like to acknowledge those who have helped me throughout this effort.

Foremost, I would like to express my sincere gratitude to my advisor Prof. Heng Yin for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof.Wenliang Du, Prof.Joon S. Park, Prof.Yuzhe Tang, Prof.Senem Velipasalar, and Prof.Yanzhi Wang for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also go to Dr. Lenx Tao Wei, who provided me an opportunity to join their team as intern. His precious support provides me valuable insights and industrial experience.

I thank my fellow labmates for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years. The help and friendship of my labmates and fellow graduate students (both current students and those long since graduated), has made the long doctoral process much more enjoyable. In particular, the guidance of Lok, Aravind, Mu and Andrew, and the company

of Xunchao, Minghua, Rundong, Jinghan, Yue Duan, Chengcheng, Charles and Brain have helped see me through qualifier exams, submission deadlines, and years of cold and snowy Syracuse weather.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis and my life in general. I am proud to acknowledge my parents Zhicheng Feng, Liya Pang and my boyfriend Lu Jiang for all of the support and encouragement that they have given me during my doctoral studies. Without their help to make the doctoral process more tenable, I doubt I would have made it through. For this, I will forever be thankful. I look forward to all of the long walks and trips to the park that we will have in our future.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Software is ubiquitous. It exists in any device which facilitates our daily life such as a car, a router, TV, printer, even a tiny thermo. Different programs hold their own applications and share various common libraries. Considering any piece of code as a building block, software developments usually build their skyscrapers based on existing code blocks. This phenomenon is referred as code reuse, and it encompasses an opportunity to both learn from the reused code, and leverage that knowledge for the benefit of its commercial clients.

However, the code reuse phenomenon is a double-edged sword. On one hand, it increases the potential security risks. Any vulnerable code block could put the whole project in danger, and it is not uncommon to reuse the vulnerable code block during the software development. For example, Heartbleed bug, a notorious vulnerability in OpenSSL, affected a huge number of popular websites, including Google, YouTube, Yahoo!, Pinterest, Blogspot, etc, since most of website servers reuse functionalities of vulnerable versions of OpenSSL for encryption. This is neither the first nor the last crisis caused by the vulnerability reuse. On the other hand, the code reuse phenomenon can facilitate the security analysis process. Security analysts can reuse their analysis results for the similar code across software of different versions or architectures. They can also investigate the code reuse properties among different versions of software for the lineage study [80].

Binary code search is one of the prevalent research directions towards the code reuse analysis [78, 133, 139]. Given a binary function, it tends to find equivalent or similar binary

functions from a large corpus of binary executables. It has many security applications, such as plagiarism detection, malware detection, vulnerability detection.

1.1 Code Search in Vulnerability Identification

A vulnerability is a system flaw or weakness in an application that could be exploited to compromise the security of the application. The explosion of various applications and devices challenges the existing analysis techniques due to limited resources of time, hardware, money, and human ability. The existing techniques find potential vulnerabilities of an application by running various test cases via white-box testing [63, 67] or fuzzing techniques [33, 118]. These vulnerability identification techniques handle applications case by case. For each application, the vulnerability identification depends on a good seed selection policies, enough computation resources, and a configured running environment. The preparation for these requirements takes time for even a single application, not to mention a million of applications or devices. Even worse, it is still a research problem to set up simulation environment for some applications such as IoT devices [35, 141].

In fact, the vulnerability could be duplicated during the software development due to the code reuse phenomenon. Studies found that 49 bugs in Linux were caused by the fact that developers did not fix buggy code that was copied from one project to another [92]. By searching a known vulnerable code snippet in the codebase, the security analysis can quickly localize the potential candidate with the same vulnerability in the codebase and conduct further inspection. The whole process is lightweight without any source code or running environment requirement.

1.2 Code Search in Memory Analysis

The memory analysis on COTS binaries is known for its tedious manual efforts [47, 61, 65]. Existing memory analysis tools still rely on cumbersome reverse engineering techniques to build the data structure profile. In most cases, the profile generation still depends on the manual effort. Unfortunately, the daunting profile creation task is not a one-time effort. It is tightly coupled to a specific version of the software being analyzed, and needs to be constantly rebuilt for new versions of the software.

As a result, the effort spent on building the analysis profile for one particular version of a program will not be applicable to its future versions. In the big-data era, the security experts might handle hundreds of memory dumps at one time. Therefore, the manual profile generation process could not keep up with handling memory dumps on an unprecedented scale. For example, in an Infrastructure as a Service (IaaS) cloud, the cloud provider usually depends on memory analysis tools to monitor the security condition of guest OSes [69].

The code search can help the traditional memory analysis to automatically locate the code of similar functionality in the new version of the software, and transfer the knowledge from the profile, which is related to the same functionality in the original software.

1.3 Challenges in Code Search Techniques

Many studies utilize different binary code comparison techniques for code similarity analysis in the commercial software, such as graph isomorphism technique [59], tracelet based approach [41], N-gram or N-perm technique [86] etc. Some of them begin to explore the bug identification in IoT devices in the cross-architecture setting. Unfortunately, there

are two drawbacks in existing binary code comparison techniques which prevent them from being applied for code search.

Scalability. Existing binary code search techniques lag in scalability due to expensive feature extraction and comparison algorithms or search strategy techniques. Some binary code search techniques adopt complex in-depth binary analysis for feature extraction, which itself takes time for a single binary, not to mention conducting the search in tens of thousands of binaries [64, 99, 113]. For some techniques of efficient feature extraction processes, their comparison algorithms are too expensive, and therefore fail to be scalable. A few works like DiscovRe [53] target on the efficient code search by pre-filtering unlikely noises, but their approach is not reliable, and can still not be applied for bug search in the large scale codebase.

Accountability. For most existing binary code search techniques, accountability also remains a challenge. When a match is reported, it is hard to understand underlying reasons [41]. The match could be a result of sharing similar control flow graphs, or opcode sequences instead of similar functionalities. Without providing enough execution semantics, it is still hard for analysts to verify a vulnerability only based on matched CFGs or opcode sequences. They could conduct further tedious reverse engineering work on the binary of matched code for functionality comparison, and make a fair conclusion based on their manual analysis results.

1.4 Thesis Statement

Built on the demand of the binary code search, the thesis of this work is that scalable and accountable code search techniques can boost existing security analysis techniques (vulnerability identification and memory analysis) in terms of efficiency and accuracy.

More specifically, I propose a scalable and accountable binary code analysis platform depicted in Figure 1.1. I firstly propose new techniques to address scalability and accountability problems in the binary code search respectively. I then discuss how to use the code search technique in two security scenarios: memory analysis and software vulnerability tracking at scale.

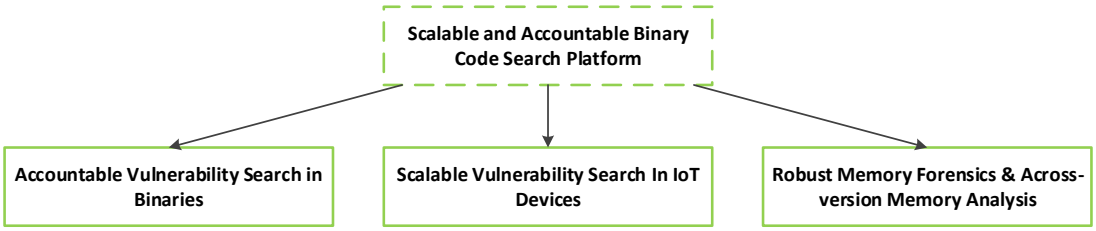


Fig. 1.1.: Overview of Thesis Work.

1. **Scalable graph-based code search approach.** To address the scalability issue in the binary code search, instead of comparing binary code with "raw features", I adopt the machine learning approach to learn "high-level" numeric features from raw features without the loss of essential semantic information. The resulting numeric feature representation can be conveniently indexed via mature hashing techniques, and therefore produce real-time search speed. This work has resulted in a publication in CCS 2016 [57].

2. **Accountable code search approach.** I introduce the novel semantic-level feature representation, "conditional formula" to address the accountability issue of the binary code search. A conditional formula is distilled directly from binary code, and explicitly captures two cardinal factors of a code logic: 1) data dependencies and 2) condition checks. As a result, the binary code alignment on conditional formulas produces a one-to-one optimal behavior mapping which provide meaningful logs for human analysts to further examine the search results. This work has resulted in a publication in AsiaCCS 2017 [58].

3. **Cross-version memory analysis.** To better understand the binary code search problem, I, with help of my collaborators, systematically investigate the code reuse phenomenon in common software libraries, and design a code-search based approach to automatically update the data profile to enable the cross-version memory analysis. This work has resulted in publications in ACSAC 2014 [55] and AsiaCCS 2016 [56].

In summary, my contributions in this thesis work are as following:

- **Novel Techniques:** I propose a scalable and accountable binary code search platform, which can address two of fundamental limitations (efficiency and accuracy) in existing binary code search techniques.

- **Real Applications:** I apply the novel binary code search platform into two real-world applications: bug search in IoT devices and memory analysis.

- **Solid Evaluation:** To demonstrate the effectiveness of the novel binary code search platform, I collect 3,3045 firmware images and prepare 3 baseline approaches to

evaluate the scalability of binary code search for vulnerability. Besides, I also collect 4 types of software including 40 versions of OpenSSH over a span of 13 years for the effectiveness evaluation of code search in the application of memory analysis.

2. BACKGROUND

This chapter presents a survey of existing code search techniques and its applications to better understand the capabilities and limitations of these prior works. It also provides an understanding of how the state of the art in the code search technique is advanced by the design of our proposed approaches.

2.1 Feature Engineering in Code Search

Feature engineering is the process of using domain knowledge of the code to create features that make code search algorithms work. Feature engineering includes the source-code based search and binary-only approach depending on whether the source code is available.

Source-code based Approach. Feature engineering usually extracts features from source code for finding code clones at the source code level. For example, [139] generates a code property graph from the source code as the feature and conducts a graph query to search for code clones with the same pattern. Similarly, token based approaches such as CCFinder [84] and CP-Miner [92] utilize token sequence as features and scan for duplicate token sequences in other source code. DECKARD [36] generates numerical vectors based upon abstract syntax trees and conducts code similarity matching for code clone detection. ReDeBug [79] normalizes the program file and extracts the tokenized sequences

as the feature to find unpatched code clones in OS-distribution bases. However, for most of commercial softwares without source code, these approaches cannot be applied.

Binary-only Approach. Binary-only feature engineering approaches are proposed to address the problem of code search without source code. There are three types of feature engineering on the binary code: 1) `syntax-based analysis`, 2) `static analysis`, and 3) `dynamic analysis`.

1) Syntax-based analysis. The feature in syntax-based approach could be a simple binary sequence or mnemonic code without understanding the semantics of code [78, 85]. This is an efficient way to extract features from binaries, since it does not need further complex disassembling process. N-grams and N-perms are two representatives of syntax-based approach. N-grams is a contiguous sequence of n items obtained by a sliding window of n bytes over the extracted byte sequence of binary code [87, 133]. As a variant of N-gram, N-perms represents every possible permutation of n items in an n-gram. Since the order of n items in N-perms does not affect the matching process, N-perms analysis is expected to be more tolerant of reordering and yield higher similarity scores [85, 86].

2) Static Analysis: Syntax-based analysis is not accurate, since it cannot capture semantics of binary code [41]. Static analysis addresses this issue via analyzing binary code and models its semantics as features. Control-flow graph is a prevalent feature. Each node on the graph is a basic block and edges are the control dependency. As a result, the control flow graph can effectively represent the code logic of binary code. Flake et al. [59] proposed to match CFGs of a function to defeat some compiler optimizations such as instruction reordering and changes in register allocation. However, the approach relies upon exact

graph matching which is too expensive to be applied for large scale bug search. Pewny et al. [113] use Minhashing to capture semantics at the basicblock level and construct an attributed graph as features for bug search. Compared with Pewny et al. [113], DiscovRe [53] also utilizes an attribute graph, but each node in the graph is a list of the statistical features to represent the semantics of each basic block. Zynamics BinDiff [51] and BinSlayer [24] also use the control flow graphs for the whole binary file comparison.

Tracelet-based approach [41] captures execution sequences as features for code similarity checking, which can defeat the CFG changes. TEDEM [112] captures semantics by the expression tree of a basic block. BinHunt [64] and iBinHunt [99] utilize symbolic execution and a theorem proving to check semantic equivalence between basic blocks. These two approaches are expensive and cannot be applied for large scale bug search since they need to conduct binary analysis to extract the equations and conduct the equivalence checking.

A call graph as another feature is also well used in binary code analysis or clustering. It represents the control flow between functions in a directed graph where a node represents a function and an edge indicates the calling relationship. For example, Hu et al. [73] presented a function-call graph-based malware detection system where each malware is represented by its function-call graph. Call graph matching is expected to be less susceptible to deception by polymorphism than syntax-based matching.

3) Dynamic Analysis. Blanket execution [52] uses the runtime environment of the program as features to conduct the code search. This approach can defeat changes in CFGs, but it is only evaluated in a single architecture. Besides, the support of dynamic analysis is not general enough to support all platforms and devices. For example, the support to

firmware images is at the initial stage [35, 141], and still has not been demonstrated its effectiveness with respect to the run-time environments for programs in large scale analysis.

2.2 Similarity Metrics in Code Search

The similarity metric is used to quantify the similarity of two pieces of binary code based on their features. It is important for the code search. A good similarity metric can assist us to localize the target code in the code base. Otherwise, the false positives can overwhelm the true positive and make the code search futile.

The similarity metric depends on the feature representation. Basically, there are two types of metrics for similarity matching. The first one is the sequence based code matching, and the second one is the graph based matching. ReDebug [79] utilizes Jaccard similarity to quantify the similarity score of two tokenized sequences. The drawback of Jaccard similarity is that it ignores the order of a sequence, so it will produce false positives. Tracelet-based approach [41] utilizes the edit distance as the metric to quantify the similarity of sequences to mitigate the drawback of Jaccard similarity. Although the string edit distance can increase the accuracy, the sequence-based code search still cannot get good results due to the limitation of the sequence-based approach.

Graph based similarity metric is well adopted in the graph based code search approaches. It is usually used to quantify the similarity of two graphs. Hu et al. [73] utilize the bipartite graph matching to quantify the similarity of two function-call graphs. Pewny et al. [113] also utilize the bipartite graph matching but they consider the distance of nodes on two compared graphs to increase the matching accuracy. However, the bipartite graph

matching ignores the structural info during matching, so DiscovRe [53] utilizes maximum common subgraph matching as the similarity metric to further increase the similarity accuracy. However, the graph matching is very expensive, so it cannot be used for the scalable code search. How to scalable search in the "big code" is still an open question.

2.3 Applications in Code Search

2.3.1 Memory Analysis

Memory forensics is one application for the memory analysis. Several memory analysis tools [2, 9, 62, 98, 111, 121] etc. have been proposed to aid the automatic memory forensics. They aim at analyzing and retrieving sensitive information from a memory dump. A key aspect of memory forensics is to encode the semantics related information into the data structure profile and follow the profile to conduct the specific analysis. The profile is designed to the specific version of the image being analyzed, and needs to be updated version by version. State-of-the-art techniques rely on reverse engineering to reconstruct the profile of semantic information, which is the manual effort or use nontrivial scripts [2] that operate on the source code. The code search can facilitate the memory analysis to be scalable by transferring the analysis results across versions or similar code.

VMI (Virtual Machine Introspection) is another application of memory analysis. It extracts semantic knowledge from a running virtual machine to monitor and inspect semantic behaviors of the guest machine.Due to the nature of isolation, VMI has been applied for many security applications. For example, many intrusion detection applications utilize the VMI technique to conduct more accurate detections [65, 108, 109]. Some malware analysis

approaches also relies on the VMI to capture the detail malware behaviors which cannot be captured by previous work [43, 83]. Furthermore, VMI techniques are also well used in memory forensics and process monitoring [71]. The main challenge in the VMI technique is to bridge the semantic gap between the guest OS and external analysis tools. Many existing works have already made successful attempts to solve this problem [48, 61]. A recent tool, DECAF [72] performs VMI to retrieve key semantic information from a guest OS. In each of the above efforts, similar to memory forensics, a non-trivial effort is required to construct a profile) of key semantic values and their concrete interpretations within the guest OS. Although VMST [61] can reuse the OS code pieces of the introspection property to achieve the automatic VMI. However, the approach used in VMST could not be general enough to support introspection for some internal and close-sourced data structures.

2.3.2 Vulnerability Identification

Fuzzing and concolic execution are two ways of automatic vulnerability identification. Fuzzing is a popular and effective choice for finding bugs in applications. No matter the black-box fuzzing or white-box fuzzing, it generates as many as possible input seeds to trigger the testing software to find buggy code [33, 63, 67, 118]. Fuzzing suffers from lack of guidance new inputs are generated based on random mutations of prior inputs, with no control over which paths in the application should be targeted at [33]. Therefore, how to find an effective seed selection strategy is still an open question due to the complex internal of a software [33, 118]. It takes money, resources and time to fuzz hundreds of thousands of programs in a scalable and efficient way. A light-weight and scalable code search technique

could be used as a potential to address the scalability of fuzzing techniques. Considering fuzzing as the foot of a human being, the code search technique is the "brain" and can provide the guide for fuzzing to selectively explore the path which could be vulnerable first, which can greatly reduce the search space of the seed selection for fuzzing and save time and resources.

Concolic execution (also known as dynamic symbolic execution) generates inputs based on the program analysis instead of blindly exploring the whole seed space [132]. It interprets an application, models a user input using symbolic variables, tracks constraints introduced by conditional jumps, and use a constraint solver to create inputs to drive applications down specific paths. Although concolic execution based approaches are effective, it still suffers from the path explosion problem. Many techniques are proposed to mitigate the path explosion problem [18, 116, 128], but some explosions still eventually occur or many produced inputs are not directly actionable.

In addition to the limitations of fuzzing and concolic execution discussed above, the key limitation for the existing vulnerability identification techniques is that they require running the test program first to test its vulnerability. In most cases, directly running devices or cyber-physical systems for testing are not realistic and expensive [35, 141]. Simulating these running environment is still an open problem. Therefore, directly conducting fuzzing or concolic execution on these systems at scale is infeasible.

On the contrary, the code search technique mitigates the limitations of the existing vulnerability identification techniques. Firstly, it provides the guide for these identification techniques. Instead of blindly generating useless seeds, the code search technique can provide the guide for fuzzing to selectively explore the path which could be vulnerable

first. This can greatly reduce the search space of the seed selection for fuzzing and save time and resources. The concolic execution can also be limited to the vulnerable candidates provided by the code search to avoid exploring irrelevant path exploration.

The most important is that the code search is light-weight and does not need the running environment [53, 113]. Given a list of known vulnerable code pieces, the code search technique can quickly pinpoint the potential vulnerable code in the target system or devices. Even if these devices cannot be directly analyzed by concolic execution or fuzzing, the code search still provides an effective way to find potential vulnerabilities.

3. SCALABLE AND ACCOUNTABLE CODE SEARCH PLATFORM

In this thesis, I propose a scalable and accountable code search platform. It comprises of three components: the static binary analysis platform described in Section 3.1, scalable code search engine described in Section 3.2 and accountable code search engine described in Section 3.3. The static binary analysis platform is responsible to extract features for the code search. The scalable code search engine enables the code search in real time, and the accountable code search engine will further refine the search result from the scalable search engine and provide execution semantics matched results for the analyst to quickly screen.

3.1 Static Binary Analysis Platform

I have implemented a static binary analysis platform, on top of IDA Pro [75] and LLVM [1] in 23k lines of code. This tool will extract features for search on demand. In our scenario, it supports the extraction on two types of feature: the attributed control flow graph and conditional formula.

3.1.1 Attributed Control Flow Graph

The CFG (Control Flow Graph) is the common feature used in bug search. More recently, different attributes on the basic blocks, such as I/O pairs and statistic features [53,

113], are explored to further increase the matching accuracy. Following the idea, this chapter utilizes the control flow graph with different basic-block level attributes called ACFG (Attributed Control Flow Graph) as the raw feature to model the function in our problem.

Definition 3.1.1 *(Attributed Control Flow Graph) The attributed control flow graph, or ACFG in short, is a directed graph* $G = \langle V, E, \phi \rangle$*, where* V *is a set of basic blocks;* $E \subseteq V \times V$ *is a set of edges representing the connections between these basic blocks, and* $\phi : V \to \Sigma$ *is the labeling function which maps a basic block in* V *to the attributes in* Σ*.*

The attribute set Σ in Definition 3.1.1 can be tailored depending upon the level of detail required to accurately characterize a basic block. For efficiency, instead of using expensive semantic features like I/O pairs [113], I focus on two types of features in this chapter: statistical and structural. The statistical features describe local statistics within a basic block, whereas the structural features capture the position characteristics of a basic block within a CFG. Inspired by [53], in Σ I extract six types of statistical features and two types of structural features, listed in Table 3.1. Existing works have demonstrated the advantages of the statistical features in Σ for the cross-architecture bug search [53].

Table 3.1: Basic-block level features.

Type	**Feature Name**	**Weight** (α)
Statistical Features	String Constants	10.82
	Numeric Constants	14.47
	No. of Transfer Instructions	6.54
	No. of Calls	66.22
	No. of Instructions	41.37
	No. of Arithmetic Instructions	55.65
Structural Features	No. of offspring	198.67
	Betweeness	30.66

Inspired by the work on complex network analysis, I propose two types of structural features: *no. of offspring* and *betweenness centrality*. The *no. of offspring* is the number of children nodes in the control flow graph. This information helps locate the layer of a node in the graph. The *betweenness centrality* measures a node's centrality in a graph [105]. Nodes in the same layer in the CFG could have different betweenness centrality. In summary, the proposed features consider not only the statistical similarity but also the structural similarity between two ACFGs.

Implementation To generate the attributed graph for a binary function, I first extract its control flow graph, along with attributes in Σ for each basic block in the graph, and store them as the features associated with the basic block.

3.1.2 Conditional Formula

I also introduce a novel semantic feature called "conditional formula" for the accountable code search. It is a middle ground between binary-level syntactic features and source-code level representation. It factorizes tangled code logic into conditional formulas as logic-independent units. Generally, a conditional formula consists of an If-clause and a Then-clause, and each clause is a symbolic formula, describing that under what condition (stated in the If-clause) a given action (in the Then-clause) will take place. A conditional formula explicitly captures two cardinal factors of a piece of binary code: (1) data dependencies, and (2) condition checks. Instead of treating the binary function as a whole, searching on structured conditional formulas can effectively represent code logic in more

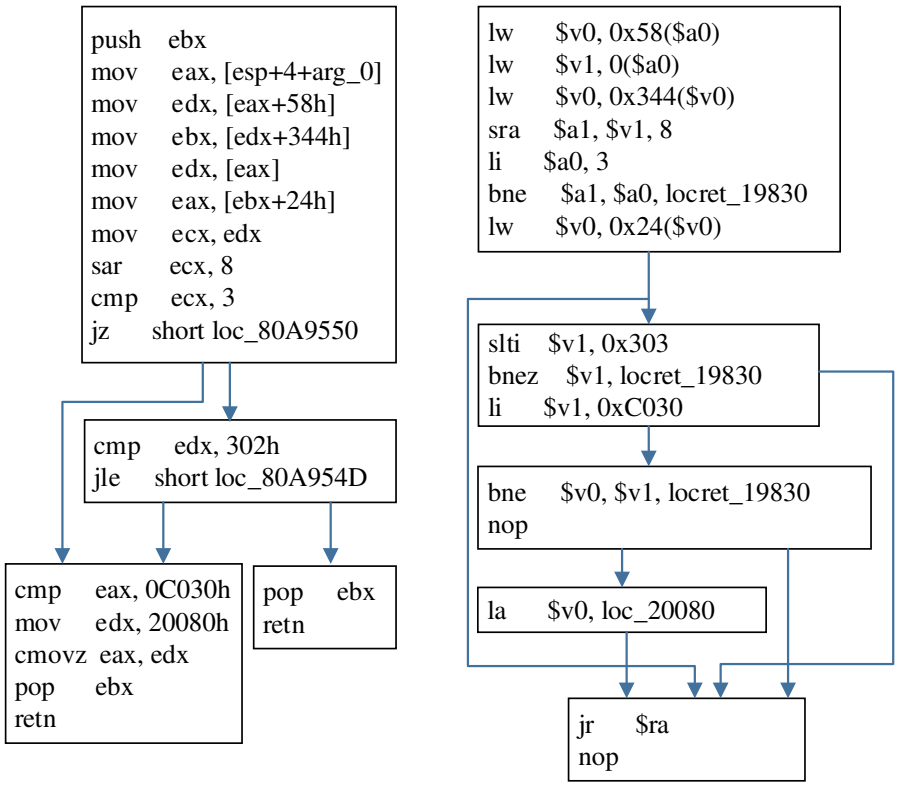
descriptive way. By contrasting conditional formulas between two matched candidates, an analyst can quickly diagnose whether the target is semantically similar or a false positive.

Definition 2. A *conditional formula* (`CF`) consists of an *action* (which describes how a function output is computed from one or more function inputs), and an optional *condition* (which is a boolean expression that triggers the execution of the formula).

⟨*CondFormula*⟩ ::= ⟨*Action*⟩
| ⟨*Condition*⟩ '→' ⟨*Action*⟩
⟨*Condition*⟩ ::= ⟨*Expression*⟩
⟨*Action*⟩ ::= ⟨*Assignment*⟩
| ⟨*Function*⟩
⟨*Function*⟩ ::= ⟨*FnName*⟩ '(' ⟨*ParamList*⟩ ')'
⟨*ParamList*⟩ ::= ⟨*Expression*⟩
| ⟨*Expression*⟩ ',' ⟨*ParamList*⟩
⟨*Assignment*⟩ ::= ⟨*Expression*⟩ '=' ⟨*Expression*⟩
⟨*Expression*⟩ ::= ⟨*Name*⟩
| ⟨*Number*⟩
| ⟨*Function*⟩
| ⟨*Expression*⟩ ⟨*BinOp*⟩ ⟨*Expression*⟩
| '(' ⟨*Expression*⟩ ')'
| '[' ⟨*Expression*⟩ ']'

Fig. 3.1.: Abbreviated BNF for Conditional Formula

An abbreviated version of Backus-Naur Form for conditional formula is shown in Figure 3.1. A conditional formula `<CondFormula>` consists of a condition expression `<Condition>` together with an action statement `<Action>`, with a connector → in between. If the triggering condition is always true, `<CondFormula>` is simply `<Action>`.



a) x86 assembly **b) MIPS assembly**

Fig. 3.2.: The control flow graph comparison for the vulnerable function `ssl_get_algorithm2` (*CVE-2013-6449*) under different architectures(x86 vs. MIPS).

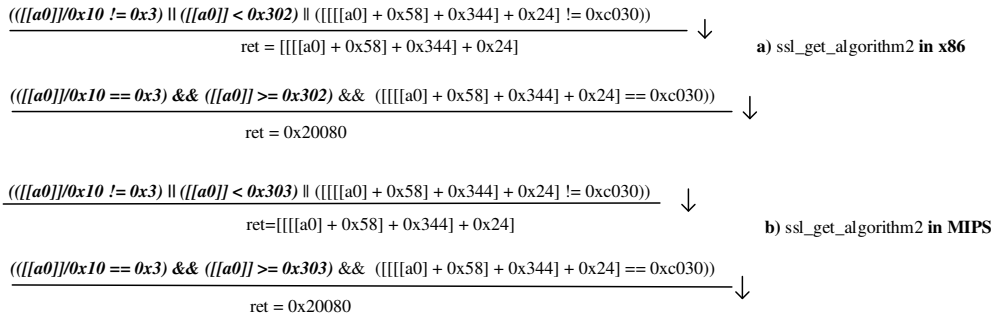


Fig. 3.3.: Conditional Formulas for the Motivating Example

While a condition `<Condition>` is a standard expression `<Expre- ssion>`, an action `<Action>` can be either an assignment `<Assign- ment>` or a function call `<Function>`. An expression can be as simple as an integer number, a variable name, a function call, a

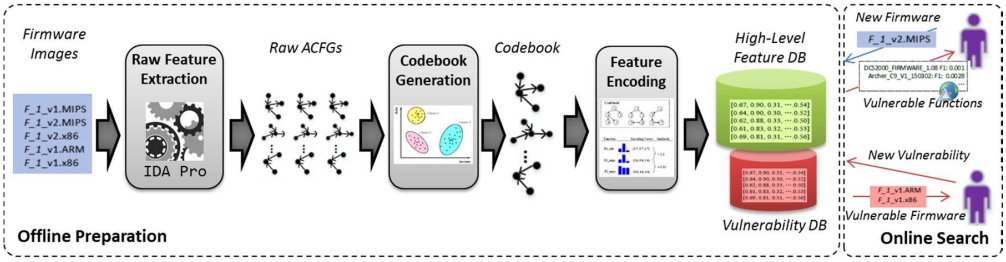


Fig. 3.4.: **The approach overview**

combination of two subexpressions connected by a binary operation (such as '+', '-', '&&', etc.), or surrounded by a pair of parentheses '()' or square brackets '[]'. Note that a pair of square brackets '[]' denote a memory dereference.

Figure 3.3 shows the conditional formulas for the two binary functions in Figure 3.2. Given the vulnerable `ssl_get_algorithm2` in x86, I can precisely label the vulnerable logic, the invalid condition check on its conditional formulas. Our approach searches the whole `OpenSSL` binary in DD-WRT and finds a candidate, `ssl_get _algorithm2` in MIPS. It shares the similar code logic. Our approach also produces the best match between the conditional formulas in both functions. As we see, the code logic is interpreted by the conditional formulas, and it becomes evident for an analyst to diagnose the vulnerability in DD-WRT firmware based on the matched conditional formulas.

3.2 Scalable Code Search Engine

Inspired by image retrieval techniques, the scalable code search engine includes the following main steps, as shown in Figure A.1: 1) *raw feature extraction*, 2) *codebook*

generation, 3) *feature encoding*, and 4) *online search*. The first step aims at extracting the attributed control flow graph, which is referred to as the raw feature, from a binary function (Section 3.1.1). *Codebook generation* utilizes unsupervised learning methods to learn higher-level categorizations from raw attributed control flow graphs (Section 3.2.1). *Feature encoding* encodes the attributed control flow graph by learned categorizations into higher-level feature vector residing in the high-dimensional space (Section 3.2.1). Finally, given a function, *online search* aims at efficiently finding its most similar functions by Locality Sensitive Hashing (LSH) [15]. Since each function is transformed into a higher-level numeric feature in the *feature encoding* step, I can directly apply LSH to conduct efficient searches in terms of the approximated cosine and Euclidean distance between the higher-level features (Section 3.2.1). The details of each step will be discussed in the following sections.

Generally, there are two stages in the proposed method: offline indexing and online search. Offline indexing, which includes raw feature extraction, codebook generation and feature encoding, is applied to existing functions before I can perform searches. Similar to text and image search methods, this step is a one-time effort and can be trivially paralleled across multiple CPU cores. The online search phase, which includes feature encoding and search, is applied against a few unseen functions. Due to the limited number of online operations, online search is typically sufficiently fast for large-scale search engines. This section outlines the basic steps for the scalable bug search approach. It involves two steps: the high-level feature generation and search engine construction.

3.2.1 High-level Feature Generation

As I discussed before, I select the attributed control flow graph as the raw feature for efficiency. The high-level feature generation works on raw features and outputs a numeric feature vector, each dimension of which represents the similarity to one categorization learned from raw features. More specifically, it involves two steps: the codebook generation and high-level feature encoding.

The codebook generation aims at learning a set of categorizations, that is, codewords, from raw features. Formally, a codebook $\mathcal{C}$ is a finite and discrete set: $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$, where c_i is the i-th codeword, or "centroid", and i is the integer index associated with that centroid. The codebook is generated from a training set of raw features by an unsupervised learning algorithm. In our case, the raw features are the control flow graphs.

A. Raw Feature Similarity I consider the raw feature similarity computation as a labeled graph matching problem. By definition, ACFGs are matched not only by their structures but also by their labels (attributes) on the structures. Theoretically, graph matching is an NP-complete problem, but many techniques have been proposed to optimize the process for an approximate matching result [27, 119]. For efficiency, I utilize bipartite graph matching to quantify ACFG similarity. Although other approaches such as MCS (Maximum Common Subgraph) matching [27] may also be applied to this problem, efficiency is still a major concern. The primary limitation of bipartite graph matching is that it is agnostic to the graph structure, and the accumulation of errors could result in less accurate results. To address the issue, I have the attributed control flow graph in Section 3.1.1, to allow bipartite graph matching to incorporate some graph structural information.

Essentially, bipartite graph matching utilizes the match cost of two graphs to compute the similarity. It quantifies the match cost of two graphs by modeling it as an optimization process. Given two ACFGs, G_1 and G_2, the bipartite graph matching will combine two ACFGs as a bipartite graph $G_{bp} = (\hat{V}, \hat{E})$, where $\hat{V} = V(G_1) \cup V(G_2)$, $\hat{E} = \{\hat{e_k} = (v_i, v_j) | v_i \in V(G_1) \wedge v_j \in V(G_2)\}$, and edge $\hat{e_k} = (v_i, v_j)$ indicates a match from v_1 to v_2. Each match is associated with a cost. The minimum cost of two graphs is the sum of all edges cost on the mapping. Bipartite graph matching can go over all mappings efficiently, and select the one-to-one mapping on nodes from G_1 to G_2 of the minimum cost.

In our problem, a node in the bipartite graph is a basic block on the ACFG. The edge cost is calculated by the distance between the two basic blocks on that edge. Each basic block on the ACFG has a feature vector discussed in Section **??**. Therefore, I calculate the distance between two basic blocks by $\text{cost}(v, \hat{v}) = \frac{\sum_i \alpha_i |a_i - \hat{a}_i|}{\sum_i \alpha_i \max(a_i, \hat{a}_i)}$. It is the same distance metric used in the paper [53] to quantify the distance of two basic blocks. a_i and $\hat{a}_i$ are the i-th feature in feature vectors of two basic block v and $\hat{v}$ respectively. If the feature is a constant, $|a_i - \hat{a}_i|$ is their difference. If the feature is a set, I use Jaccard to calculate the set difference. α_i is the corresponding weight of the feature which will be discussed below.

The output of bipartite graph matching is the minimum cost of two graphs. Normally the match cost of two graphs is greater than one, and positively correlated to the size of compared ACFGs. Therefore, I normalize the cost to compute the similarity score. For cost normalization, I create an empty ACFG Φ for each compared ACFG. Each node in the empty graph has an empty feature vector, and the size of the empty graph is set to that of the corresponding compared graph. By comparing with this empty ACFG, I can obtain the maximum matching cost the compared graph can produce. I compute the matching cost

with the empty graph for the two graphs, and select the maximum matching cost as the denominator, and use it to normalize the matching cost of two graphs. Suppose $cost(g_i, g_j)$ represents the cost of the best bipartite matching between two graphs g_1, g_2, the ACFG similarity between two graphs can be formally represented as following:

$$\kappa(g_1, g_2) = 1 - \frac{\text{cost}(g_1, g_2)}{\max(\text{cost}(g_1, \Phi), \text{cost}(\Phi, g_2))}, \tag{3.1}$$

I found that the features in Table 3.1 have different importance in computing graph similarity. I learn weights of the raw features to capture the latent similarity between two ACFGs. Basically, the learning objective is to find weight parameters that can maximize the distance of different ACFGs while simultaneously minimizing the distance of equivalent ACFGs. To approach this optimization problem, I adopt the approach used by Eschweile et al [53]. More specifically, I use a genetic algorithm using GALib [136]. I also execute an arithmetic crossover using a Gaussian mutator 100 times. The learned weights for each feature are listed in Table 3.1.

B. Clustering After defining the similarity metric for the ACFG, the next step is to generate a codebook using the unsupervised learning method. This process can be regarded as a clustering process over a collection of raw features: ACFGs, where each cluster comprises a number of similar ACFGs.

In this chapter, I use spectral clustering [106] as the unsupervised learning algorithm to generate the codebook. Formally, the spectral clustering algorithm partitions the training set of ACFGs into n sets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ so as to minimize the sum of the distance

of every ACFG to its cluster center. $c_i \in \mathcal{C}$ is the centroid for the subset S_i. I define the centroid node as the ACFG that has the minimum distance to all the other ACFGs in S_i, and the collection of all centroid nodes constitutes a code book.

Unlike traditional clustering algorithms, in which the inputs are numerical vectors, in this chapter I propose to use a kernelized spectral clustering where the input is a kernel matrix. The similarity computed in Section 3.1.1 can be used to generate the kernel matrix for the spectral clustering. Suppose the kernel matrix is $\mathbf{M}$, and each entry in $\mathbf{M}$ is a similarity score of two corresponding ACFGs. The kernelized clustering works on $\mathbf{M}$ and outputs the optimal partitions (clusters) of ACFGs in the training data.

The codebook size n would affect the bug search accuracy. To this end, I systematically study a suitable n in the bug search in the future. In order to reduce computational cost in constructing the codebook, a common strategy is to randomly sample a training set from the entire dataset. I observed that there is a significant variance in ACFG size. To reduce the sampling bias, I will collect a dataset which covers ACFGs of different functions from various architectures, then split ACFGs into separate "strata" with different size ranges. Each stratum is then sampled as an independent sub-population, out of which individual ACFGs are randomly selected. This is a commonly used approach known as stratified sampling [125].

The codebook generation is expensive. However, since the codebook generation is an offline and one-time effort, it will not detrimentally impact the runtime for the online searches. Besides, some approaches can be used to expedite this process, such as the parallelled clustering approximate clustering [21] or the hierarchical clustering algorithm [102].

C. Feature Encoding Given a learned codebook, the feature encoding is to map raw features of a function into a higher-level numeric vector, each dimension of which is the similarity distance to a categorization in the codebook. It is known as feature encoding [34].

There are two benefits for feature encoding. First, the higher-level feature can better tolerate the variation of a function across different architectures, as each of its dimensions is the similarity relationship to a categorization which is less sensitive to the variation of a binary function than the ACFG itself. This property has been demonstrated by many practices in the image search to reduce the noises from the scale, viewpoint and lighting. Second, the ACFG raw features after encoding becomes a point in the high dimensional space which can be conveniently indexed and searched by existing hashing methods. Therefore, the encoding enables a faster real-time bug-search system. I will demonstrate these two benefits in the future.

Formally, the feature encoding is to learn a quantizer $q : \mathbb{G} \rightarrow \mathbb{R}^n$ over the codebook $\mathcal{C} = \{c_1, ..., c_n\}$, where $\mathbb{G}$ is the set of all ACFG graphs following Definition 1, and $\mathbb{R}^n$ represents the n-dimensional real space. In this chapter, I discuss two approaches to derive q. For a given graph g_i, let $NN(g_i)$ represent the nearest centroid neighbors in the codebook:

$$NN(g_i) = \arg\max_{c_j \in \mathcal{C}} \kappa(g_i, c_j) \tag{3.2}$$

where κ is defined in Eq. (3.1). A common practice in image retrieval is to consider not only the nearest neighbor but a few nearest neighbors, e.g. 10 nearest neighbors [82, 140].

Bag-of-feature encoding. The bag-of-feature encoding, which maps a graph to some centroids in the codebook, represents each function as a bag of features. The bag-of-feature quantizer can be defined as:

$$q(g_i) = \sum_{g_i:NN(g_i)=c_j} [\mathbb{1}(1=j), \ldots, \mathbb{1}(n=j)]^T, \tag{3.3}$$

where $\mathbb{1}(\cdot)$ is an indicator function which equals 1 when $\cdot$ is true and 0 otherwise. Eq. (3.3) indicates that the output encoded feature will add 1 to the corresponding dimension of the nearest centroid. This representation is inspired by the bag-of-words model used in text retrieval [97], where each document is represented by a collection of terms in the English vocabulary. In analogy, in our problem, each function is represented by a collection of representative graphs in the learned codebook. After encoding each function becomes a point in the high dimensional vector space.

VLAD encoding. The drawback of the bag-of-word model is that the distance between a given graph and a centroid is completely ignored as long as the centroid is the graph's nearest neighbors. The VLAD [16] encoding was proposed to incorporate the first-order differences and assigns a graph to a single mixture component.

$$q(g_i) = \sum_{g_i:NN(g_i)=c_j} [\mathbb{1}(1=j)\kappa(g_i,c_1), ..., \mathbb{1}(n=j)\kappa(g_i,c_n)]^T, \tag{3.4}$$

Compared to Eq. (3.3), Eq. (3.4) adds the similarity information to the centroids in the encoded features. Note as our raw features are graphs, in Eq. (3.4) I use the kernelized similarity function in the VLAD encoding which is different from the traditional VLAD

defined for image retrieval. In VLAD encoding, a dimension represents the similarity to a corresponding ACFG centroid in the codebook. As a result, the vector is of latent semantic meaning that reflects a similarity distribution across all centroids in the learned codebook. Empirically I found that VLAD encoding performs better than the bag-of-feature encoding for bug search.

3.2.2 Search Engine Construction

The encoded features may be directly used in search. However, this straightforward solution may not be scalable for millions of functions in real-world applications. This section introduces a scalable solution by LSH (Locality-sensitive hashing) to scale the search. In this chapter, I utilize LSH as opposed to other indexing methods such as k-d tree, as the k-d tree may not be suitable for our problem due to its inefficiency in high-dimensional spaces especially when the codebook is large [137].

Given a query function, I first derive its encoded feature by Eq. (3.3) and (3.4), then I are interested in finding the function in a large dataset that are closest to the query with a high probability. LSH achieves this goal by learning a projection so that if two points are closer together in the feature encoding space, they should remain close after the projection in the hashing space. Following [129], given the encoded feature $q(g)$, I employ the projection functions h_i defined as:

$$h_i(q(g)) = \lfloor (\mathbf{v} \cdot q(g) + b)/w \rfloor, \tag{3.5}$$

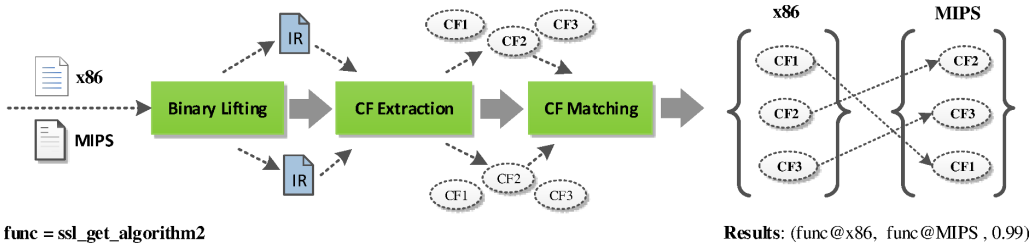


Fig. 3.5.: The overview. The inputs are binaries of `ssl_get_algorithm2` function for x86 and MIPS, which contains the vulnerability *CVE-2013-6449*. First, the two binaries are lifted into an intermediary representation (IR). Second, conditional formulas are extracted from the lifted binary function. Finally, the conditional matching works for the similarity score, and one-to-one mapping results are outputted.

where w is the number of quantized bin, $\mathbf{v}$ is a randomly selected vector from a Gaussian distribution, and b is a random variable sampled from a uniform distribution between 0 and w. In addition, $\lfloor \cdot \rfloor$ is he floor operator. Essentially, a hashing function defines a hyper-plane to project the input encoded features. For any functions $q(g_i), q(g_j) \in \mathbb{R}^n$ that are close to each other in the encoding space, there is a high probability $P[h(q(g_i)) = h(q(g_j))] = p_1$ that they fall into the same bucket. Likewise, for any functions that are far apart, there is a low probability $p_2 (p_2 < p_1)$ that they fall into the same bucket.

The locality sensitive hash of an encoded feature $q(g)$ as $lsh(g) = [h_1(q(g)), .., h_w(q(g))]$ where w is the number of hash functions. After LSH, a function is projected as a point in the hashing space. I experiment on two classical distance metrics defined in the hashing space: Euclidean distance and the cosine distance [115] in our bug search problem.

3.3 Accountable Code Search Engine

I outline our accountable code search engine in Figure 3.5. It consists of three components: binary lifting, conditional formula extraction, and conditional formula matching.

Binary Lifting. I first utilize binary lifting to convert different native machine code to the same higher-level intermediate representation (IR). The lifted binary retains semantics that are consistent with the original binary program. Our subsequent operations will be directly conducted on the lifted binary.

Conditional Formula Extraction. I apply the binary analysis techniques on the lifted binary to construct conditional formulas. I carefully handle the data dependency via pointers. Besides, not all the variables in a lifted binary function are of interest. I conduct the action point selection to filter irrelevant variables.

Conditional Formula Matching. I match functions by their unified conditional formulas. I model such a matching problem as a linear assignment problem and leverage integer programming techniques to find an optimal solution. The matching result is then a one-to-one mapping of `CF`s, in addition to a simple similarity score. Human analysts can thus inspect the in-depth mapping results to understand and verify any discovered bugs.

3.3.1 Binary Lifting

Binary lifting transforms binary code of different architectures into a common code representation to facilitate subsequent analyses. In the domain of cross-platform bug search, the work by Pewny et al. [113] also conducts binary lifting to model the basic-block level semantics. In our case, since I need to extract conditional formulas for one function, such a transformation must preserve the semantics of the entire function. To do so, I first re-

cover the control flow graph of a function, and then transform the binary code instruction by instruction following the control flow graph.

With respect to the implementation, our binary lifting is based on McSema [42], a code translation framework that translates x86 instructions to LLVM IR (Intermediate Representation). To address the problem of cross-platform bug search, I have extended McSema in two fronts: 1) multi-architecture support; and 2) function prototype based translation.

A: Support for Multiple Architectures

McSema only supports translation from x86 instructions to LLVM IR. In our use scenario, I would like to translate binary code from a wide variety of CPU architectures into LLVM IR. Therefore, I have to extend its support for other architectures. Fortunately, McSema provides a generic framework enabling us to easily support other instruction sets. In the current implementation of XMATCH, I extended its support for MIPS, since it is the popular CPU architecture for embedded systems and IoT devices.

McSema requires two steps to translate a binary function: 1) control flow graph recovery, and 2) bitcode generation. The control flow-graph recovery will disassemble a binary function, retrieve its basic blocks as well as control flow dependencies among these basic blocks. The bitcode generation walks through the control flow graph, conducts the one-by-one instruction translation and generates the LLVM bitcode file. I utilize IDA Pro to retrieve the control flow graph for a MIPS binary function. McSema translates each instruction in a x86 binary by modeling its execution semantic. I follow the similar instruction translation process. MIPS belongs to RISC instruction sets, so the amount of work for adding such

support in McSema is much simpler than that of adding support for CISC instruction sets like x86. In our case, I add less than 1K LOC in McSema, and it is one-time effort work.

B: Function Prototype Based Translation

For function call translation, McSema introduces a global ”context registers” data type and uses this as the only argument for all lifted functions. ”context register” includes all the registers under corresponding CPU architecture. At the beginning of a lifted function, McSema first allocates several local variables, and spills all the registers in the global ”context register” argument to those variables. Then the following operations are performed based on the variables. When a function returns, the ”context register” will be wrapped up with the latest variables value. At each function call site, which means a lifted function would be called, the ”context register” is first wrapped up, and then is passed to the callee as the only argument. As a result, McSema can preserve the control and data flow dependencies among functions without the need of function prototype recovery. However, this translation strategy will undermine the efficacy of XMATCH. Firstly, the generated condition formula fails to represent execution semantics on the real argument of a function without the function prototype recovery, since some flaw code logic could be related to the real argument on a function call such as memcpy. Secondly, unified function prototypes will reduce the accuracy of XMATCH, since I cannot rely on the number of arguments to further refine the search result.

To address these issues discussed above, I require McSema to translate function call instruction based on the function prototype. More specifically, there are three steps to achieve

this goal: the function prototype recovery, the function call translation, and argument passing modeling. I first recover the function prototype for a binary function, and modify the function call translation mechanism in McSema based on recovered function prototype. I also add additional IR instructions to model the argument passing to the corresponding callsite.

The function prototype includes function name, its arguments and return values. Fortunately, IDA pro has the function recovery mechanism to retrieve the function name and arguments, and it supports multiple architectures, so I utilize IDA pro to recover these information for the function prototype. I assume all functions on the function callsite have the return value during the translation. I will take the screening process in Section 3.3.2 to reduce the impact of fake return value on the generated conditional formulas.

When McSema translates the function call instruction, it will predefine the number of arguments to translate. In its original design, it always assumes the number of argument to be 1. In our scenario, the number of arguments is defined by the recovered function prototype. I will create argument variables with the same number of arguments defined in the function prototype. I also create the return variable for each lifted function.

Since the function call instruction translation has been changed in McSema, I also need to add corresponding argument passing instructions to preserve the data-flow dependency between the caller function and its callee. Modeling the argument passing depends on the calling convention type of the original binary function. I model calling conventions by their types, and check the calling convention type by matching our modeled patterns. With the calling convention type, I add corresponding argument passing IR instructions before the function call instruction.

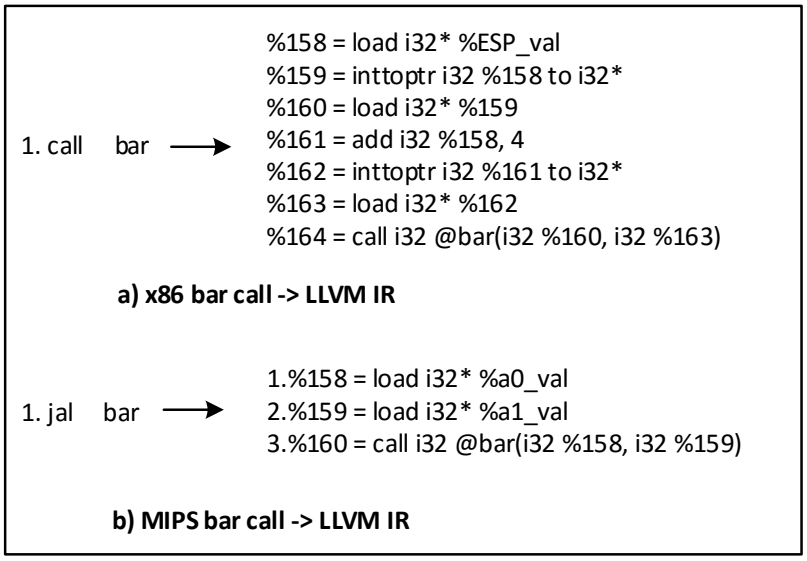


Fig. 3.6.: The example of the code translation for the `call` instruction.

Figure. 3.6 shows a concrete example of how to translate a call on both x86 and MIPS architecture. In this example, from the analysis in IDA, I know that the function `bar` has two parameters, so at the call site, `call bar` and `jal bar` will be translated into a number of IR instructions shown on the right hand side of the figure. Instructions before the call instruction describe how arguments are passed into the corresponding function.

C: Other Issues

McSema does not support translation for all sorts of x86 instructions. For instance, it only supports a small portion of floating point instructions. However, McSema is well documented and it is not difficult to add support for other instructions that are needed. In our case, it is important for us to provide support for the conditional branches which is necessary to conditional formula extraction, so I add support this kind of floating point

instructions that McSema has not supported. I believe that supporting all instructions is engineering work and leave it as future work.

3.3.2 Conditional Formula Extraction

I conduct static analysis directly on the lifted function to extract its conditional formulas. More specifically, I conduct intra-procedural dataflow analysis to construct formulas for *actions*, and perform the path slicing to retrieve the corresponding *conditions*. All static analysis are conducted on top of LLVM framework [5].

A: Action Construction

An *action* is a data-flow equation on a specific IR variable, which serves as a function output. To construct an action, I first discover all the function outputs (I call them "action points") that have external impacts. Then, starting from each action point, I compute the use-def chain to evaluate the reachability from function output to inputs. Eventually, I fold all the IR statements on each trace to produce an data-flow equation as an action for the corresponding action point.

Action Point Selection. Intuitively, I can compute data-flow equations for any IR variables hosted in a function. However, the lifted function still keep many architecture specific variables, such as `ESP_val` in Fig. 3.6a) and `a0_val` in Fig. 3.6b). This will make the generated conditional formulas to be dramatically different across architectures. Therefore, I are more interested in the stable output states, which indicate consistent program behav-

ior. To this end, I aim to calculate backward data-flow from three types of function outputs: 1) return values, 2) memory variable, 3) function calls.

1) **Return value.** I conduct a conservative analysis to identify the variables that hold return values, even if I assume all functions in a lifted binary have the return values during the binary lifting process discussed in Section 3.3.1. First, I seek IR variables with specific register names. This is due to the fact that a certain architecture uses specific registers to hold return values and IR variables, though lifted from binary, still preserve the original register names. Second, among these candidate variables, I further search for those that are never redefined in the same function. I then consider these variables to contain the return values.

2) **Memory variables.** A function can also write to memory. This is translated into a memory write operation in lifted binary. Therefore, I obtain memory variables by first searching for memory write instruction $storeinst$ in LLVM-IR. Next, I perform value-set analysis [19] to determine the memory region a pointer points to. Once a memory region is not updated in a function, its pointer is now pointing to the actual output, and thus can be considered as an action point.

3) **Function calls.** A function may call another function. If the caller function uses or checks the return value of the callee function, then the callee function will be eventually included in the data-flow expression of the variable that uses the return value. If the return value of the callee function is never used inside the caller function, I will treat it as an action point.

Data-flow Analysis. Once I have discovered all the action points, I then compute backward data-flow for each one of them. To this end, I first perform use-def chain analysis. The use-def chain analysis needs to consider two types of variables in a lifted binary: register variables and memory variables. LLVM IR has already been in SSA (Static Single Assignment) form, so I can directly retrieve the use-def chain for register variables. However, memory variables are address-taken variables which are not in SSA form in LLVM IR. I need promote memory variables to register variables to obtain their use-def chains.

Memory promotion is to promote memory references to register references. It collects all possible memory variables by finding pointers holding their addresses. The pointer variable is transformed to a register variable by rewriting the lifted IR function, and traversing the function in depth-first order to rewrite all its uses as appropriate. This follows the standard SSA construction algorithm.

The binary lifting translates the pointer variable by using "inttoptr" instruction. I find all potential pointer variables by scanning the whole binary function for this instruction. The next step is to rewrite the pointer as well as its all uses. I need conduct the value-set analysis [19] on pointer variables to find all its uses. Since the memory variable is not in SSA form, it is hard for us to know which pointers share the same value. To this end, each pointer is labeled by its address, the `a-loc` (known as "abstract location") of the memory region that it accesses. Hence, the reaching-definition analysis on the pointer variable aims to track its `Kill` and `Gen` sets of `a-locs`. The result of this analysis in effect recovers the data-flow dependencies among memory regions accessed by pointers, and therefore help bridge the disconnected data-flows. After recovering the data-flows among pointer variables, I convert them into SSA form by the standard SSA construction algorithm.

I utilize the data-flow expression on the index of a pointer variable to determine its `a-loc`. This is also widely adopted in other works [20]. The data-flow expression on the index describes how the index is computed. Pointers of the same memory variable should share the similar data-flow expression. I utilize a theorem prover [11] to further unify the data-flow expression for `a-loc`.

There are three types of pointers: global pointers, local pointers and pointers from function arguments. The `a-loc` of a global pointer is a constant address which is different cross architectures. I rename global variables into "G1 to GN" by orders of their addresses. I also rename the local pointers into "L1" to "LN" by their offsets on the stack. Similarly, I also rename pointers from arguments into "a0" to "aN".

I will rewrite the IR function by renaming all pointer variables according to their `a-loc`s, and by traversing the function in depth-first order to rewrite all its uses as appropriate. The result IR function has the complete use-def chain for both register variables and memory variables.

Action Generation. The action generation works on the rewritten IR function. Starting from an action point, I fold all the instructions on every single data-flow path and produce a data-flow equation. One such equation is further simplified using theorem prover and the result is then considered to be one action.

Running Example. Figure 3.7 illustrates the action construction process. For readability purpose, I utilize pseudo code instead of LLVM-IR in the demonstration. Figure 3.7(a) shows the IR in SSA (Static Single Assignment) form. The action point in this example

a) Pseudo LLVM-IR code

1. t1 = esp - 4
2. t2 = t1 + 8
3. t3 = *t2
4. If t3 > 0 goto 5 else goto 6
5. t4 = t3 + 2
6. t5 = t3 + 1
7. *t2 = ϕ(t4, t5)
8. t6 = esp -4
9. t7 = t6 + 8
10. eax = *t7

b) Incomplete use-def chain

1. t1 = esp - 4
2. t2 = t1 + 8
3. t3 = *t2
5. t4 = t3 + 2
7. t2* = t4
8. t6 = esp -4
9. t7 = t6 + 8
10. eax = *t7

c) Use-def chain on pointers

3. t3 = a0
5. t4 = a0 + 2
7. a0 = t4
10. eax = a0

d) Data-equations and a-loc info

Variable	Data-flow equation	A-loc
*t2	*((esp-4) + 8)	a0
*t7	*((esp-4) + 8)	a0
eax	a0 +2; a0 + 1	

Fig. 3.7.: The example of lifted IR code and generated statements for its variables.

is `eax`, which holds the return value. Figure 3.7(b) shows the incomplete use-def chain without the memory variable promotion on `*t2` and `*t7`. We can see that the data flow between `eax` and `t4` is missing, so I cannot know the current value of `eax` is `[esp+4]+2`. Figure 3.7(c) and (d) illustrates the effectiveness of memory variable promotion. By analyzing data-flow equation on pointers `*t2` and `*t6`, I identify that these two pointers access the same `a-loc a0`. Such a discovery helps us further connect the use-def chain from eax to `*t2`. Figure 3.7(d) demonstrates the updated use-def chain by considering pointers. Eventually, I compute two data-flow equations for the action point eax: `a0+1` and `a0+2`, each of which is generated from one single path.

a) Action

Variable	Action
eax	a0 + 2
	a0 + 1

d) Conditional formula

Action	Condition	Value
a0+2	a0 > 0	True
a0+1	a0 > 0	False

b) Slicing criterion for action (a0+2)

(eax=t7*,{t7,t6,t2*,t4,t3,t1})

c) Path slice

```
1.  t1 = esp - 4          5.  t4 = t3 + 2
2.  t2 = t1 + 8           7.  *t2 = t4
3.  t3 = *t2              8.  t6 = esp -4
4.  If t3 > 0 goto 5      9.  t7 = t6 + 8
    else goto 6           10. eax = *t7
```

Fig. 3.8.: The condition generation for IR variable `eax`.

C: Condition Extraction

I further utilize the path slicing algorithm [81] to generate conditions for each action. In our scenario, the path slicing is used to extract conditions for the specific path holding the action in the lifted binary function.

Given the action and the computed data-flow, I set the slicing criterion to include all variables in the action. With the slicing criterion, the path slicing algorithm will trace backwards to find the path slice, which contains all the variables in the slicing criterion. I extract all the comparison variables from the path slice and generate predicate expressions for these variables. Each predicate expression includes the condition expression and its boolean value that will cause the action to be executed. In LLVM-IR, the comparison variable is the first operand of the branch instruction, if it is the conditional jump. I generate the data-flow equation for the condition variable to get the condition expression on this basic

block. I can obtain the boolean value by checking the successor block on the path. If its address is the second operand of the branch variable, the boolean value is `true`. Otherwise, `false`. If the boolean value is false, I will negate the condition expression. I make a conjunction of the discovered predicate expressions on the path slice as the condition for the action v.

Running Example. Figure 3.8 demonstrates the condition generation process for actions of `eax`. Figure 3.8(a) lists two actions for `eax`: `a0+2` and `a0+1`. This indicates that the eax may hold two different values depending on which data-flow path to take. Figure 3.8(b) shows the path criterion of the two actions, each of which involves 3 IR variables. The corresponding path slices for the action are also presented in Figure 3.8(b). This slices include not only the data-flow but also all the conditions. I then can walk through these path slices, extract all branch variables and make a conjunction of their data-flow equations. The result becomes the condition for this action. The outputs of condition generation for action `a0+1` and `a0+2` are listed in Figure 3.8(c).

Arch	**Condition Expression**
X86	((%t3-0)==0, ((%t3-0)<0) == and(lshr(and
	(xor([esp-8],3),xor(xor([esp-8],3)), (%t3-0))),31),1)))
MIPS	(%t3-0)==0

Fig. 3.9.: The demo example for condition expression in x86.

Architecture-specific Conditional Expression. For some architectures, such as x86, I cannot directly use the generated condition for comparison, since its conditions are data equations on status registers. As a result, the condition expression is completely different

from those in other architectures, such MIPS. For example, Figure 3.9 shows the condition expression for comparison variable $\%t3$ in x86 and MIPS. I address this issue by building the model for each condition expressions on status registers. Then I map them into corresponding real conditional expressions.

3.3.3 Conditional Formula Matching

The accountable code search is to match two functions by their conditional formulas. It consists of two steps. Firstly, I compute the matching cost of two *CF*s. Secondly, I seek the optimal matching among *CF*s by selecting the minimum matching cost between two sets of *CF*s, and then output the similarity score of two functions.

Intuitively, one can utilize string edit distance to calculate the match cost between two *CF*s. However, two semantically equivalent *CF*s may appear to be different due to distinctive ordering. For instance, **((**$a > 0$**) && (**$b > 0$**)) /** *ret = 0x20800* will be considered unequal to **((**$b > 0$**) && (**$a > 0$**)) /** *ret = 0x20800*, even though they share the same behavior-level semantics because of the commutative property. To avoid the reordering problem, I instead match two *CF*s by their AST structure. Since the AST is a tree-like structure, I adopt the graph edit distance to calculate the cost to transform between two *CF*s.

I utilize the algorithm presented in [120] to compute the graph edit distance $ged(cf_i, cf_j)$. In our case, not all nodes are inter-changeable. For example, the condition related node cannot be replaced by action related node. Therefore, in the pre-computed mapping cost matrix in the algorithm, I assign the infinite number for the mapping cost between action and condition node.

The distance of two CFs will calculate the match cost of two functions. Suppose I are given two functions f_1 and f_2, where f_1 contains the CF set $\{cf_1, cf_2, \ldots, cf_n\}$ and f_2 holds the set $\{\hat{cf}_1, \hat{cf}_2, \ldots, \hat{cf}_m\}$. Let $\mathbf{m}_{ij}$ be the matching factor for a CF pair: if cf_i matches $\hat{cf}_j$, $\mathbf{m}_{ij} = 1$; otherwise, $\mathbf{m}_{ij} = 0$. Hence, all the matching factors form a matching matrix $\mathbf{M}_{n\times m}$, which demonstrates how these two functions correspond to each other.

Following the graph edit distance, I define the *function distance* as the minimum distance of all matched CF pairs between f_1 and f_2. As we see, finding the function distance is equivalent to finding the $\mathbf{M}$ that minimizes $\sum_{(i,j)\in\{\mathbf{m}_{i,j}=1\}} dist(cf_i, \hat{cf}_j)$. In other words, we need to find the best match (with the minimum distance) between two functions. Note the greedy approach, which matches the CF in each function individually, cannot be applied as it can only produce the suboptimal solutions. To find the global optimal solution, I formulate the following objective function:

$$\begin{aligned} \textbf{min} \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} \mathbf{m}_{ij} \times dist(cf_i, \hat{cf}_j) \\ \textbf{subject to} \quad & \sum_{i=1}^{n} \mathbf{m}_{ij} = 1, \forall j \in \{1, m\} \\ & \sum_{j=1}^{m} \mathbf{m}_{ij} = 1, \forall i \in \{1, n\} \\ & \mathbf{m}_{ij} \in \{0, 1\} \forall i \in \{1, n\}, \forall j \in \{1, m\}. \end{aligned} \tag{3.6}$$

The objective in Eq. (3.6) is to calculate the distance between matched CF pairs, and minimize this value. The constraints indicate every CF in the functions can be matched only once. Based on Eq. (3.6), I can formally introduce the function distance.

Definition 2. The function distance of f_1 and f_2 is the minimum distance of all matched CF pairs between f_1 and f_2. Let $\mathbf{M}^*$ represent the optimal solution of Eq. (3.6), i.e. the best match between the two functions, the distance is calculated from:

$$dist(f_1, f_2) = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbf{m}_{ij}^* \times dist(cf_i, \hat{cf}_j), \tag{3.7}$$

where $\mathbf{m}_{ij}^*$ is an element in the optimal solution $\mathbf{M}^*$.

The function distance provides a finer-grained comparison for two functions. It not only quantifies the dissimilarity between two functions (minimum distance transforming one set of CFs into the other), but also explains how the statements of the two functions are matched together. The best match is stored in $\mathbf{M}^*$, where for all $\mathbf{m}_{ij}^* = 1$, I can plot a best match between the statement c_i in f_1 and the statement c_j in f_2. By tracking the conditional formulas in a vulnerable function, this matching helps human analysts locate potential vulnerable statements in other functions.

According to Definition 2, to calculate the function distance, we need to find the optimal solution of Eq. (3.6). Fortunately, the problem in Eq. (3.6) is a well studied problem called integer linear programming which can be efficiently solved by various techniques such as constraint relaxation [66, 134]. In this chapter, I leverage the solution in [90] to solve the problem. Our experimental results show that the employed matching algorithm is efficient.

The function distance quantifies the difference between two functions which could be roughly proportional to the sum of the functions sizes (in bytes). Thus, two larger functions are likely to be more dissimilar. To reduce the bias, I normalize this absolute distance by

the total length of conditional formulas in the two compared functions. I define the *function similarity* of f_1 and f_2 as:

$$sim(f_1, f_2) = 1 - \frac{dist(f_1, f_2)}{\sum_{i=1}^{n} len(cf_i) + \sum_{j=1}^{m} len(\hat{cf}_j)}, \tag{3.8}$$

where len counts the string length of each conditional formula. Its enumerator is the function edit distance and its denominator denotes the largest possible string edit distance between two completely different functions.

4. APPLICATION I: SCALABLE VULNERABILITY SEARCH IN IOT DEVICES

In this chapter, I demonstrate the vulnerability search ability of our scalable code search engine. I implemented a proof-of-concept scalable code search engine GENIUS, and compared it with existing state-of-the-art bug search approaches.

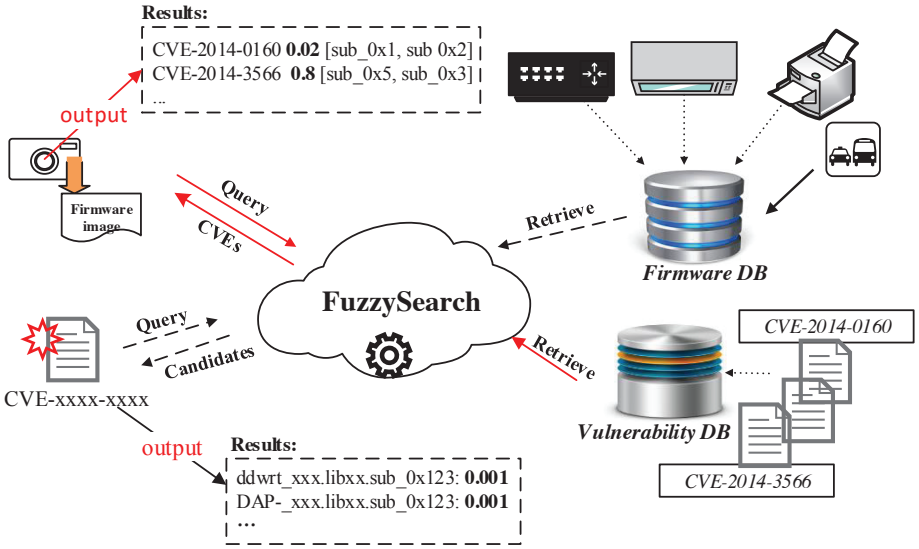


Fig. 4.1.: **The deployment of GENIUS.**

4.1 Deployment

Figure 4.1 shows two use scenarios for GENIUS. 1) *Scenario I*: Given a device repository, GENIUS will index functions in the firmware images of all devices in the repository based upon their CFGs. When a new vulnerability is released, a security professional can

use GENIUS to search for this vulnerability in their device repositories .GENIUS will generate the query for the vulnerability and query in the indexed repository. The outputs will be a set of metadata about all potentially infected devices with their brand names, library names and the potentially vulnerable functions. All outputs will be ranked by their similarity scores for quick screening of the results. 2) *Scenario II*: Security professionals may upload unseen firmware images that do not exist in the repository for a comprehensive vulnerability scan. In this case, GENIUS will index these firmware images for the security professionals. As a result, they can simply query any vulnerabilities in our vulnerability database. GENIUS will retrieve the most similar vulnerabilities in the existing indexed firmware images and output metadata of all potentially vulnerable functions including their names, library names holding these functions and firmware device types where these functions are used. Again, all outputs will be ranked by their similarity scores to facilitate quick screening of the results.

4.2 Experimental Evaluation

In this section, I empirically evaluate GENIUS with respect to accuracy, efficiency, and scalability. First, I briefly describe the experiment setup and the data sets used in our evaluation (Sections 6.6.1 and 4.2.2). Second, I conduct a systematic baseline comparison against the existing bug search methods in terms of the accuracy and efficiency in the cross-platform setting (Section 4.2.3). Third, I evaluate GENIUS on 33,045 firmware images and demonstrate its scalability (Section 4.2.5). Finally, I present two case studies to show the deployment of GENIUS under realistic conditions (Section 4.2.6).

4.2.1 Experiment Setup

I wrote the plugin to the disassembler tool IDA Pro [75] for ACFG extraction. I implemented codebook generation, feature encoding in python, and adopted Nearpy [8] for LSH hashing and search. I utilized MongoDB [7] to store the firmware images and encoded features. Our experiments were conducted on a server with 65 GB memory, 24 cores at 2.8 GHz and 2 TB hard drives. All the evaluations were conducted based on four types of datasets: 1) baseline evaluation dataset; 2) two public firmware images; 3) 33,045 firmware images and 4) the vulnerability dataset.

4.2.2 Data preparation

***Dataset I* – Baseline evaluation.** This dataset was used for baseline comparison, and all functions in this dataset has known ground truth for metric validation. I prepared this dataset using BusyBox (v1.21 and v1.20), OpenSSL (v1.0.1f and v1.0.1a) and coreutils (v6.5 and v6.7). All programs were compiled for three different architectures (x86, ARM, MIPS; all 32 bit) using three compiler versions (gcc v4.6.2/v4.8.1 and clang v3.0). I also enabled four optimization levels (O0-O3) for each version of a compiler. These settings have been used in existing studies as well [113]. I kept the symbol names during compilation which allowed us to maintain ground truth for evaluations.

These different compilation combinations resulted in a dataset of over 568,134 functions. As several of the existing techniques could not scale to a dataset of this size, I randomly sampled 10,000 functions from this dataset as the baseline dataset, and used that for baseline evaluation. Each function in the baseline dataset has at least two instances:

one for query and another for search. The remaining functions in Dataset I were used for codebook generation.

***Dataset II* – Public dataset.** Recent work such as Pewny et al [113] and Eschweiler et al [53] used the same public dataset based upon two publicly-available firmware images for baseline comparison [12, 13]. I also evaluated GENIUS using this dataset to provide for fair comparison with the state-of-the-art systems.

***Dataset III* – Firmware image dataset.** This dataset of 33,045 firmware images was collected from the wild. I used it to evaluate the scalability of GENIUS. The images in this dataset were collected from three sources: 9,000 firmware images from [35], the entire dataset from [38] and 500 randomly selected images from our own crawl of the DDWRT ftp site [3]. Of the total 33,045 firmware images collected, I successfully unpacked 8,126 images from 26 different vendors. The vendors include such as ATT, Verizon, Linksys, D-Link, Seiki, Polycom, TRENDnet. The product types from each vendor include IP cameras, routers, access points and third-party or open-source firmware.

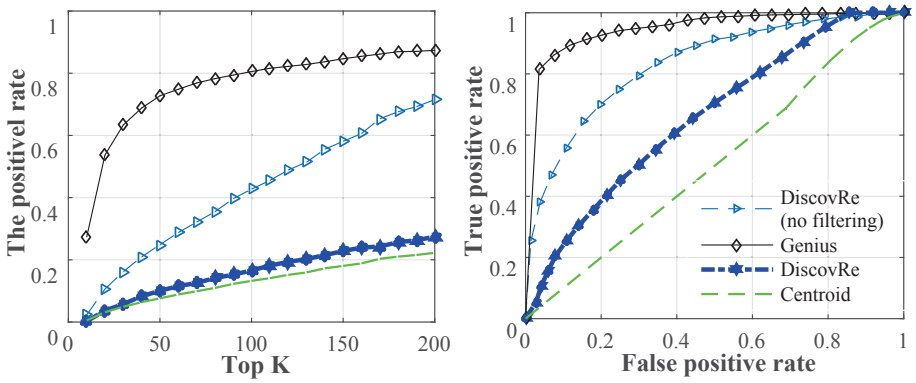
***Dataset IV*–The vulnerability dataset.** This dataset is a mapping between CVE numbers and the corresponding functions that actually introduced the vulnerabilities. To construct a query which can be used by GENIUS for bug search, I need to find binary code for these vulnerable functions. While other works have already investigated construction of vulnerability databases [110], none of them fit our purposes; they cannot extract the binary code for vulnerable functions required by GENIUS. As a result, I created a freely available vulnerability database for this effort and for the broader research community.

To build this database, I mined official software websites to collect lists of vulnerabilities with the corresponding CVE numbers. I were also able to retrieve information about the software commits to fix the vulnerabilities, which provided us the vulnerable function names and symbols. I then downloaded the source code for the vulnerable versions of the software, compiled the source and extracted the vulnerable functions from the binary code using the symbol names. I then used GENIUS to generate higher-level features for each vulnerable function. In the end, I utilized MongoDB [7] to build the database which stored the vulnerable functions and their corresponding feature vectors for later use. In our evaluation, I were only interested in vulnerabilities related to libraries widely used in firmware devices. I selected `OpenSSL` for demonstration, since it is widely used in IoT devices. The resulting vulnerability database includes 154 vulnerable functions.

4.2.3 Cross-Platform Baseline Comparison

I first evaluated GENIUS with baseline methods under the cross-platform setting. All evaluations in this subsection were conducted under the baseline dataset in Dataset I and Dataset II. Since each function name has multiple instances in Dataset I, I collected the query set Q by randomly selecting one instance for each function name, and considered the rest of the baseline dataset as a codebase. The codebook of GENIUS in this part is also trained on Dataset I, and its size is 16 for all baseline evaluations.

Evaluation Metrics. I used two metrics to evaluate the accuracy of the proposed and baseline methods: the recall rate (A.K.A true positive rate) and false positive rate. In the code search scenario, the search results are a ranked list. For each query q, there are m



a) Recall rates across different threshold K b) ROC curves for different approaches

Fig. 4.2.: **Baseline comparison for accuracy on Dataset I.** K **means that I consider retrieved candidates on top K as positives** Two figures share the same legends.

matching functions out of a total of L functions. If I consider the top-K retrieved instances as positives, the total number of correctly matched functions, μ, are true positives, and the remaining number of functions in the top K, that is $K - \mu$, are false positives. Based on the definitions of recall and false positive rate, the recall rate is calculated as $recall(q) = \frac{\mu}{m}$ and the false positive rate is $FPR(q) = \frac{(K-\mu)}{L-m}$.

Preparation of Baseline Systems. I prepared three representative, state-of-the-art, cross-architecture bug search techniques to establish our evaluation baseline: discovRe [53], Multi-MH and Multi-k-MH [113] and a centroid based search [36]. Our first task was to prepare versions of these solutions for this evaluation.

- *discovRe* [53]: Due to unavailability of the source code, I reimplemented discovRe[1] and set the iteration limitation to be the same (i.e., 16*max($|G_1|, |G_2|$)) as the one used in their work work. I evaluated GENIUS against two versions of discovRe: the original version with pre-filtering and the version without pre-filtering. For the pre-

[1]I contacted the author of discovRe for comparison by providing their search results, but they have not provided us results yet.

filtering version, I set the threshold to 128 as specified in [53]. The version without pre-filtering uses only their graph matching metric for search.

- *Multi-MH and Multi-k-MH* [113]: Their source code is not available. Due to the complexity, it was less possible for us to reimplement their approach within a reasonable amount of time. Fortunately, discovRe has already conducted a thorough baseline comparison against these two approaches and published the results. The binaries used for the evaluation are also available online. Hence, I evaluated GENIUS on the same setting for this baseline comparison, and compared the published numbers on the benchmark.
- *Centroid* [36]: The centroid-based approach is known for its efficiency with respect to the Android malware clone and repackage problem [36]. I implemented a centroid-based bug search system for IOT devices. Note that while the centroid method is not directly designed for cross-platform code matching, it is still meaningful to compare it with GENIUS in terms of efficiency and accuracy.

A. Accuracy comparison To evaluate the efficacy of GENIUS, I first conducted thorough comparisons with DiscovRe and Centroid on Dataset I, since I have reimplemented these two approaches. I compared with the published results of Multi-MH and Multi-k-MH on Dataset II.

The first round of evaluations worked on the baseline dataset in Dataset I. I randomly selected 1000 functions as queries to feed into the target approach, and evaluated search results in terms of two metrics. Fig. 4.2a) lists the average recall rates for 1,000 queries across different thresholds of K, where the the y-axis indicates the recall for the corresponding

K values. We can see that GENIUS significantly outperforms the baseline methods for every value of K. For example, GENIUS ranks 27% functions at top 1, whereas discovRe only ranks 0.5%. I also found that the performance of discovRe is worse than the version without pre-filtering. Fig. 4.2b) shows the ROC curves for each approach. This was the macro-average result across 1,000 queries. We can see that the ROC curve of GENIUS is better than those of the baseline approaches, especially when the false positive rate is small. The results in Fig. 4.2 (a) and (b) substantiate that GENIUS can achieve even better accuracy than the state-of-the-art methods.

I inspected search results and found that the performance of GENIUS is because of the salient and robust feature representation learned on top of the ACFGs. As an example, `ssl3_get_message` was ranked at top 1 by GENIUS but ranked below 40th by baseline methods because its CFG extracted from the function of X86 and MIPS was changed. Our method managed to capture the change and thus showed better results. I also analyzed the cases where GENIUS yield worse results than baseline methods. I hypothesize the reason is about the quality of the learned codebook. We will discuss it in Section 4.2.4.

Table 4.1: Comparison with Multi-MH and Multi-k-MH, discovRE, Centroid with the propose method for OpenSSL. Each cell contains the rank, separated by the colon, for both vulnerable functions: heartbeat for TLS and DTLS.

	Multi-MHTLS [113]		**Multi-k-MH** [113]		**discovRE** [53]		GENIUS		**Centroid** [36]	
From ->To	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**	**TLS**	**DTLS**
MIPS → DD-WRT	1:2	2:4	1:2	1:2	1:2	1:2	1:2	1:2	46:100	87:99
MIPS → ReadyNAS	1:2	6:16	1:2	1:4	1:2	1:2	1:2	1:2	88:190	678:988
x86 → DD-WRT	70:78	1:2	5:33	1:2	1:2	1:2	1:2	1:2	97:255	102:89
x86 → ReadyNAS	1:2	1:2	1:2	1:2	1:2	1:2	1:2	1:2	145:238	333:127
Query Normalized Avg. Time	0.3s		1 s		4.1×10^{-4} s		1.8×10^{-6} s		1.4×10^{-6} s	

Table 4.2: Baseline comparison on preparation time.

			Preparation Time in Minutes				
Firmware Image	**Binaries**	**Basic Blocks**	**Multi-MH**	**Multi-k-MH**	**discovRE**	GENIUS	**Centroid**
DD-WRT r21676 (MIPS)	143 (142)	329,220	616	9,419	2.1	4.9	3.2
ReadyNAS v6.1.6 (ARM)	1,510 (1,463)	2,927,857	5,475	83,766	54.1	89.7	69.6

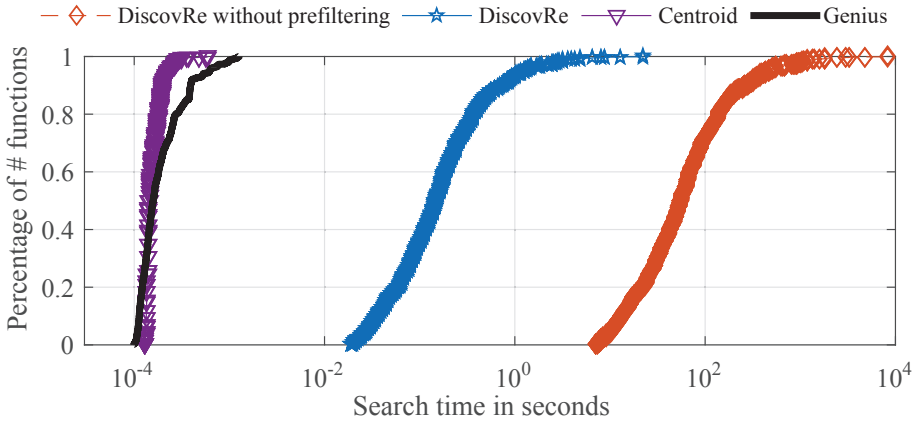


Fig. 4.3.: **The CDFs of search time on Dataset I.**

B. Efficiency comparison I conducted efficiency comparison in terms of online search and offline preparation. For online search, I evaluated on both Dataset I and II. For offline preparation efficiency, I evaluated Dataset II as a demonstration.

Offline Preparation Efficiency. The preparation includes ACFG extraction and feature encoding. Table 4.2 shows the aggregation of preparation time for the phases of GENIUS on Dataset II. We can see that GENIUS outperforms Multi-MH and Multi-k-MH. DiscovRe only considers the control flow graph extraction time, whereas GENIUS needs extra time to encode these graphs. Even if GENIUS is slower than DiscovRe at the preparation stage, as the preparation is an offline stage and only an one-time effort, it is reasonable to sacrifice some preparation time for the online search efficiency.

Online Search Efficiency. Similar to the accuracy comparison discussed above, I evaluated the online search efficiency on Dataset I and II, respectively. I first conducted the search on Dataset I, searched all of the functions in the dataset and recorded their search times for each target approach. Fig. 4.3 lists the Cumulative distribution function (CDFs) of search time for the four approaches on Dataset I, where the x-axis plots the search time in seconds. We can see that GENIUS and the centroid-based approach have least search time.

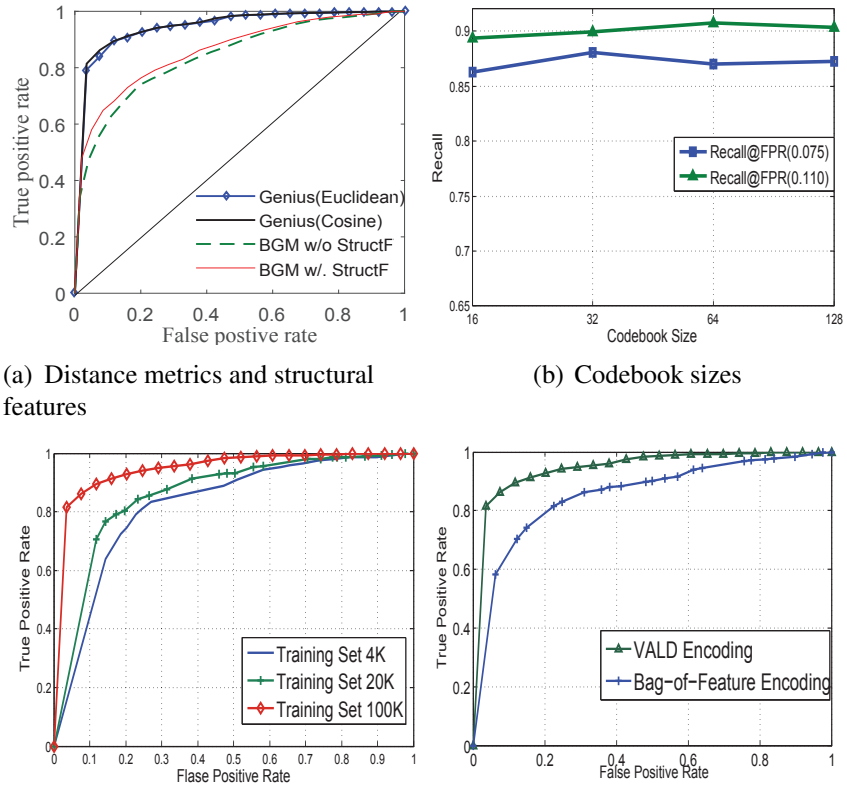
DiscovRe, on the other hand, has the longest search time because it requires expensive online graph matching. In the best case, discovRe takes 10 ms for a query, whereas GENIUS only requires 0.1 ms to return more accurate results. Unsurprisingly, I also found that the version of discovRe without pre-filtering has even worse performance. It required nearly 2 hours for a single query in the worst case, and was still less accurate than GENIUS. Although the centroid approach had comparable efficiency with GENIUS, as previously mentioned, centroid significantly under-performs GENIUS in terms of the accuracy.

I also conducted a second round evaluation on Dataset II for all baseline approaches. I utilized the search time for the Heartbleed vulnerability as the metric. Table 4.1 lists the search results. It shows that GENIUS is orders of magnitude faster than Multi-MH, Multi-k-MH and discovRE. This demonstrates that GENIUS outperforms most of the existing methods in terms of efficiency.

4.2.4 Parameter Studies

I studied the parameter's impact on the accuracy of GENIUS under different settings. The parameters for evaluation included the structural features used in bipartite graph matching, the codebook size, the size of training data for codebook generation, and the feature encoding methods. All evaluation settings were conducted on Dataset I.

A. Distance metrics and structural features. To verify the contribution of the proposed structural features, I conducted bipartite graph matching experiments with and without structural features. As shown in Fig. 4.4a), the matching with structural features outper-



(a) Distance metrics and structural features

(b) Codebook sizes

(c) Training set size

(d) Feature encoding

Fig. 4.4.: **Accuracy comparison with different parameter settings. a), c) and d) are ROC curves**

forms the matching without it. Besides, I also evaluated two distance metrics used in the LSH. Results show that the cosine distance performs better than the Euclidean distance.

B. Codebook sizes. I created codebooks of different sizes and studies their search accuracy. I evaluated the accuracy in terms of the recall rate at two representative false positive rates. Fig. 4.4b) illustrates the results for the codebooks of 16, 32, 64, and 128 centroids. We can see that the codebook size seems not having a significant influence on the accuracy of GENIUS. This result provides an insight that allow us to reduce codebook preparation time by using a smaller codebook $n = 16$.

C. Training data sizes. Another important parameter is the size of training set used to generate codebook. I selected training data samples of different sizes to generate the codebook

for search. Fig. 4.4c) shows their search results. We can see that the more samples used for training, the better GENIUS performed, but the increase in accuracy becomes saturated when the training data is sufficiently large, in our case up to 100 thousand functions. This is consistent with observations from image retrieval methods.

D. Feature encoding methods. I discussed two feature encoding methods in Section 3.2: bag-of-feature and VLAD encoding. I compared their impacts on the search accuracy while fixing other parameters. Fig. 4.4d) illustrates the ROC curves using two encoding methods. As I can see, VALD performs better than Bag-of-Feature encoding. This observation suggests considering the first-order statistics is beneficial for bug search problem. As the computational cost is similar between VALD and bag-of-features, I recommend using VALD feature encoding in practice.

4.2.5 Bug Search at Scale

I evaluated the scalability of GENIUS on Dataset III, which consists of 8,126 firmware images containing 420,336,846 functions, in terms of the preparation phase and search phase. I investigated the time consumption for each stage to demonstrate that GENIUS is capable of handling firmware images at a large scale.

I encoded 1 million functions randomly selected from Dataset III and collected the preparation time for each of them. The preparation time included the control flow graph extraction and graph encoding time. Fig. 4.8a) demonstrates the Cumulative Distribution Function (CDF) of time consumption for randomly selected 1 million query functions. We can see that nearly 90% of the functions were encoded in less than 0.1 seconds. Addition-

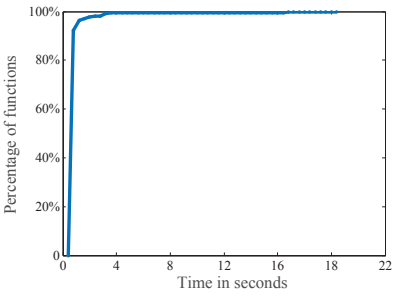


Fig. 4.5.: The CDF of preparation time over#1 million functions

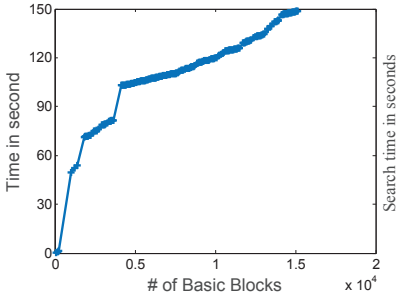


Fig. 4.6.: The preparation time cross different size of CFG

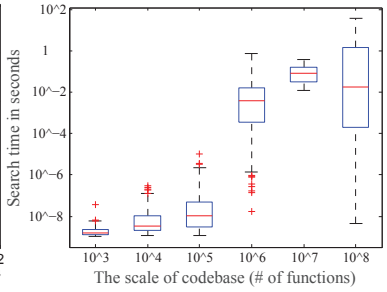


Fig. 4.7.: The search time crossscales of fimware codebases(# of funcitons

Fig. 4.8.: **The breakdown of the performance for** GENIUS**.**

ally, less than 10% of the functions needed more than 4 seconds to encode. This is because these functions have more than 1000 basic blocks, and thus take longer to encode. The prepartion time across different sizes of ACGFs is illustrated in Fig. 4.8b).

I further evaluated the search time for GENIUS in the large scale codebase. I partitioned Dataset III into six codebases of different scales from $s = 10^3$ to $s = 10^8$, where s is the total number of functions in the codebase. GENIUS was tested against 1 to 10,000 sequentially submitted queries. Fig. 4.8c) shows the log-log plot of the time consumption for GENIUS at the online search phase. As I can see, the search time grows sublinearly according to the increase of the codebase size, and the average search time observed was less than 1 second for a firmware codebase of about 100 million functions.

4.2.6 Case Studies

I also evaluated the efficacy of GENIUS in real bug search scenarios. Case studies were conducted on GENIUS for the two use scenarios discussed in Section A.2.

Table 4.3: Case study results for Scenario II

DIR-810L_REVB_FIRMWARE_2.03B02			DIR-810L_REVB_FIRMWARE_2.02.B01		
CVE	**Patched**	**Vulnerability Type**	**CVE**	**Patched**	**Vulnerability Type**
CVE-2016-0703	No	Allows man-in-the-middle attack	CVE-2015-0206	No	Memory consumption
CVE-2015-1790	No	NULL pointer dereference	CVE-2014-0160	Yes	Heartbleed
CVE-2015-1791	Yes	Double free	CVE-2015-0289	No	NULL pointer dereference
CVE-2015-0289	No	NULL pointer dereference	CVE-2016-0797	No	Heap memory corruption
CVE-2014-8275	No	Missing sanitation check	CVE-2016-0798	No	Memory consumption
CVE-2015-0209	No	Use-after-free	CVE-2014-3513	No	Memory consumption
CVE-2015-3195	No	Mishandles errors	CVE-2014-3508	No	Information leakage
#	#	#	CVE-2015-0206	No	Memory consumption
#	#	#	CVE-2014-8275	No	Missing sanitation check

With the aid of case studies, I demonstrated how GENIUS would work in the real world to facilitate the vulnerability identification process.

Scenario I. In this scenario, I conducted a vulnerability search on Dataset III of 8,126 images using vulnerability queries extracted from Dataset IV. I performed a comprehensive search for two vulnerabilities (*CVE-2015-1791* and *CVE-2014-3508*), which took less than 3 seconds. I then manually verified the vulnerability authenticity for the returned candidate functions. I disassembled the binary code for each candidate, and looked into their semantics to check whether they were patched or not. Due to the workload of manual analysis, I only verified the top 50 candidates for the two selected vulnerabilities. I found 38 potentially vulnerable firmware devices across 5 vendors, and confirmed that 23 were actually vulnerable. I also contacted these product vendors for further confirmation. The following gives the detailed discussion about search results.

CVE-2015-1791. This vulnerability allows remote attackers to cause a denial of service (double free and application crash) on the device. In the top 50 candidates, I found that there were 14 firmware images potentially affected by this vulnerability. I were able to confirm that 10 of these images were actually vulnerable. These images were from two vendors: D-LINK and Belkin.

CVE-2014-3508. This vulnerability allows context-dependent attackers to obtain sensitive information from process stack memory by reading the output of sensitive functions. I found that there were 24 firmware images which could have this vulnerability, and I were able to confirm that the vulnerabilities existed in 13 images from three vendors. These vendors included CenturyLink, D-Link and Actiontec.

This clearly demonstrated that a security evaluator, after only 3 seconds, could get a list of candidate functions to prioritize their search for vulnerable device firmware.

Scenario II. I chose the two latest commercial firmware images from D-Link DIR-810 model as our evaluation targets. I built the LSH indexes for these two firmware images and then searched those two images for all 185 vulnerabilities from Dataset IV (discussed in Section 4.2.2). It took less than 0.1 second on average to finish searching for all 154 vulnerabilities. I conducted manual verification at the top 100 candidates for each vulnerability and found 103 potential vulnerabilities in total for two images, 16 of which were confirmed (see Table 4.3). I contacted the product vendor for further confirmation.

Overall, these two case studies substantiate that GENIUS is an effective tool to facilitate IoT firmware bug searching process for security evaluators.

4.3 Discussion

While I have demonstrated the efficacy of GENIUS for accurate, scalable bug search in IoT devices, there are several relevant technical limitations. Our method utilizes static analysis to extract syntactical features, and thus cannot handle obfuscated code which is used to avoid similarity detection (e.g., malware).

Additionally, the accuracy of GENIUS heavily relies on the quality of CFG extraction. Although IDA pro [75] provides us reasonable accuracy in our evaluation, I can rely on more advanced techniques to further improve its accuracy such as [127].

Furthermore, the accuracy of GENIUS could be impacted by function inlining, since it may change the CFG structures. Since our main focus in this chapter is to improve the

scalability of existing in-depth bug search, I will leave the evaluation of GENIUS for this case as future work.

Like other CFG-based code search approaches, the accuracy of GENIUS is also affected by the size of the CFG. The smaller the size of CFG is, the more likely it is to have collisions. To be aligned with other work [53], I also considered functions with at least five basic blocks. I believe that this is a reasonable assumption since small functions have significantly lower chance to contain vulnerabilities in a real-world scenario [96].

4.4 Related Work

I have discussed closely related work throughout the chapter. In this section, I briefly survey additional related work. I focus on approaches using code similarity to search for known bugs. There are many other approaches that aim at finding unknown bugs, such as fuzzing or symbolic execution [17, 33, 35, 118, 128, 132] etc. Since they are orthogonal to our approach, I will not discuss these approaches in this section.

Source-Level Bug Search. Many works focused on finding code clones at the source code level. For example, [138] generates a code property graph from the source code and conducts a graph query to search for code clones with the same pattern. Similarly, token-based approaches such as CCFinder [84] and CP-Miner [92] utilize token sequence and scan for duplicate token sequences in other source code. DECKARD [36] generates numerical vectors based upon abstract syntax trees and conducts code similarity matching for code clone detection. ReDeBug [79] provides an efficient and scalable search to find

unpatched code clones in OS-distribution bases. All of these approaches require source code, and cannot find bugs in firmware images unless the source code is available.

Binary-Level Bug Search. Since I do not always have access to firmware source code, bug search techniques that work on binary code are very important. One common issue with the current approaches is that they only support a single architecture. It is common that bugs from firmware images in x86 can appear in images of another architecture such as MIPS or ARM, so finding bugs in firmware images demands the capability to handle binary code in a cross-architecture setting.

For example, the tracelet-based approach [41] captures execution sequences as features for code similarity checking, which can defeat the CFG changes. However, the opcode and register names are different across architectures, so it is not suitable for finding bugs in firmware images cross architectures. Myles et al. [103] uses k-grams on opcodes as a software birthmark technique. TEDEM [112] captures semantics using the expression tree of a basic block. The opcode difference on different architectures will easily defeat these two approaches. Rendezvous [86] first explored the code search in binary code. However, it has two limitations. It relies on n-gram features to improve the search accuracy. Secondly, it decomposes the whole CFG of a function into subgraphs. Our evaluation demonstrates that two CFGs as a whole by graph matching is much more accurate than comparing their subgraphs since one edge addition will introduce great difference on the number of subgraphs for two equal CFGs. Therefore, subgraph decomposition will reduce the search accuracy. Finally, as with the other approaches described thus far, it is designed for a single architecture.

Control flow graph (CFG)-based bug searching is a prevalent approach for finding bugs in firmware images. However, most existing works focus on how to improve the matching accuracy by selecting different features or matching algorithms. Flake et al. [59] proposed to match CFGs of a function to defeat some compiler optimizations such as instruction re-ordering and changes in register allocation. However, the approach relies upon exact graph matching which is too expensive to be applied for large scale bug search. Pewny et al. [113] use I/O pairs to capture semantics at the basic-block level for code similarity computation. It is still expensive for feature extraction and graph matching. DiscovRe [53] utilizes the pre-filtering to facilitate CFG based matching, but our evaluation demonstrates that the pre-filtering is unreliable and outputs tremendous false negatives. Zynamics BinDiff[51] and BinSlayer [24] use a similarity metric based on the isomorphism between control flow graphs to check similarity of two binaries. They are not designed for bug search, especially for finding bug doublets across different binaries where the CFGs of two binaries are totally different. Besides, BinHunt[64] and iBinHunt [99] utilize symbolic execution and a theorem prover to check semantic equivalence between basic blocks. These two approaches are expensive and cannot be applied for large scale firmware bug search since they need to conduct binary analysis to extract the equations and conduct the equivalence checking.

The field of automatic large-scale firmware analysis has also made a breakthrough. Costin et al. [38] carried out an analysis of over 30,000 firmware samples, but it does not perform in-depth analysis. Instead, it extracts each firmware sample and investigates it for artifacts such as private encryption keys. Therefore, this approach is not suitable for finding more general vulnerabilities without these obvious artifacts.

Dynamic analysis based bug search in firmware images. Blanket-execution [52] uses the dynamic run-time environment of the program as features to conduct the code search. This approach can defeat the CFG changes, but it is only evaluated in a single architecture. Besides, dynamic analysis to support firmware images is at the initial stage [35, 141], and still has not been demonstrated its effectiveness with respect to the run-time environments of programs for large scale firmware images.

4.5 Summary

In this chapter, inspired by the image retrieval approaches, I proposed a numeric-feature based search technique to address the scalability issues in existing in-depth IoT bug search approaches. I proposed methods to learn higher-level features from the raw features (control flow graphs), and performed search based upon the learned feature vector rather than directly performing pair-wise matching. I have implemented a bug search system (GENIUS), and compared GENIUS with the state-of-the-art bug search approaches. The extensive experimental results show that GENIUS can achieve even better accuracy than the state-of-the-art methods, and is orders of magnitude faster than most of the existing methods. To further demonstrate the scalability, GENIUS was evaluated on 8,126 devices of 420 million functions across three architectures and 26 vendors. The experiments show that GENIUS can finish a query less than 1 second on average.

5. APPLICATION II: ACCOUNTABLE BUG SEARCH IN BINARY PROGRAMS

In this chapter, I have further demonstrated the accountability of our proposed platform in the bug search scenario. I implemented a prototype, XMATCH, and systematically evaluated its performance against existing baseline methods and demonstrated its efficacy.

5.1 Experiment Evaluation

I evaluate XMATCH with respect to its accuracy, explainability and runtime performance. First, I systematically compare the performance of XMATCH against existing baseline methods under the cross-platform setting (Section 5.1.2). Second, I apply XMATCH and baseline methods to detecting real-world vulnerable code snippets (Section 5.1.3). Then, I evaluate the precision of XMATCH via matching vulnerable code with patched ones (Section 5.1.4). Further, I demonstrate the explainability of XMATCH (Section 5.1.5). In the end, I measure the runtime performance of our tool (Section 5.1.6).

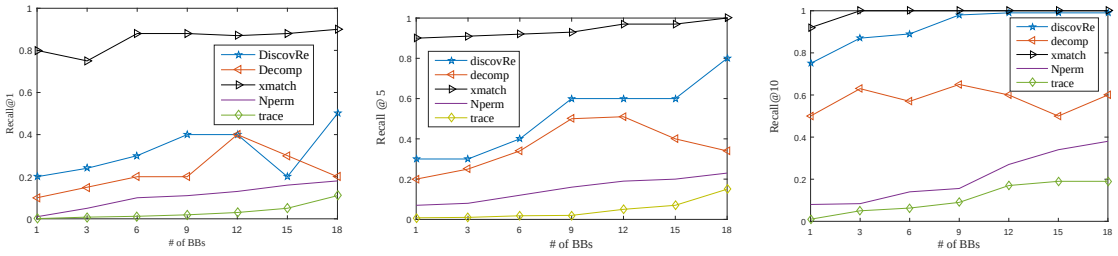
5.1.1 Experiment Setup

All experiments have been conducted on a machine with an Intel(R) Core i5 @ 2.9GHz and 16 GB DDR3-RAM, running 64-bit Ubuntu 14.04. I compiled the source code of two typical software `OpenSSL` and `BusyBox`, which are widely used in the firmware of IoT

devices, and perform code search on the generated binaries. Specifically, I have compiled two versions (1.0.1.a, 1.0.2.d) of `OpenSSL` and two revisions (1.19.0, 1.20.0) of `BusyBox`, both on two 32-bit architectures (x86 and MIPS), with three compilers (gcc v4.8.4, gcc v4.8.1 and clang v3.4), and across three major OSes (Windows, Linux and Mac OS X). Thus, I have created 72 binaries in total as the baseline dataset. All the code searching experiments were conducted on the dataset, and I kept their debug symbols because they provide the ground truth to enable us to verify the correctness of matched functions. I also selected 10 representative vulnerabilities for evaluation, including the notorious HeartBleed bug.

Table 5.1: The vulnerabilities used in our experiments.

Codebase	**Function**	**Type**
OpenSSL	EVP_DecodeUpdate	CVE-2015-0292
	X509_cmp_time	CVE-2015-1789
	dtls1 process heartbeat	CVE-2014-0160
	tls_decrypt_ticket	CVE-2014-3567
	dtls1_buffer_record	CVE-2015-0206
	X509_verify	CVE-2014-8275
	c2i_ASN1_OBJECT	CVE-2014-3508
	ssl_get_algorithm2	CVE-2013-6449
BusyBox	make_device	CVE-2013-1813
	xmalloc_optname_optval	CVE-2011-2716



(a) Recall@1 (b) Recall@5 (c) Recall@10

Fig. 5.1.: The cross-platform baseline recall comparisons under different function sizes. Recall@k means that the recall rate if we consider top k candidates as positives.

5.1.2 Cross-Platform Baseline Comparison

To demonstrate the efficacy of XMATCH in terms of cross-platform code search, we compare our system with baseline methods.

Preparation of Baseline Systems. I have prepared 4 baseline systems for the comparative experiments. They include the state-of-the-art cross-architecture bug search technique discovRe [53], a decompiler-based approach, the n-gram based permutation-resistant matching technique Nperm [85] and the tracelet-based method [41]. Notice that, although Nperm and tracelet-based methods are not designed for cross-platform code matching, their techniques can be applied to cross-architecture setting once a binary is lifted to a uniformed code representation (i.e., intermediate representation).

- *discovRe*: Due to unavailability of the source code, we have re-implemented discovRe[1] and set the iteration limitation to be the same (i.e., 10,000) as the one used in the prior work.

- *Decomp*: I have also implemented a decompiler-based bug search system `Decomp`. It relies on the on-line retargetable decompiler service [14] to conduct decompilation and leverage a prior technique [79] to compute the similarity between two recovered C functions in order to perform code search.

- *Nperm*: I employed the "*N*-perms" technique in lifted binaries. The length parameter N is set to be 3, as suggested in the previous work [86].

[1] I contacted the author of discovRe to assist us by providing their search results, but they have not responded.

- *Tracelet*: Similarly, I applied tracelet-based method to lifted binaries. Besides, to facilitate the matching process, I replaced the original optimization algorithm with a maximum value section. That is, I select the similarity score of two most similar traces as the one for the two functions.

Our baseline experiments mainly focuses on the function-level matching on the aforementioned 72 binaries. Given a function, I use each method to calculates its similarity scores against all functions in the binaries and produce a list of functions sorted in descending order of the scores. I disable function inlining during compilation since XMATCH does not currently support inclined code. Notice that this is a common limitation also shared by prior CFG-based approach [53].

Metrics. Since the accountable code search is used for refining the search results from the scalable code search engine, I would like to demonstrate how effectiveness XMATCH is to promoting ranks for true positives. Therefore, I utilize the recall rate to quantify this effectiveness. It is a standard evaluation metric, which are commonly used by many binary code search techniques for evaluation [53, 113]. It calculates the fraction of correctly matched functions in the top-k retrieved instances. Here, k means I consider top k candidates as positives. Intuitively, a larger k leads to a higher recall for every method.

Table 5.2: The vulnerability ranking baseline comparison on DDWRT firmware and BusyBox

Software	Function Name	Decomp Rank	N-perm Rank	Trace Rank	DiscovRe Rank	Xmatch
OpenSSL1.01.a x86→ DDWRT(r21676)	dtls1_process_heartbeat	50	14	843	1*	1*
	EVP_DecodeUpdate	63	40	1569	10	1
	X509_cmp_time	266	4	1662	81	1
	tls_decrypt_ticket	90	7	1913	1	1
	X509_verify	844	413	540	125	1*
	c2i_ASN1_OBJECT	1100	624	689	3	1
	dtls1_buffer_record	574	7	1108	12	1
	ssl_get_algorithm2	59	230	1009	6	1
BusyBox1.19.0 x86 → BusyBox1.20.0 MIPS	make_device	180	69	4	6	1
	xmalloc_optname_optval	87	93	428	10	1

* means that there are multiple functions with the same similarity scores.

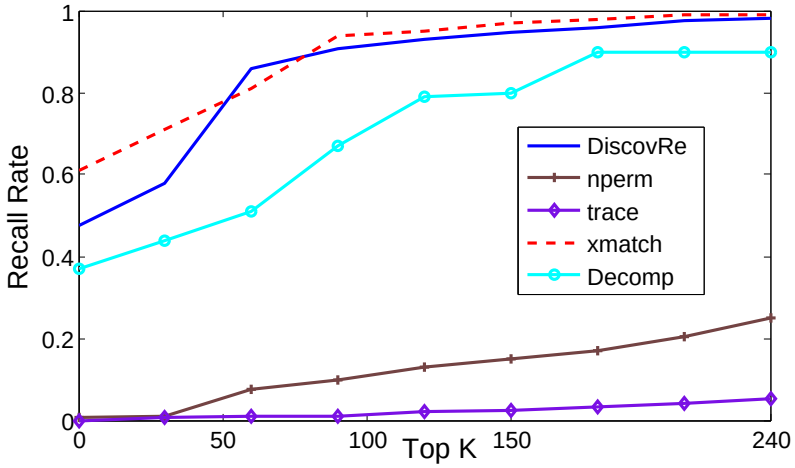


Fig. 5.2.: The cross-platform baseline comparison on 1,000 functions randomly selected from the dataset.

Comparison Results. I conducted two sets of experiments for baseline comparison. Firstly, I investigated the performance of proposed and baseline methods when handling different sizes of functions. Secondly, I systematically compared XMATCH with baseline methods on a real-world library `OpenSSL`.

For the first experiment, I clustered functions of different names in our dataset by the number of basic blocks. I randomly selected 500 functions for each cluster i. For each function, I collect its x86 and MIPS versions compiled by gcc v4.8.4. As a result, I have a new data set C=$\{c_1, \ldots, c_i\}$ where c_i contains 1000 functions with same size i. For each function in cluster c_i, I search the MIPS version using its x86 version in c_i. I use Recall@K to measure its accuracy. I ran XMATCH and baseline methods in different clusters c_i and obtained the evaluation results for Recall@1, Recall@5 and Recall@10.

Figure 5.1 shows their matching accuracy for functions from size 1 to 18. In general, it demonstrates the accuracy of XMATCH is better than the state-of-the-art approaches. It is worth noting that the performance is evaluated 7,000 functions from the same benchmark,

and the advantage of XMATCH over all other methods is statistically significant, according to a paired t-test (at the level $p = 0.05$).

Particularly, XMATCH outperforms all of the baseline methods for small functions (size$\leq$12). Figure 5.1 indeed justifies our argument: although discovRe is accurate for matching large functions whose basic block number is greater than 15, this technique is not favorable in searching small functions. This is because small functions have fewer constraints for CFG-based approaches to utilize. In contrast, XMATCH can achieve better accuracy due to the conditional formula based matching. Even small functions still have semantic-rich and thus unique conditional formulas. For instance, function `X509_verify` has only one basic block and therefore DiscovRe cannot rank it at recall@100. On the contrary, XMATCH can give it a top ranking because the enclosed conditional formulas are relatively unique.

In the second experiment, I randomly selected 1,000 functions without considering their sizes, and applied all mentioned approaches on this dataset. For each method, I also collected its recall rates for different threshold K. Figure. 5.2 shows the comparison results. We also can see that XMATCH can still outperform all baseline methods.

I also notice that the decompiler-based approach does not provide meaningful results in two sets of experiments. Most false positives are caused by the decompilation errors. This justify the motivation of our approach.

I investigated false positives of XMATCH and found that most of them are caused due to two reasons. First, different functions share the similar code logics. I found that different functions may have similar code patterns except that they access different fields of the same object. This is common in the functions with small amount (e.g., only one) of

basic blocks. I can handle such cases by further introducing context information, such as its callers and callees. Second, different functions share the same constants. Sometimes, identical constants such as address of global variables or object offsets are encoded into conditional formulas that are originated from binaries compiled with even different architectures. Such noises may dominate the similarity computation and lead to false matching. Nevertheless, XMATCH has already reduced false positive rate significantly, compared to baseline methods, due to its fundamental advantage of semantic and contextual awareness. Furthermore, the self-explanatory property of conditional formulas can facilitate further manual verification and help human experts easily screen these false positives.

Table 5.3: Cross-platform patch code matching.(unpatched:OpenSSL 1.0.1a vs. patched:1.0.2d, x86 vs MIPS)

Vulnerable Function	**P(x86) →U(MIPS)**	**P(x86)→P(MIPS)**	**U(x86)→U(MIPS)**	**U(x86)→P(MIPS)**
dtls1_process_heartbeat	1/0.81	1/0.97	1/0.96	1/0.83
EVP_Decode_Update	1/0.90	1/0.99	1/0.99	1/0.88
X509_cmp_time	1/0.74	1/0.95	1/0.95	1/0.74
tls_decrypt_ticket	1/0.90	1/0.99	1/0.99	1/0.90
c2i_ASN1_OBJECT	1/0.66	1/0.99	1/0.99	1/0.66

P means patched version, and U means unpatched version. For each matching result `x/y`, `x` means ranking and `y` is similarity score.

5.1.3 Searching Vulnerable Functions in Real-World Software

To understand the effectiveness of our technique, I apply both XMATCH and the baseline methods to real-world vulnerable binaries. Specifically, I focus on 10 representative vulnerabilities in `OpenSSL` and `BusyBox`, each of which is corresponding to an individual function as presented in Table 5.1.

Aiming at finding bugs in real-world `OpenSSL` libraries, I perform binary code matching on Linux-based DD-WRT router firmware (r21676) [12]. To enable cross-platform search, I utilize known vulnerable functions in `OpenSSL1.0.1a` binary (compiled under x86 using gcc 4.8.1) as the bug signatures to discover the same bugs in `OpenSSL` binary used directly in this DD-WRT firmware. Similarly, to uncover cross-platform vulnerabilities in `BusyBox` binary, I examine its MIPS distribution using bug signatures generated from x86 binary code. More concretely, I first compile an x86 version of `BusyBox1.19.0` using gcc 4.8.1; I create a MIPS version of `BusyBox1.20.0` using gcc 4.8.4. Then, I attempt to match the vulnerable functions from the former one with the unknown functions in the latter.

Table 5.2 illustrates the comparison results. XMATCH can always correctly discover the vulnerable functions as top candidates in the target binary. On the contrary, none of the baseline methods, including the state-of-the-art technique DiscovRe, can produce a high ranking for most of the buggy functions. For example, XMATCH can rank `X509_cmp_time` at top 1; on the contrary, discovRe can only rank it at top 81.

In the case of function `X509_verify`, aside from the true vulnerable function, XMATCH also identifies three other functions (e.g., `X509_REQ_verify`) to be top candidates be-

cause these functions all bear the same conditional formulas. By investigating these false positives in source code, I find that these "same" conditional formulas in fact access different types of data objects. However, due to the lack of high-level type information, XMATCH cannot distinguish such formulas from one to another. However, it is worth noting that these three functions may potentially be vulnerable since the presence of same conditional formulas could indicate the identical buggy program logics, which are left unpatched. I have contacted OpenSSL team to request further confirmation.

5.1.4 Unpatched versus Patched Code

One major challenge of bug search lies in that a patched version may have some differences with the original vulnerable function. Such difference may confuse XMATCH to get some false negatives. Thus, I would like to evaluate XMATCH with both buggy code and the corresponding patch in order to understand whether XMATCH can find the patched version. Furthermore, I hope to perform such a measurement in a cross-platform setting.

To this end, I first compiled a vulnerable `OpenSSL` (1.0.1a), with 5 representative bugs, using gcc 4.8.1 under both x86 and MIPS, and compiled a patched one (1.0.2d) using Clang 3.4 also under the same architectures. Then, I matched generated x86 binaries to MIPS ones. This involves four classes of matching: 1) patched-to-unpatched, 2) patched-to-patched, 3) unpatched-to-unpatched, and 4) unpatched-to-patched. Table 5.3 shows the results on 5 representative vulnerabilities. For each function, the matching result includes a candidate ranking and a similarity score.

Patched-to-Patched and Unpatched-to-Unpatched As a baseline, I first evaluated the matching between two patched or two unpatched binaries on different architectures. As depicted in Table 5.3, XMATCH can produce fairly high similarity scores (around 0.98 on average) between matched functions in these two classes. This again demonstrates that our conditional formulas can distill the essential program logics (vulnerable or not) while abstracting away the low-level architecture-specific details. However, I did notice that we cannot always exactly match the conditional formulas extracted from binaries in two different architectures and therefore cannot achieve a 1.00 similarity score. In a further investigation, I found that this imperfection is caused by the existence of global variables: indexes for global pointers used in conditional formulas may be different across architectures. I will address this issue in the further work by proposing a better way to index global pointers.

I found that even if the similarity score produced by XMATCH cannot differentiate the patched from unpatched code. Since the conditional formula has the explanatory property, the security expert can use it to facilitate the manual verification. I will discussed it in the latter section.

Patched-to-Unpatched and Unpatched-to-Patched For the function matching in these two classes, I noticed that a patched function is still considered as the top candidate of an unpatched one and vice versa. For instance, the patch for `tls_decrypt_ticket` consists of merely 3 lines of code and the one for c2i_ASN1_OBJECT, which is already fairly large, contains only 12 lines of code. In such cases, the rest of conditional formulas, which are left unchanged, may dominate the similarity computation.

5.1.5 The Case Study On Explainability

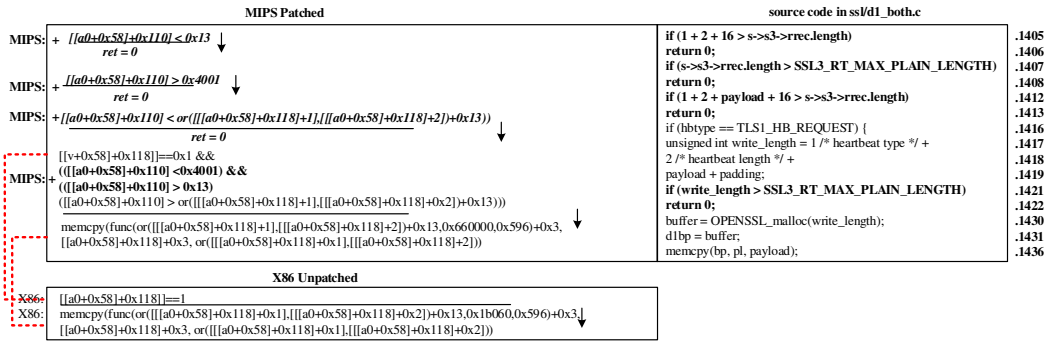


Fig. 5.3.: The explainability demo for `dtls1_process_heartbeat` vulnerability

In this section, I demonstrate that the conditional formulas is able to provide self-explanatory evidence to human experts for further verification. In the paper, I will analyze one example as a demonstration. **More examples can be found in the provided anonymous supplementary materials.**

To exemplify the intrinsic explainability of conditional formulas, I study the matching results from an unpatched x86 function to a patched MIPS version. Due to the page limit, I only demonstrate our analysis on the function `dtls1_process_heartbeat` whose unpatched version bears the Heartbleed bug (i.e., CVE-2014-0160). The analyses on other functions are presented in Appendix.

CVE-2014-0160 The result for the `dtls1_process_heartbeat` is shown in Figure. 5.3. The code snippet on the left represents conditional formulas extracted from patched MIPS and unpatched x86 binaries, respectively. The one on the right presents corresponding source code. The matched formulas are linked by red dotted lines.

The root cause of Heartbleed vulnerability is the missing length check for `memcpy()` arguments and such a check is introduced in the patch code. In contrast, the dataflow

that reaches `memcpy()` remains intact. Both the modified condition and invariant dataflow can be directly observed from the two matched conditional formulas. Specifically, the If-clauses are different due to the introduction of new boundary check (depicted in bold), while the Then-clauses denoting `memcpy()` activities remain unchanged.

In this case, the identical and sophisticated Then-clauses indicate that the two functions are indeed very similar to one another. This explains why XMATCH considers the patched MIPS function to be the top matching candidate for a vulnerable one. Nevertheless, since two functions bear different behaviors in terms of condition check, their similarity score is relatively low (0.81).

In addition, due to the behavior level matching, XMATCH can explain that these two functions, buggy and patched, are corresponded to one another exactly because they share these similar conditional formulas. Thus, by assessing the difference between two *CF*s, which includes barely 2 predicates in the If-clauses, a human analyst can easily understand and rule out such a false alarm. In contrast, prior work can only output the similarity score and matched control flow graphs without pinpointing the exact matching regions. In that case, human experts will have no choice but to manually analyze the binary code to dig out the vulnerable logic for further verification.

5.1.6 Runtime Performance

I tested XMATCH on about 1000 randomly selected functions from the dataset in Section 5.1 and evaluated the runtime in three steps, i.e. binary lifting, conditional formula

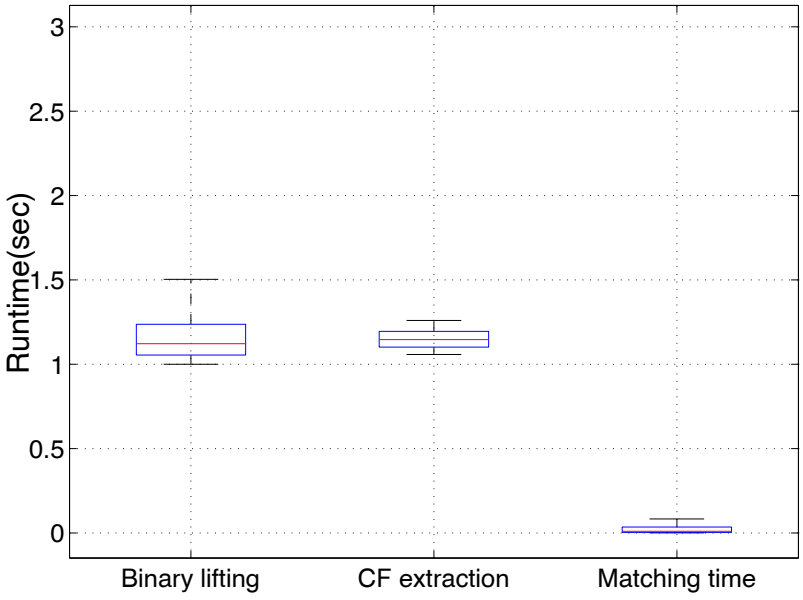


Fig. 5.4.: The runtime performance of XMATCH

extraction, and function matching. The first two steps are offline steps that can be preprocessed beforehand; whereas the last step is an online search step.

The results are presented in Figure 5.4. On average, the binary lifting and conditional formula extraction take 2.3 seconds, and it takes 0.029 seconds to perform the function level matching. Overall, no matching takes longer than 0.55 seconds for all functions. The maximum preprocessing time (binary lifting and conditional formula extraction) is about 2.9 seconds. The preprocessing can be easily executed in parallel across multiple machines, and thus is not the bottleneck of our system. The search time grows linearly with the number of searched functions, and thus the function search is reasonably fast. I plan to further improve the performance of XMATCH by using the indexing techniques in our future work.

5.2 Discussion

In this section, I mainly discuss about the limitation and potential challenges of this work.

Loop Handling I only unroll the loop once during the data flow analysis. This is a safe strategy, because it does not increase the false negatives of the match result. Besides, this strategy is also attempted in other works [25], In the future, I could apply the existing loop analysis such as the technique[76] to further improve the accuracy of our approach.

Vulnerability across multiple functions. The goal of XMATCH is not to create the abstract formula cross multiple functions to find the potential vulnerabilities. If the vulnerability is related to multiple functions. XMATCH will find all related functions for the further vulnerability diagnosis.

Function Inlining. XMATCH can handle the inlined function which does not affect the most of code logics in the caller function. If the inlined function changes most of code logics in the caller function, I need to extend XMATCH to support inter-procedure analysis to not only generate the conditional formulas for one function but a set of functions. I will study how to systematically address this problem in future work.

5.3 Related Work

I have discussed closely related work throughout the chapter. In this section, I briefly survey additional related work.

Feature Representations in Code Search. Current code search techniques can be broadly divided into source code-level and the binary-level searches. At the source code-

level, search techniques have an avanced understanding about the high-level semantics of the vulnerability. The work [138] models the vulnerable code as a definition graph for vulnerability pattern generation. This definition graph captures the definition of function arguments and their sanitization checks. This work shares the similar idea about how to characterize the pattern of vulnerable code. However, it can only be applied at the source code-level. This chapter targets the identification of bugs in compiled binaries.

Compared with source code-level code searching, binary-level code searching is far more challenging. Most of these works do not focus on matching functions by their code logics. Instead, they focus on matching syntactics features [77, 86], semantic features [112], I/O pairs of code semantics [113], or code environments [52]. All these approaches only give the similarity score. Therefore, they cannot provide a reasoning scheme which can also give effective evidence about why the target code is vulnerable. Tracelet-based approach [41] gives the accountable matching result, which shares the similar idea with ours. However, it cannot be applied on the lifted binaries. I have demonstrated this point in Section A.5.

Unknown Vulnerability Discovery. The main problem our paper addresses is to find known bugs in new binaries. It has not been designed to find unknown bugs. There are many works on unknown vulnerability discovery. Fuzzing is a common technique for doing this. Carefully selected fuzzing seeds will effectively trigger unknown vulnerabilities [33, 118]. Symbolic execution is another technique for vulnerability discovery. The symbolic execution of a program explores all possible execution paths throughout that program and determines whether any combination of inputs could cause the program to crash [29, 117].

Binary Analysis related approach. In this chapter, I do not invent new binary analysis techniques. Instead, I leverage the existing binary analysis techniques to extract conditional formulas for code search. Therefore, our proposed approach can be applied into more mature platforms such as BAP [26], Bitblazer [131], or Panda [49]. Besides, lifting binaries into the intermediate representation has been well studied. I choose LLVM-IR, because LLVM framework is mature and has many excellent optimization features. The binary lifting of the paper can also be implemented by other types of IR such as Valgrind VEX [104], BAP BIL [26], or REIL [50].

Decompilation Related Approach. Decompilation can provide more readable code, but that is not explainable. This is because it does not conduct the factorization on the function to extract independent code logics. An analysis still needs to manually check the text to locate the potentially vulnerable code logic. Program analysis on generated C code could facilitate this process, but the quality of decompiled C code cannot be guaranteed, due to the limitations of decompilation [124]. I also substantiate this point in Section A.5. Instead of conducting the source code analysis on decompiled C code, XMATCH targets on the lifted binary code which is more accurate than decompiled C code. Furthermore, it is more explainable, since it can locate potentially vulnerable code logics in the function. This is substantiated in Section A.5.

5.4 Summary

In this chapter, I extracted the novel feature representation *conditional formulas* to conduct the cross-architecture code search. The conditional formula explicitly captured two

cardinal factors of a bug: 1) erroneous data dependencies and 2) missing or invalid condition checks. To better facilitate human bug verification, I formulated the matching of conditional formulas as a linear assignment problem and leverage integer programming techniques to correlate the statements in two binary programs in an optimal fashion. I had implemented a prototype, XMATCH, and evaluated it using the well-known software `OpenSSL` and `BusyBox`. Experimental results had shown that XMATCH outperforms existing bug search techniques. At the same time, it also provided evidence of detected vulnerabilities, which can then be easily examined via human inspection.

6. APPLICATION III: ACROSS-VERSION MEMORY ANALYSIS

6.1 Introduction

Memory analysis aims at extracting security-critical information from a memory snapshot of a running system or a program. It has many security applications, such as virtual machine introspection [65], malware detection and analysis [83], game hacking [28], digital forensics [2, 55], etc. Most of these applications require retrieving desired information from a memory snapshot of a running software or system, so I refer to them as memory analysis tools in general.

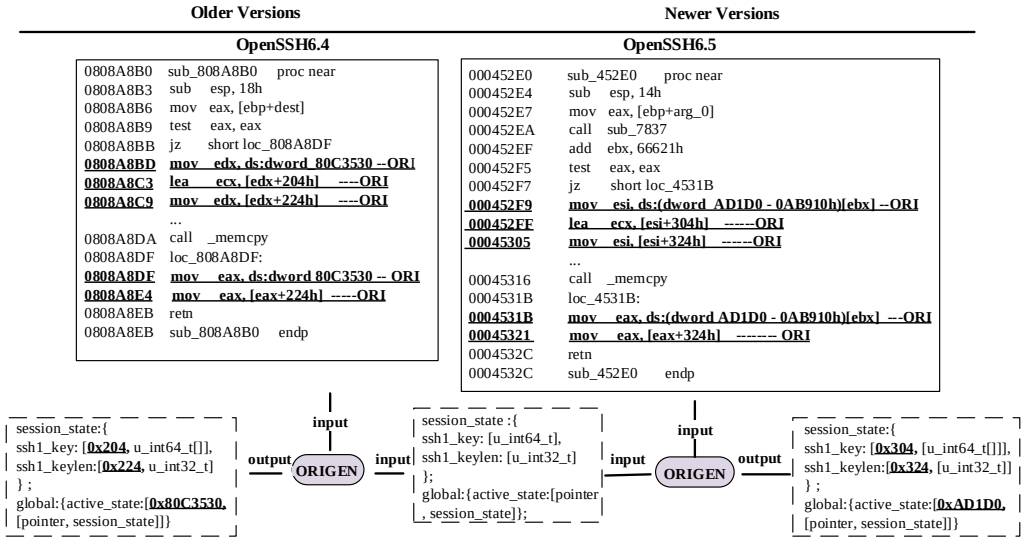


Fig. 6.1.: **The OpenSSH example.** It shows code snippets to retrieve the session key for openssh in two versions. Offset-Revealing Instructions (ORIs) are highlighted in both versions. Given the abstract profile, the profile localization determines the offsets from the identified ORIs and produces a localized profile for each version.

For all these memory analysis applications, I need to have the precise knowledge about data structures that are relevant to the specific analysis purpose. Most of existing memory analysis tools usually build a data structure profile, i.e. a mapping between data structures to their offsets in the target binary, to derive analysis decisions. The data structure profile is constructed to incorporate precise knowledge about data structures. For instance, I may build a precise data structure profile about the offset values of important fields, such as the process name, process ID, and the pointer to the next `EPROCESS` structure, in the `EPROCESS` data structure in order to retrieve running processes from a memory snapshot for Windows OS.

The creation and maintenance of the data structure profile is a nontrivial problem, especially for COTS binaries. It requires the expert knowledge about the internal working of the target software. Existing work, such as Volatility [2], VMST [61] and Virtuoso [48], have made a big progress on automatic introspection code generation. Their techniques work will when the target software is open-source [2, 48], or when the well-defined code pieces are provided, which can be reused for introspection [61].

For COTS software, however, existing memory analysis tools still rely on cumbersome reverse engineering techniques to build the profile. In most cases, the profile generation still depends on the manual effort. Unfortunately, the daunting profile creation task is not a one-time effort. It is tightly coupled to the specific version of the software being analyzed, and needs to be constantly rebuilt for new versions of the software. As a result, the effort spent on building the analysis profile for one particular version of a program could not be applicable to its future versions. For example, a memory analysis tool, such as Volatility [2], has to create a profile for every version of a COTS software to be analyzed.

Once the version is changed, the profile has to be manually updated for the exact same software so that the analysis can proceed correctly.

In this chapter, I propose a novel notion of "cross-version memory analysis". That is, the data structure profile used in one version can be adapted to other versions of the same software without manual efforts. With the *cross-version memory analysis* property, I can automatically build profiles for new versions of a software by transferring the knowledge from the profile that has already been trained for its old version. Our intuition is that adjacent versions of the same software tend to be similar. The experimental results in Section 6.6.2 substantiate this claim. Based on this idea, I can transfer the relevant knowledge from an already trained profile to build the profile for an unseen new version. The less different a new version is from the previous version, the more accurately the profile can proceed the analysis .

To achieve the cross-version memory analysis, I combine program analysis and code searching techniques to automatically transfer the data structure profile across different versions of a software. I observed that some instructions, at the binary level, reveal the actual offsets (as constant values) for the specified data structure fields and global variables, as these offsets have been statically determined at compile time. I name these instructions "offset-revealing instructions" (in short, ORI). Given a trained profile on one version, I label ORIs in the binary of this version by program analysis techniques. With the knowledge of learned ORIs in this version, I can identify semantically-equivalent ORIs in new versions by the code searching technique, and localize the introspection profile by updating offset values for correspondent data structure fields based on identified ORIs.

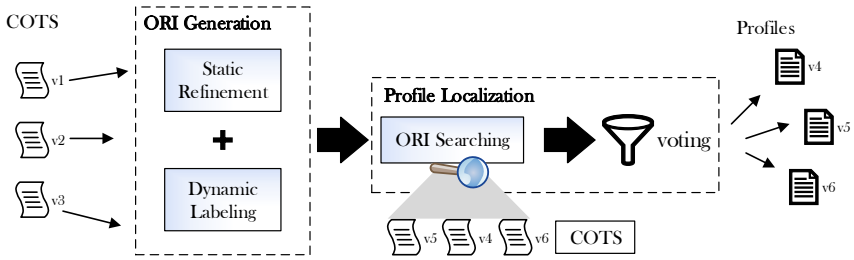


Fig. 6.2.: The overview of ORIGEN

I have developed a prototype system called ORIGEN and evaluated its capability on a number of software families including Windows OS kernel, Linux OS kernel, and `OpenSSH`. Particularly, I systematically evaluate it on 40 versions of `OpenSSH`, released between 2002 and 2015. The experimental results show that ORIGEN can achieve a precision of about 90% by transferring relevant knowledge in the profile of a different version automatically. The results suggest that ORIGEN advances the existing memory analysis methods by reducing the manual efforts while maintaining the reasonable accuracy. I further have developed two plugins to the Volatility memory forensic framework [2] and integrated them in ORIGEN, one for OpenSSH session key extraction, and the other for encrypted file system key extraction. I show that each of the two plugins can construct a localized profile and then can perform specified memory forensic tasks on the same memory dump, without the need of manual effort in creating the corresponding profile.

Certainly, I admit that ORIGEN may not work when our assumption does not hold, i.e. when a software version is significantly different from the base version on which the ORI signatures are generated. For these cases, I can generate a new profile to cover its ORI signatures and apply to many other similar versions. Nevertheless, ORIGEN introduces

a promising solution for cross-version memory analysis and demonstrates an empirically validated approach to greatly reducing the manual effort for profile creation. The research along this direction is important because it could streamline the memory analysis process, with minimal manual intervention required.

In summary, the contribution of this paper is threefold:

- I propose a novel notion of cross-version memory analysis. I made the first attempt to conduct the memory analysis across different versions of the software. Our study demonstrates that the across-version memory analysis can be achieved with a minimal or reduced human intervention.

- I developed a prototype system ORIGEN, which combines the program analysis and code search technique to address the new problem domain.

- I systematically evaluated the accuracy of ORIGEN under 40 versions of the `OpenSSH` family, and the evaluation results show that ORIGEN can achieve a precision of more than 90%. The case studies also demonstrate ORIGEN can successfully recover the offsets for key semantic fields across different versions of `OpenSSH`, Windows, Linux, a loadable kernel module for Linux.

6.2 Overview

I utilize a running example in Figure A.2 to demonstrate our problem. Although I target at the memory analysis for the COTS software, for clarity, I utilize the open-source software `OpenSSH` to demonstrate our basic idea. Figure A.2 shows code snippets for two versions of OpenSSH (6.4 and 6.5), where several highlighted instructions are used to

access `ssh1_key` and `ssh1_keylen` fields in the structure of `session_state`, and a global variable `active_state`, which points to the structure `session_state`. The constant values carried by these instructions indicate the exact offsets of these fields inside the data structure. Therefore, these highlighted instructions are ORIs.

In this case, there are three symbols shared by OpenSSH (6.4 and 6.5). I utilize the abstract profile to denote these common symbols. Given this abstract profile, I develop an SSH key extraction tool that can locate encryption keys for active SSH sessions in a memory snapshot in the cross-version manner. ORIGEN will automatically identify ORIs in `OpenSSH6.4`, and transfer the profile for `OpenSSH6.4` to a localized profile for `OpenSSH6.5` based on identified ORIs in the older version. Using this localized profile, the SSH key extraction can immediately work for `OpenSSH6.5`, without any code modification. This demonstrates the nature of cross-version memory analysis for ORIGEN.

Problem Statement In this chapter, I aim to achieve the cross-version memory analysis. That is, we can automatically generate profiles for new versions of a software by transferring the knowledge from the model that has already been trained on its old version. Given an abstract profile that a memory analysis tool relies on and a base version of target software, ORIGEN locates ORIs in the base version and searches these ORIs in the target version. With newly identified ORIs in the target version, we can localize the profile for the new version.

More specifically, when provided a different version of the same software, I aim at achieving the following goals: 1) identify instructions that are semantically equivalent to the ORIs identified from the base version; 2) extract the offsets from these instructions; 3)

```
0x80037324: mov eax, [edx+0Ch]          R offset 0x170 base: 0x80090a08 type: session_state
0x800370a4: mov dword ptr [eax+8], 0 W offset 0x15c base: 0x80090a08 type: session_state
0x80046659: mov edx, [eax+21Ch]         R offset 0x21c base: 0x80090a08 type: session_state
0x80037324: mov eax, [edx+0Ch]          R offset 0x160 base: 0x80090a08 type: session_state
0x80045624: mov ecx, [esi+214h]         R offset 0x214 base: 0x80090a08 type: session_state
```

Fig. 6.3.: The demo of the session_state object tracing log.

generate a localized profile for the new software version. In summary the challenge is to find ORIs in the target program of a given base version.

If we have the source code for the program to be analyzed, a straightforward way would be to use the compiler tool-chain to output such information directly while the compiler generates the binary code. In many cases, the source code is often not available (e.g., VMI for Microsoft Windows). Therefore, we need to develop a binary analysis technique to extract this information from binary code.

System Overview Figure A.1 illustrates an overview of our solution. It involves the ORI labeling and the profile localization.

In general, ORI labeling takes a base binary as the input, and performs dynamic and static analysis to finally output all labeled ORIs in the base binary. Profile localization searches a target binary for the instructions that are semantically equivalent to labeled ORIs in the base binary, and localize the profile for the target binary. The details will be discussed in latter sections.

6.3 ORI Signature Generation

6.3.1 ORI Signature Definition

An ORI is an instruction that has a constant field that reveals the offset of a field in the data structure definition, or the location of a global variable within the data section. The definition is as follows:

Definition 6.3.1 *Offset Revealing Instruction (ORI) is a tuple of* (p, c, t, f)*, where* p *is the program counter,* c *is the constant field within the instruction,* t *indicates the data structure type, and* f *denotes the field name within the data structure definition. For a global variable,* t *is "data section", and* f *is the name of the global variable.*

6.3.2 ORI Labeling

In this section, I describe how to label ORIs in a binary and generate signatures for the labeled ORIs. It can be considered as a learning stage. At this stage, I attempt to learn ORI signatures which will be used for latter version-independent memory analysis.

ORIs for Global Variables It is straightforward to identify ORIs for global variables. Once the exact location is determined for a global variable in the base version, I can simply scan the binary code to identify all the instructions that refer to this location. The location for a global variable often has a distinct value, because it is located in the data section of the binary module. For the running example, I can see that `active_state` is a global

variable and I can find its address 0x80C3530 from the debug symbol. Through scanning in the binary, I can label the 0x808ABD as an ORI directly.

For the rest of this section, I focus on ORI identification. The offsets of data structure fields are often very small, and small constant numbers are pervasive in binary code. Therefore, I use a different solution. I first dynamically trace the binary program and identify a set of instructions that access the specified data structure fields (which is described in "Dynamic Labeling"). I call these instructions "ORI candidates". Based on ORI candidates, I perform static analysis to filter out false ORIs and discover more ORIs, which is described in " Static ORI Discovery".

Dynamic ORI Labeling The goal of dynamic labeling is to collect a set of instructions that either read or write the given data structure field defined in the abstract profile. To do so, I need to know not only when an instance of the data structure is created and later destroyed, but the lifetime of data structure instance during the program execution. With the aid of the information about live data structure instances, I can pinpoint the instructions that access their specific fields during tracing the program execution.

To this end, I should have certain knowledge about data structures in the base version of the software. There are three types of information I need to know about the data structures in the base version: 1) The functions which create and delete the data structure instances of interest; 2) Data structure definitions that are relevant to the analysis task; 3) Actual offset for each data structure field of interest.

I hook functions which create and delete the data structure instances of interest during the binary execution to label the live data structure instances in the memory. I can further

identify all instructions which have `write` or `read` operation on these live data structure instances by monitoring all the memory read and write operations during the execution. The data structure definition and its field offset information can help to extract ORIs in these identified instructions. For the programs with source code, such knowledge can be easily obtained. Even for the many binary programs (e.g., Windows), I can still obtain the knowledge from documentation of APIs. For the binary programs with limited documentation, I have to rely on reverse engineering to retrieve the needed knowledge. This is a reasonable assumption, because without this knowledge, memory analysis is not even possible in the first place.

As for our running example shown in Figure A.2, I have to know the definition of `session_state` and a global variable `active_state` pointing to this structure in `OpenSSH6.4`. Moreover, for the data structure fields of interest, I need to know their actual offsets within the data structure `session_state`. I hook `alloc_session_state()` function to keep track of the creation of `session_state`. As OpenSSH sever never frees the `session_state` instance, I do not hook any other functions.

When tracing the program execution, I may face several situations: (1) if an instruction does not access the field of interest at all, I simply drop it; (2) if an instruction accesses multiple data structure fields at different times, I also drop it due to its ambiguity; (3) if an instruction is observed to only access a single field of interest and the constant value carried in it matches with the field's actual offset, I treat this instruction to be an ORI; and (4) if an instruction is observed to only access a single field of interest but it does not carry a constant or the constant value does not match with the field's actual offset, I keep it as an

ORI source. Although this instruction is not a real ORI by definition, it may lead us to find a real ORI through the following static analysis.

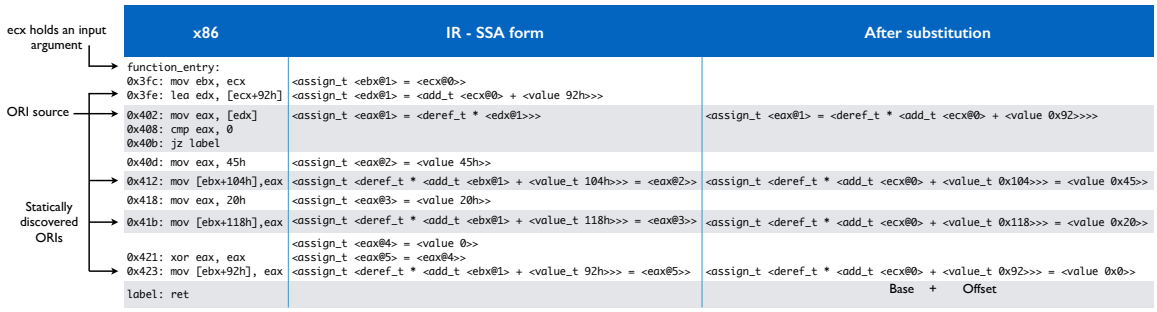


Fig. 6.4.: Static discovery of ORIs.

Static ORI Discovery Based on the ORIs and ORI sources labeled through dynamic analysis, I further perform static analysis to discover more ORIs which are missed by dynamic analysis.

Starting from an identified memory access instruction (either ORI or ORI source), I perform the backward data-flow analysis to know how the memory operand is computed. More specifically, I perform backward data-flow analysis on the memory operand in that instruction, and look for a variable that holds the base address and a constant value that holds the offset. For example, in Figure 6.4, the memory-access instruction at 0x402, which is the source for the ORI at 0x3fe, is first identified via dynamic analysis. ORI, by definition is an instruction of the form 'base + offset' where offset is equal to the offset within the object that the access corresponds to. I first perform backward data-flow analysis from the ORI-source to reach the ORI, then, I extend the analysis to identify the source of the base register. With the base register identified, flow-insensitive forward-data-flow analysis on the base register reveals all the ORIs present in the function. That is, in Figure 6.4, an ORI

source at 0x402 is first identified via dynamic analysis. Then, the corresponding ORI is identified at 0x3fe. The register containing the base address is identified as `ecx@0`.

From the variable that holds the base address, I perform the forward analysis in the same function to discover more ORIs. If I observe a constant value being added to the base, and that value matches with one of our data structure fields in the profile, whichever instruction carries this constant is a new ORI. In Figure 6.4, I start from `ecx@0` and perform forward data-flow analysis and discover ORIs at 0x412, 0x41b and 0x423.

For the data-flow analysis, the x86 code is converted into an IR-SSA form (column 2 in Figure 6.4) and the *use-def* and *def-use* chains are directly derived from them [101]. Then, the definitions are recursively propagated by substituting them into the uses until each of the statements is composed of only the entry point definitions (e.g., ecx@0 in Figure 6.4). Column 3 in Figure 6.4 presents the IR statements after substitution. In the end, I identify a statement to be an ORI if and only if (1) The expression contains a 'base + offset'[1] form where base is equal to the previously identified source of the base register (e.g., `ecx@0` in the example) and (2) The offset equals to a valid offset value within the profile.

6.4 Profile Localization

For each symbol defined in the abstract profile, I have one or (often) multiple ORIs for the base version of a binary. To localize the profile for a new version of the binary, I try to find instructions in the new binary that match with these ORI signatures and update the profile based on the abstract profile and identified ORIs in the new binary.

[1]Offset of 0 is a special case where the memory access appears like a regular dereference.

6.4.1 ORI Identification

I consider matching ORI signatures in a new binary as a code search problem, and leverage the existing code search technique to conduct the profile localization.

To precisely label ORIs in a new binary, I need to conduct a CFG-based code search approach. The assumption is that two versions of a binary share the similar control flow graphs. This has been substantiated by existing literatures [51, 99, 113], and many other works also apply this assumption into many applications [89]. The CFG-based code search considers a instruction with the similar position in the control flow structure as a match. Even if the ORI in the new binary has a different representation, the CFG-based code search can still find it, if two versions of the binary share similar control flow graphs.

The CFG-based search includes the control flow graph extraction and graph matching. I leverage the existing tool BinDiff [51] to achieve the CFG-based code search. It has two advantages. Firstly, its control flow graph matching and instruction alignment perfectly suit our usage scenario. Secondly, it is a mature tool with good runtime performance. Therefore, I utilize Bindiff to match the base version of a binary with the new version.

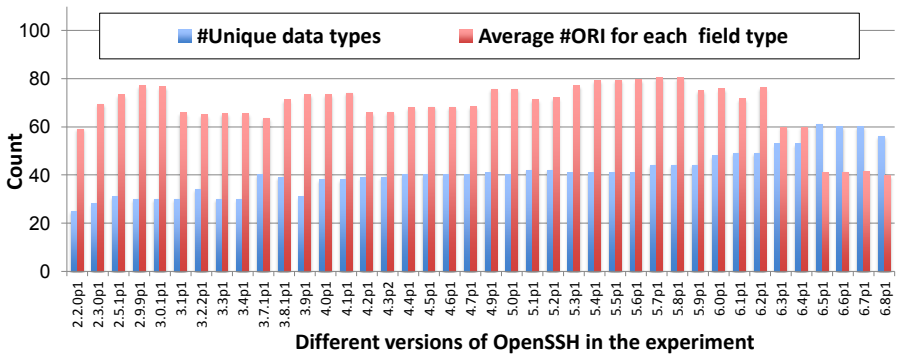


Fig. 6.5.: **The statistics of the data types and the average number of ORIs to the field type in the OpenSSH dataset.**

6.4.2 Profile Generation

The output of Bindiff is a one-to-one mapping between instructions of two binaries. I can generate the profile for the new binary, according to the abstract profile and the mapping. The profile generation is to walk through each symbol in the abstract profile and update the field offset information based its correspondent ORIs.

To this end, ORIGEN locates ORIs in the new binary based on the mapping, identifies all ORIs for each symbol, and updates offset information based on these identified ORIs. ORIGEN can locate the semantically equivalent ORI instructions in the new binary by looking up the instruction mapping. It considers instructions mapped by ORIs of the base version as qualified ORIs. ORIGEN clusters all identified ORIs by their symbol names, and update the offset information for each symbol based on its ORI cluster.

By the ORI definition in Section 6.3.1, I know each symbol involves the object type and field name. Each ORI cluster have one or more ORIs. If there is only one ORI in the cluster, I can directly extract its offset information from the ORI and assign it to the symbol. In most cases, the ORI cluster contains multiple ORIs. I adopt the voting mechanism to update the offset of a symbol. This is because that the CFG-based code search could introduce the erroneousness, and this could wrongly consider some instructions as ORIs. Without false ORIs, the ORIs for the same symbol share the same offset value. False ORIs will break this consistency and generate different offset values to confuse ORIGEN. The voting mechanism is designed to automatically filter offset values from false ORIs and improve the accuracy of the profile generation.

Considering each offset value as a vote from its ORI, the voting mechanism will rank all offset values by the number of votes, and select the offset with the largest number of votes as the true offset for the symbol. Repeat this process, the profile generation will assign each symbol with an offset value and generate the profile for the new binary.

6.4.3 Error Correction

It is possible that ORIGEN fails to update the offset value for a symbol in the new binary, if all of ORIs of some symbol in the abstract profile are misidentified in the new binary. I adopt two strategies to resolve this problem.

The first strategy is the conservative strategy. I can filter out symbols with the high possibility to be wrongly labeled in the generated profile. Each symbol has a cluster. I use the variance from the set of offset values in the cluster to determine its false possibility. A threshold is set to determine whether the symbol is filtered or not. If the variance of the symbol value is above the threshold, I consider this symbol as a false and filter it out.

The second strategy is that I do not discard any symbol in the profile. Instead, I apply the profile to conduct the memory analysis. During the memory scanning, I collect the values from these symbols, and screen false ones by heuristics. Once I found some abnormal values, I filter the symbol from the profile.

I also can combine two strategies together to conduce the error correction. In all, the error correction can greatly reduce the false positive rate for the generated profile. This is substantiated by the experiment in Section 6.8.

6.5 Implementation

I have implemented the prototype of ORIGEN in C and Python. More specifically, I write the dynamic labeling plugin for DECAF [72] in C. As a whole-system dynamic analysis platform, I use DECAF to trace a user-level program, an entire OS kernel, or a specific kernel module. Besides, I write an IDA Pro plugin for static binary analysis, based on IDA-decompiler [74]. I leverage BinDiff for the ORI search. The entire ORIGEN has around 300 lines of C code and 2K lines of Python code.

6.6 Experiments

This section empirically evaluates ORIGEN. First, I represent the experiment setup in Section 6.6.1, and then I systematically evaluate the accuracy of ORIGEN in the cross-version setting in Section 6.6.2 and Section 6.6.3. In Section 6.6.5, I apply ORIGEN into two use cases: memory forensics and VMI. Finally, I evaluate the runtime performance of ORIGEN in Section 6.6.6.

6.6.1 Experiment Setup

All experiments are conducted on a machine with Intel(R) Core i5 @ 2.9GHz and 16 GB DDR3-RAM running 64-bit Ubuntu 14.04. I evaluate ORIGEN on four sets of software families: including `Windows`, `Linux`, `OpenSSH` and `dm_crypt`, as shown in Table 6.1. To verify the accuracy of the proposed method, I systematically evaluate ORIGEN on `OpenSSH` family. For the rest of the software families, I conduct case study analysis on some representative versions.

The experimental set is representative for the following reasons: 1) the set is a sufficient sampling of real-word softwares. The versions in our experiments cover a span of 13 years of `OpenSSH`, from 2.2.0p1 in 2002 to 6.8p1 in 2015; 2) the data types and the structs in `OpenSSH` are rich and representative. For example, there are 1,904 structs and 22,618 fields in total for 40 versions of `OpenSSH`. Figure 6.5 illustrates the number of unique data types in each version. The size and diversity of the data should provide a systematic and objective evaluation for the proposed approach; 3) the source code of `OpenSSH` provide a gold standard for evaluating the performance of ORIGEN.

Evaluation Metrics: I employ `precision` to evaluate the performance of ORIGEN. Given a source version s, our task is to predict the offsets of correspondent data types in the target version t. The offset precision for the target version is calculated from:

$$precision = \frac{|\delta|}{|s \cap t|}, \tag{6.1}$$

where $|s \cap t|$ represents the total number of shared data field names in the two versions, and $|\delta|$ represents the number of correctly predicted offsets. The ground truth of the data field names can be directly obtained from the source code of `OpenSSH`. Note, the source code is not used in prediction.

Program	# of Ver	Start Ver		End Ver	
		Ver	Date	Ver	Date
`Windows`	3	XP3	2001	Wind7	2009
`Linux`	9	2.6.32	2010	3.13.0	2014
`OpenSSH`	40	2.2.0	2002	6.8.0	2015
`dm_crypt`	8	3.5	2012	3.13.0	2015

Table 6.1: Datasets of released versions

6.6.2 Overall True/False Positive Analysis

I also evaluate the accuracy of ORIGEN using the `OpenSSH` family. I use its 40 versions which covers a span of 13 years. Each version gets the true profile from its source code. I conduct the pair-wise profile generation on the 40 samples by ORIGEN, and calculate the offset prediction precision. For each version, ORIGEN utilizes it as a base version to localize profiles for all 40 versions. Each localized profile calculates precision by differing itself with the true profile from the source code of that version.

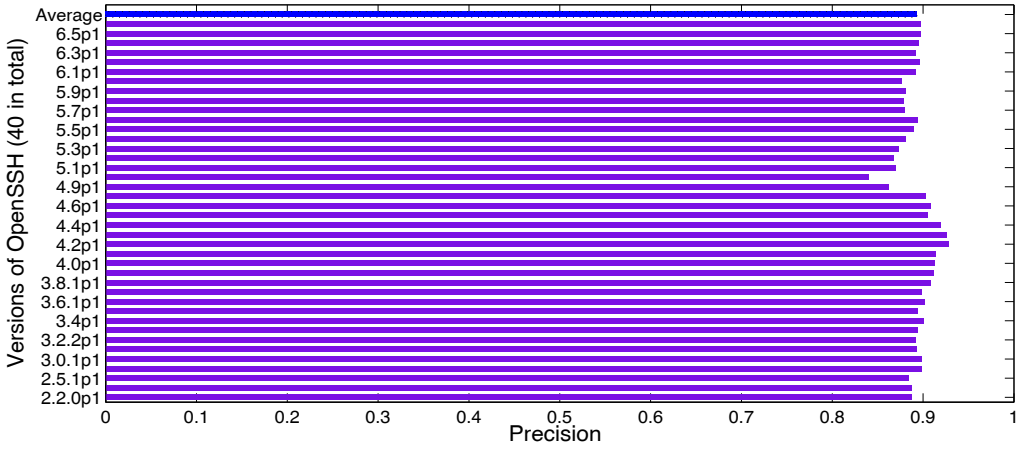


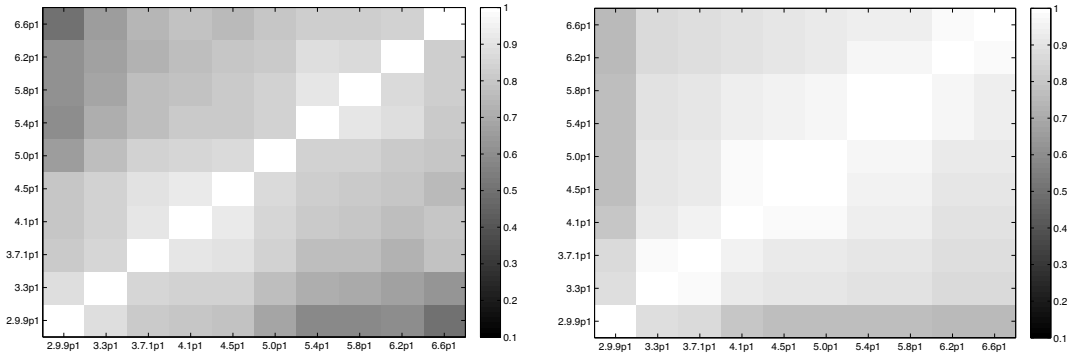
Fig. 6.6.: **The average precision of our method on 40 versions of OpenSSH. The dashed bar on top shows the average.**

Figure 6.6 shows the overall precision of ORIGEN on each test `OpenSSH` version, where the x-axis represents the offset prediction precision and the y-axis lists the versions. The dashed bar labeled as "average" on top represents the average precision across all 40 versions. As I see, on average, ORIGEN obtains a reasonable precision of 89.33%. The variance of the precision across versions is only 0.003, with the highest precision of 92.88% and the lowest of 83.98%. The small variance suggests that the proposed method

is robust. The results shown in Figure 6.6 substantiate the efficacy of ORIGEN and suggests that ORIGEN points to a feasible solution for cross-version memory analysis.

I inspect the results and hypothesize that the accurate result derives from two main reasons: 1) the most of field types are referenced by multiple ORIs. A single or a few ORI searching failures can be corrected through the voting mechanism; 2) the code search based cross-version inference is resistant to some data structure reorganizations. I calculate the statistics on ORIs for each field to explain the first reason. As shown in Figure 6.5, I can see that each data type has more than 50 ORIs for its fields on average. Any single or a few ORI searching failures can be re-corrected by rest of correct ORIs. I also manually investigate 40 reconstructed data profiles from Figure 6.6 and find that ORIGEN still correctly infers `connection_in` and other fields in `session_state` in `OpenSSH2.2`, even if `session_state` data structure first appears in `OpenSSH5.3`. The reason is that `OpenSSH5.3` creates `session_state` as a wrapper to wrap these fields in previous versions. The code accessing these fields are relatively stable. ORIGEN can still identify ORIs from these codes and update the type information.

I further inspect the false positives in our method and find most of false positives are caused by the inaccuracy of the code search technique used by ORIGEN. For example, Bindiff cannot yield good alignment results if source code are compiled from different compiler or different optimization level. I can further improve the accuracy of the binary alignment by leveraging more advanced techniques [41, 52, 64]. In this chapter, I will discuss how to address the false positive issue in Section 6.6.4.



(a) The offset prediction precision (b) The true data structure similarity

Fig. 6.7.: The illustration of pair-wise experiments on 10 representative versions of `OpenSSH`.

6.6.3 In-depth True/False Positive Analysis

I also conduct an in-depth analysis to evaluate the accuracy of ORIGEN. Figure A.5 presents detailed comparison results in the heat map. For the convenience of illustration, I only include 10 representative versions from 2.9.9p1 to 6.6p1, where each block indicates a pair-wise prediction experiment on the two versions. The brightness of the block in Figure A.5(a) shows the offset prediction precision for 100 pair-wise profile generations; in Figure A.5(b), the brightness indicates the true profile similarity for the 100 pairs.

We can see that ORIGEN exhibits better performance for adjacent versions, or in other words, it has the better performance when the time interval of two versions is smaller. For example, two adjacent versions of `OpenSSH` 3.3p1 and 4.5p1 have a very high offset prediction precision. This is reasonable, because two adjacent versions tend to have less differences in their binaries. In most cases, these differences in adjacent versions are from minor code changes such as security patches, so these two binaries still share most of similar codes. When the time interval of two versions is large enough, ORIGEN may not generate the profile with the good quality. In this case, we can either use the method in

Section 6.6.4 or create a new base model on the more recent version. The new model creation is much less frequent than the version change. In fact, we only need to create 2 models for the 40 versions of `OpenSSH`.

The true data structure similarity in Figure A.5(b) shows a good explanation about the performance of ORIGEN. Each true data structure similarity in this matrix is calculated by differing true data structures of two versions. We can see that most of adjacent versions can reach 100% similarity. When the time interval increases, the drop of data structure similarities is marginal. This also demonstrates that adjacent versions have few design changes and look similar. This results substantiate our intuition that software of different versions tend to be similar.

6.6.4 Handling False Positives

The accuracy of ORIGEN has been verified in Section 6.6.2. The average precision is about 90%, but there are still 10% false positives, which might not be desirable in some mission critical applications. To this end, I incorporate a thresholding method to reduce the number of false positives. The idea is that we can adopt the number of accesses to quantify the searching robustness of the data field type, and only consider the data field type above the threshold as the searching candidate. We admit it will sacrifice coverage for accuracy, but it is necessary for the practical integration in some cases.

The result as shown in Figure 6.8 illustrates the precision under different thresholds, where the x-axis lists the threshold, and the y-axis represents the precision. For each threshold, the 95% confidence interval of 40 versions is also plotted. As we see, the precision

increases along with the threshold, and a bigger threshold leads to a more accurate result, e.g. the precision is 98.53% under the threshold 32. As the threshold determines the searching robustness of the data type, a method with a bigger threshold behaves more prudently, and makes less yet more accurate predictions. For example, when the threshold is 2, our method yields 116,446 predictions; but when the threshold is 16, it yields only 42,324 confident predictions. The experimental results substantiate the claim that our method can be tailored to produce very few false positives.

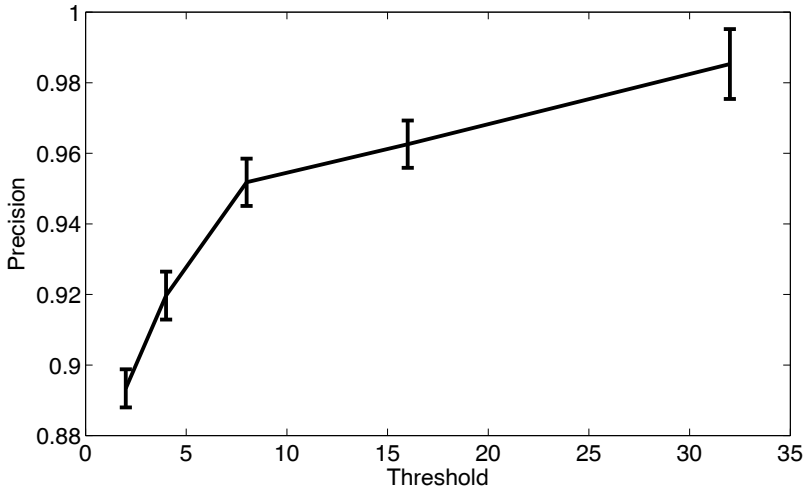


Fig. 6.8.: **Precision of our method under different thresholds.**

6.6.5 Case Studies

In this section, I conduct a qualitative analysis to evaluate the practice of ORIGEN. I select several key data fields in all of the software samples listed in 6.1 and conduct case studies in two application scenarios: virtual machine introspection and memory forensics.

For virtual machine introspection (VMI), I choose to enhance DECAF [72], the dynamic analysis platform. DECAF relies on VMI to retrieve the running processes and

loaded modules inside a virtual machine to analyze the behaviors of specified processes or kernel modules, for automatic malware detection and analysis. However, it only supports a limited number of guest OS versions (including Windows and Linux), due to the hard-coded profiles. To support a new guest OS version, a user must compile and load a kernel module inside the virtual machine to generate the corresponding profile. I aim to demonstrate that with help of ORIGEN, we can eliminate this manual task by automatically generating the profile from a given virtual machine image within just a few minutes. This case study can demonstrate how ORIGEN greatly improves the usability of VMI for the cloud provider.

For memory forensics, I show two forensic analysis tasks: OpenSSH session key extraction, and dm_crypt[2] encryption key extraction. I develop two plugins on Volatility memory forensics framework [2] to accomplish these two tasks, respectively. I aim to demonstrate that with help of ORIGEN, I can perform these analysis tasks in a cross-version manner. It means that without knowing the version information of the application in a memory dump, I can automatically create a localized profile and then immediately perform the forensic analysis on the memory dump.

I select key data fields as a demo for each analysis. The second column in Table 6.2 lists key data fields of interest. To be more specific, for Windows VMI, I need the global variable `PsActiveProcessHead` as the starting point to traverse the linked list of `EPROCESS`, and then within each `EPROCESS` object, I obtain the process ID in `UniqueProcessID`, the name in `ProcessName`, and so on. I visit the next `EPROCESS` object through `ActiveProcessLinks`. Similarly for Linux VMI, I need to start from `init_task` to

[2]dm_crypt is a disk encryption tool in Linux.

traverse the `task_struct` linked list and locate the process ID in `pid`, and the process name in `comm`, and so on.

In memory forensics scenario, for dm_crypt, I create a signature using the five fields in the structure `crypt_config` to scan the memory and find the actual encryption key in `crypt_config.key`.

I select three base versions for each software, as shown in Table 6.2. In order to evaluate the strength of ORIGEN, these test versions span several major revisions, ranging from `Windows XP`, `Linux 2.6.32`, and `OpenSSH 5.3`, to `Windows 7`, `Linux 3.13.0`, and `OpenSSH 6.5`.

Name	Field Name	ORI Statistic on Windows XPSP0			WinXPSP2	WinVista	Win7
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
Windows	EPROCESS.UniqueProcessId	5	7	12	√(12/0)	√(9/3)	√(9/3)
	EPROCESS.EitTime	0	2	2	√(2/0)	√(2/0)	√(2/0)
	EPROCESS.ActiveProcessLinks	1	3	4	√(4/0)	√(4/0)	√(4/0)
	EPROCESS.ProcessName	0	4	4	√(4/0)	√(3/1)	√(3/1)
	EPROCESS.PEB	5	2	7	√(7/0)	√(4/3)	√(4/3)
	EPROCESS.DirectoryTableBase	2	1	3	√(3/0)	√(3/0)	√(3/0)
	.data : PsActiveProcessHead	0	3	3	√(3/0)	√(3/0)	√(3/0)
Linux		ORI Statistic on Linux3.5.0			Linux2.6.32	Linux3.8.0	Linux3.13.0
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	.data: init_task	0	10	10	√(8/2)	√(8/2)	√(8/2)
	task_struct.tgid	9	1	10	√(10/0)	√(8/2)	√(8/2)
	task_struct.pid	8	2	10	√(5/5)	√(8/2)	√(7/3)
	task_struct.comm	1	4	5	√(4/1)	√(5/0)	√(5/0)
	task_struct.tasks	1	2	3	√(3/0)	√(3/0)	√(3/0)
	task_struct.mm	42	5	47	√(29/18)	√(37/10)	√(37/10)
	mm_struct.pgd	12	4	16	√(12/4)	√(11/5)	√(11/5)
OpenSSH		ORI Statistic on OpenSSH5.9			OpenSSH5.3	OpenSSH6.0	OpenSSH6.5
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	.bss: active_state	1	5	6	√(6/0)	√(6/0)	√(6/0)
	session_state.ssh1_key	0	2	2	√(2/0)	√(2/0)	√(2/0)
	session_state.ssh1_key_length	0	4	4	√(4/0)	√(4/0)	√(4/0)
dm_crypt		ORI Signature Statistic on Linux3.8.0			Linux3.5.0	Linux3.11.0	Linux3.13.0
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	crypt_config.cpher	1	3	3	√(3/0)	√(3/0)	√(3/0)
	crypt_config.cipher_string	1	4	4	√(4/0)	√(4/0)	√(4/0)
	crypt_config.iv_size	1	8	9	√(9/0)	√(9/0)	√(9/0)
	crypt_config.key_size	1	4	5	√(5/0)	√(5/0)	√(5/0)
	crypt_config.key	1	3	4	√(4/0)	√(4/0)	√(4/0)

Table 6.2: The efficacy of ORIGEN on different applications. **DL** denotes the dynamic labeling; **SL** for static labeling. **D** for "Detected". TP for correctly matched ORIs in the new version and FP for wrongly matched ORIs for the new version

ORIGEN can accurately generate a profile for each of the four analysis tasks, and the results are shown in Table 6.2. Table 6.2 lists the software family names to be tested, their base version and three test versions. For each software family, the ORI labeling and matched results are listed respectively. For ORI label, it shows the number of ORIs via the dynamic labeling (DL) and the static labeling (SL) respectively. The column of "Total" shows a sum of ORIs generated via two phases. For each test version, I also list the number of correctly labeled ORIs and missed ORIs respectively.

The results in Table 6.2 demonstrate three points. First, ORIGEN can precisely label ORIs in the base version for the data fields in each profile. We can see that each data field has more than one ORI in the base version. Second, the static labeling can improve the ORI coverage. By comparing the ORI number in DL column and total column, we see that the static ORI labeling can help find more ORIs. Finally, the error correction can help to reduce the false positive rate. I found that the profile localization for the four software families cannot find all semantically-equivalent ORIs for their test versions, but the error correction still helps to infer the accurate offset for each data type field in the generated profile. For example, there are 47 ORIs in total for the field `task_struct.mm` in the base version of Linux, `Linux 3.5.0`. However, ORIGEN only correctly finds 37 ORIs in `Linux 3.8.0`. By adopting the strategy one in discussed in Section 6.4.3, the correct offsets can still be found by filtering the false offset values from the false 10 ORIs.

The Demo of ORIGEN To the end, I show the dm_crypt key extraction result to demonstrate the effectiveness of ORIGEN shown in Figure 6.9. ORIGEN has not information about the version information for the test dm_crypt in the memory dump. It extracts the

binary from the memory dump and automatically generates the concrete profile for fields in Table 6.2. Then it utilizes the concrete profile and successfully extracts the dm_crypt key.

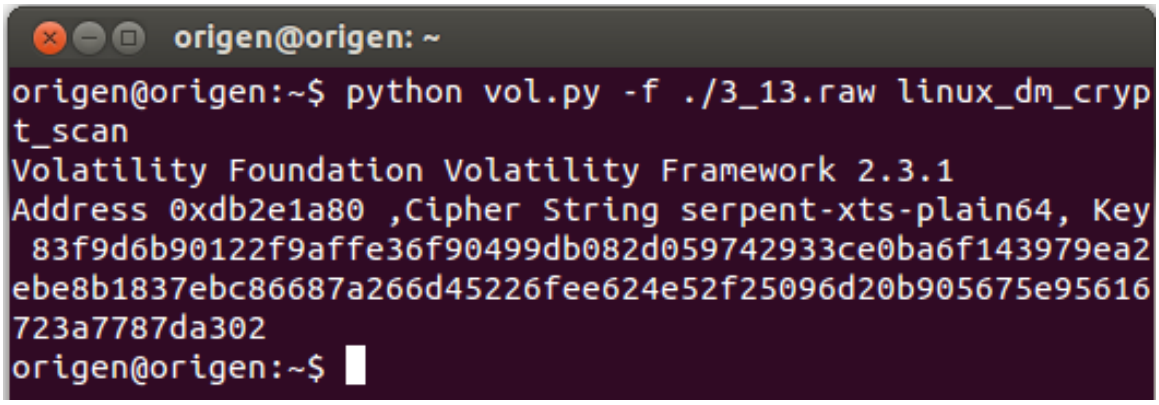


Fig. 6.9.: **The demo result of dm_crypt version-independent memory analysis.**

6.6.6 Runtime Performance

In this section, I verify the runtime performance of ORIGEN. Table 6.3 demonstrates the average running time of ORIGEN in Table 6.2. It includes the ORI labeling and the profile localization time.

We can see that it takes few seconds on average to finish the labeling for one ORI. Among steps of the ORI labeling, code disassembly takes up to 30 seconds for complex binary code like Linux kernel. The rest of steps such as the intra-procedural data-flow analysis only cause negligible runtime overhead. The profile localization takes several minutes to generate a profile. Most time is spent on the binary code alignment by BinDiff. It is reasonable, because conducting the alignment on the large scale binary is time consuming.

For VMI, ORIGEN takes around two minute to generate a profile for an unknown virtual machine image and then can immediately perform security monitoring from the hypervisor layer. This generation time could be greatly improved by conducting more efficient

code search technique. Our goal is not to completely resolve this problem but provide a promising solution for cross-version memory analysis.

Family Name	Total Time	
	ORI Labeling	Profile Localization
Windows	59 sec	1.1 min
Linux	1.3 min	3.2 min
OpenSSH	39.3 sec	18.4 sec
dm_crypt	24 sec	10 sec

Table 6.3: The total time for each application on average.

In this section, I conduct a qualitative analysis to evaluate the practice of ORIGEN. I select several key data fields in all of the software samples listed in 6.1 and conduct case studies in two application scenarios: virtual machine introspection and memory forensics.

For virtual machine introspection (VMI), I choose to enhance DECAF [72], the dynamic analysis platform. DECAF relies on VMI to retrieve the running processes and loaded modules inside a virtual machine to analyze the behaviors of specified processes or kernel modules, for automatic malware detection and analysis. However, it only supports a limited number of guest OS versions (including Windows and Linux), due to the hardcoded profiles. To support a new guest OS version, a user must compile and load a kernel module inside the virtual machine to generate the corresponding profile. I aim to demonstrate that with help of ORIGEN, we can eliminate this manual task by automatically generating the profile from a given virtual machine image within just a few minutes. This case study can demonstrate how ORIGEN greatly improves the usability of VMI for the cloud provider.

For memory forensics, I show two forensic analysis tasks: OpenSSH session key extraction, and dm_crypt[3] encryption key extraction. I developed two plugins on Volatil-

[3]dm_crypt is a disk encryption tool in Linux.

ity memory forensics framework [2] to accomplish these two tasks, respectively. I aim to demonstrate that with help of ORIGEN, I can perform these analysis tasks in a cross-version manner. It means that without knowing the version information of the application in a memory dump, I can automatically create a localized profile and then immediately perform the forensic analysis on the memory dump.

6.7 Discussion

In this section, I mainly discuss about the limitation and potential challenges.

Code Syntactic Changes I leverage the code search techniques to conduct the binary alignment for the profile localization. It is possible that some syntactic changes modify the control flow graph for the new version of a binary, such as inline functions or code optimizations. This can reduce the code search accuracy of ORIGEN. I summarize possible syntactic changes and list the robustness of the code search technique used by ORIGEN to these changes in Table 6.4. Fortunately, many related works have already focused on this issue and proposed more accurate search results [41]. The goal of the chapter is to explore the feasibility of ORIGEN. In the future, I will work on how to improve the accuracy of the generated profile by ORIGEN.

Code Semantic Changes ORIGEN by design can only infer the offset value for data fields which have been trained in the older version. If the data type is newly added, ORIGEN cannot infer the offset value for it. During the software development, it is common to add the security patches or redesign the code in the new version. These patches or code

reorganization could change the semantics of the older version. For example, the new version could add extra data types or remove some data fields. In these cases, ORIGEN will fail to generate the profile for these new coming data types. One possible way to sidestep this limitation is to train the additional model for the new version, and apply the new model to generate profiles for its similar versions.

Code Change	**Strength**
Register Assignment	Yes
Control Flow Flattening	Yes
Instruction Scheduling	Yes
Opcode Selection	Yes
Function Parameters	Yes
Function Inlining	Maybe
Calling Convention	Partial

Table 6.4: Robustness Analysis

6.8 Related Work

Code Search in Binary and Its application The code search technique recently has attracted much attentions. Most previous work put their efforts on the performance improvement for searching semantic equivalent codes in code database [41, 51, 52, 64, 86, 89, 91, 99, 107, 113, 126, 126]. Many researchers also applied these promising code search algorithms into different applications [37, 73]. Bug search utilizes the search techniques to quickly identify the program bugs [53, 86]. Patch generation applies the code similarity techniques to the semantic code discovery. Program lineage exercises the code similarity methods to infer the evolutionary relationship among a collection of software. Software plagiarism and repackage discovery also adopts the code search techniques [80], and so on. This chapter is the first attempt at the cross-version memory analysis by leveraging

the code search techniques. The experiments also shows it is promising to apply the code search techniques for the across-version memory analysis.

Memory Forensics Several memory analysis tools [2, 9, 62, 98, 111, 121] etc. have been proposed to aid the automatic memory forensics. They aim at analyzing and retrieve sensitive information from a memory dump. A key aspect of memory forensics is to encode the semantic related information into the data structure profile and follow the profile to conduct the specific analysis. The profile is predefined to the specific version of the image being analyzed, and update the profile according to versions of the target software.

State-of-the-art techniques rely on reverse engineering to reconstruct the profile of semantic information. The reverse engineering most often requires the manual effort or use non-trivial scripts [10] that operate on the source code. In this chapter, I propose the idea of cross-version memory analysis. Instead of reverse engineering version by version, it transfers the knowledge from the trained model for the older version to generate the profile for the new version.

Virtual Machine Introspection (VMI) VMI extracts semantic knowledge from a running virtual machine to monitor and inspect semantic behaviors of the guest machine. Due to the nature of isolation, VMI has been applied for many security applications. For example, many intrusion detection applications utilize the VMI technique to conduct more accurate detections [65, 108, 109]. Some malware analysis approaches also relies on the VMI to capture the detail malware behaviors which cannot be captured by previous work [43, 83].

Furthermore, VMI techniques are also well used in memory forensics and process monitoring [71].

The main challenge in the VMI technique is to bridge the semantic gap between the guest OS and outside analysis tools. Many existing works have already made a great step on this problem [48, 61]. A recent tool, DECAF [72] performs VMI to retrieve key semantic information from a guest OS. In each of the above efforts, similar to memory forensics, non-trivial efforts are required to construct a profile. Although VMST can reuse the OS code pieces of the introspection property to achieve the automatic VMI, the approach used in VMST could not be general enough to support the automatic introspection for some internal and close-sourced data structures.

Data Structure Reverse Engineering Reverse engineering data structures from binary executables is very valuable for many security problems. Particularly, Howard [130] and REWARDS [93] make use of dynamic binary analysis to recover the types and data structure definitions from the execution of a binary program. For each instruction during the execution, they infer and propagate the types of the instruction operands. Certain memory access patterns also need to be recognized to discover specific data structures like arrays, linked lists, and embedded data structures. For most COTS binaries without well defined documentation about their function prototypes, Howard [130] and REWARDS [93] can only infer the primitive data types such as integer, string or pointers. The manual efforts are still required for higher semantic data type inference. In this chapter, ORIGEN is proposed to alleviate the manual efforts. Instead of inferencing the data types for new version

of a binary from the scratch, ORIGEN can utilize the knowledge from data types in the older version which has been analyzed to assist the profile generation for the new version.

6.9 Summary

In this chapter, I presented the notion of "cross-version memory analysis". I detailed a solution and implemented a prototype called ORIGEN that is able to search the code in one binary, and locate the ORIs in another version of the code. The experimental results verified our claims. Specifically, our method successfully recovers the offsets for key semantic fields across different versions of OpenSSH, Windows, Linux, a loadable kernel module for Linux. In addition, it achieved a precision of 90% on 40 versions of OpenSSH. The experiments also demonstrated the efficiency of our method, where it took half a minute to identify all the chosen semantic fields on Windows and Linux respectively. Finally, I integrate ORIGEN into DECAF to demonstrate its effectiveness in VMI.

7. SUMMARY AND FUTURE WORK

In the era of explosion of applications and devices, the security analysis technique faces the " big data" challenge. Firstly, discovering vulnerabilities in millions of applications or devices is like finding a needle in a haystack, even when we are dealing with known vulnerabilities. Secondly, the memory analysis could be required to handle hundreds of memory dumps at one time. The existing memory analysis tools could not keep up with handling memory dumps on an unprecedented scale.

The fundamental problem of existing vulnerability identification and memory analysis techniques lies in that they are not scalable. The thesis of this work is that scalable and accountable code search technique enhances the efficiency and accuracy of vulnerability identification and memory analysis via an effective data reduction and knowledge reuse.

To address the challenges I discussed before, I propose a scalable and accountable binary code search framework. It can search the code in the large code database with the real-time efficiency, and also provide the search explanatory to aid analysts for match result inspection. To address the scalability issue in the binary code search, instead of comparing binary code with "raw features", I adopt the machine learning approach to learn "high-level" numeric features from raw features without the loss of essential semantic information. The resulting numeric feature representation can be conveniently indexed via mature hashing techniques, and therefore produce real-time search speed. This work is published in CCS 2016 [57]. I have implemented a bug search system (GENIUS), and

compared GENIUS with the state-of-the-art bug search approaches. The extensive experimental results show that GENIUS can achieve even better accuracy than the state-of-the-art methods, and is orders of magnitude faster than most of the existing methods.

I have introduced the novel feature representation, "conditional formula" to address the accountability issue of the binary code search. A conditional formula explicitly captures two cardinal factors of a code logic: 1) data dependencies and 2) condition checks. As a result, the binary code alignment on conditional formulas produces a one-to-one optimal behavior mapping which provide meaningful logs for human analysts to further examine the search results. I have implemented a prototype, XMATCH, and evaluated it using the well-known software `OpenSSL` and `BusyBox`. Experimental results had shown that XMATCH outperforms existing bug search techniques. At the same time, it also provided evidence of detected vulnerabilities, which can then be easily examined via human inspection.

To better understand the binary code search problem, I systematically investigate the code reuse phenomenon in common software libraries, and design a code-search based approach to automatically update the data profile to enable the cross-version memory analysis. This work is published in ACSAC 2014 [55] and AsiaCCS 2016 [56]. Our method successfully recovers the offsets for key semantic fields across different versions of OpenSSH, Windows, Linux, a loadable kernel module for Linux. In addition, it achieved a precision of 90% on 40 versions of OpenSSH. The experiments also demonstrated the efficiency of our method, where it took half a minute to identify all the chosen semantic fields on Windows and Linux respectively.

In conclusion, GENIUS, XMATCH and ORIGEN perform the scalable code search for the vulnerability detection and memory analysis. These tools improve upon the current state of

the art, providing empirical results applicable to real-world problems. Future work, as mentioned throughout the dissertation, will be required to address the limitations of each tool and extend their functionalities to handle additional hardware features and architectures.

APPENDIX

A. MACE: HIGH-COVERAGE AND ROBUST MEMORY ANALYSIS FOR COMMODITY OPERATING SYSTEMS

A.1 Introduction

Memory analysis has become increasingly valuable in digital crime investigation and malware analysis, as it extracts *live* digital evidence of attack footprints from the volatile memory state of a running system, which cannot be obtained from traditional hard disk based forensic analysis. Memory analysis is particularly beneficial for cloud computing security, because one can quickly scan a large number of virtual machine states to detect malicious activities, without installing security agents (which is inconvenient and can be easily subverted) inside the virtual machines. For example, a recent work proposed to detect rootkit infestation in homogeneous virtual machines in the cloud [23].

However, there exist several long-standing challenges in memory analysis especially for closed-source operating system (OS) such as Microsoft Windows.

(1) Low coverage. Without access to the source code of the commodity operating system, memory analysis tools can only resort to public symbols and documentations. As a result, these tools (e.g., Volatility [2]) can only identify documented objects (whose definitions are publicly available) and follow the pointers whose target types are also documented.

(2) Ambiguous pointers. Generic pointers (e.g., `void *`, `LIST_ENTRY`, and `struct list_head`) are prevalent in data structure definitions. It is hard to determine the exact

target types for these generic pointers, and it is common for a generic pointer to have multiple type candidates. As a result, it is difficult to follow these generic pointers to identify the target objects. Pointers can also be dangling, and following the dangling pointers would lead to extraction of bogus objects.

(3) Lack of robustness. Because of such a low coverage, it is very easy for kernel attacks to evade memory analysis. Hiding an object can be as simple as manipulating the incoming links that are followed by the analysis tools. For example, FU rootkit [60] hides a process by unlinking the corresponding `EPROCESS` object from the active process list. To evaluate the validity of a memory object, the existing tools often rely on constraints that can be easily violated, such as pool tags, string constants, object lengths, etc. In Section A.5.4, I demonstrate a synthetic attack that can completely defeat the utilities in Volatility by breaking these soft constraints.

Up to now, prior research efforts have been focused on tackling only one or two challenges above. No solution can address all the challenges in a holistic fashion. To improve robustness, several robust signature schemes have been proposed [46, 94]. These signature schemes can reliably detect important kernel objects by checking invariants (either strong value invariants or points-to relationship) in the kernel data structures. These signatures may not be distinct enough or may not even exist for many kernel objects (especially small ones). Therefore, we cannot rely on these robust signature schemes to achieve high coverage, not to mention that performance overhead is high for repeatedly searching signatures one by one throughout the memory.

Some efforts on data structure reverse engineering (e.g. REWARDS[93] and Howard[130]) may help extract kernel data structures definitions from commodity OSes. Potentially these

system can help identify previously undocumented objects and links, and thus improve the coverage. However, these systems have only demonstrate their capabilities on relatively small user-level programs. Complete reverse engineering of kernel data structures is still a daunting task due to the complexity of the commodity OS kernel code and the kernel data structures. In this chapter, we present MACE[1], a holistic solution that meets all the following requirements:

1. **Binary only approach.** MACE uses only the binary code of an OS, the public symbols, and documented data structure definitions to perform memory analysis. As a result, MACE is well suited for external forensic analysts to analyze closed-source OS like Windows.

2. **Robustness.** To achieve high robustness, MACE relies on *only* points-to relations (or pointer constraints), which are generally hard to violate, to identify kernel objects. Furthermore, MACE evaluates both deterministic and probabilistic pointer constraints throughout the entire kernel memory space, to find a nearly optimal solution. Therefore, even if an attacker manages to manipulate some pointers, these "injected" errors would likely be corrected by the remaining pointers in the memory during this *global* evaluation process. Thus, the attack impact on the overall identification results is minimized.

3. **High coverage and accuracy.** MACE can reconstruct a nearly complete kernel object graph, which consists of both documented and undocumented kernel object instances, and the connections among them. For undocumented objects, MACE can

[1] MACE stands for Memory Analysis through Correlative Evaluation.

further discover certain type information for the pointer fields in these objects. For instance, MACE can identify function pointers and target types for data pointers.

4. **Good efficiency.** MACE can scan a memory image and build a kernel object graph just a few minutes[2]. In contrast, the existing robust signature schemes [46, 94] use several minutes to only identify objects of a single type.

The core idea of MACE is to conduct supervised learning on pointers. That is, I first collect pointer constraints from a set of training memory images, in which kernel objects are correctly labeled by dynamic binary analysis. With the collected pointer constraints, I then perform probabilistic inference on pointers in an unlabeled memory image in a collective and correlative manner, to correctly label the pointers in the image. From these labeled pointers, I then reconstruct a nearly complete kernel object graph, for memory forensic purposes.

I leverage a key insight that the kernel object graph is a small-world network [55]: most kernel objects can be reached from other kernel objects within a few hops. A link from one object to another imposes a type constraint (either deterministic or probabilistic) on each side. The type constraint indicates the likelihood of a directly connected object to be of a particular type. The type constraints will accumulate and propagate to the objects that are not directly connected. Eventually, these constraints will be broadcast to the entire network until a convergence is reached.

I evaluated MACE for two closed-source operating systems: Windows XP SP3 and Windows 7 SP0 and found that MACE can achieve high recall and precision (95% and

[2]The current implementation of MACE is mostly in Python for quick prototyping. A C/C++ implementation would further reduce the analysis time to tens of seconds.

96%, respectively) for Windows XP and Windows 7. The errors mostly come from undocumented objects and volatile memory allocations.

I further evaluated the performance of MACE on memory images infected with real-world malware samples to demonstrate how MACE facilitates kernel rootkit identification. With a more complete coverage of kernel objects, MACE recognized malicious function pointers in both documented and undocumented data structures, and detected hidden objects more reliably. At last, I devised two synthetic kernel attacks to show how fragile the existing memory analysis tools (such as Volatility) can be, and how resilient MACE is against these attacks.

A.2 Problem Statement & Overview

A.2.1 Problem Statement

Given a memory image, we aim to reliably identify nearly all the kernel objects and connections between them, without access to the OS source code. I rely on public symbols, public data structure definitions. This public knowledge is used by the existing memory analysis tools (e.g., Volatility [2]). I attempt to improve the coverage and robustness of these third-party memory analysis tools, by leveraging the same amount of knowledge.

In addition to identifying documented kernel objects, we also aim to extract partial knowledge of undocumented kernel objects. In particular, we would like to discover types of the pointer fields, including both data pointers and function pointers. This knowledge on pointers can help obtain a big picture of the entire kernel object graph and benefit security analysis on this graph (e.g., kernel rootkit detection). In other words, our goal is not to

reverse engineer the kernel data structure definitions as REWARDS [93] and Howard [130], although our technique can be combined with these techniques to improve the quality of data structure reverse engineering.

I formalize the problem of kernel object labeling as follows: $M = \{m_i | 1 < i < |M|\}$ denotes the kernel memory space, where m_i is the ith machine word and $|M|$ is the total number of machine words in the kernel address space. Our goal is to assign a label l to each m_i. A label l is defined as a pair of object type and offset $l = (t, o)$, where $t \in T$ and $o \in [0, \texttt{sizeof}(t))$. Here T denotes the space of all object types.

To ensure high robustness, our solution cannot rely on soft constraints that can be easily manipulated by attackers, such as integer and string constants. For example, checking pool tags and the object size from the `OBJECT_HEADER` in Windows definitely helps verify the object types (both Volatility [2] and MAS [40] use this method to resolve type ambiguity). However, kernel rootkits can easily violate these soft constraints to evade and mislead these memory forensic tools. It means that our solution can only rely on pointer constraints, which are more difficult to tamper with. I should also anticipate that although complete sabotage of pointer constraints is not possible, attackers may manage to manipulate a certain amount of pointers. Therefore, our solution should tolerate pointer manipulation attacks to a certain degree.

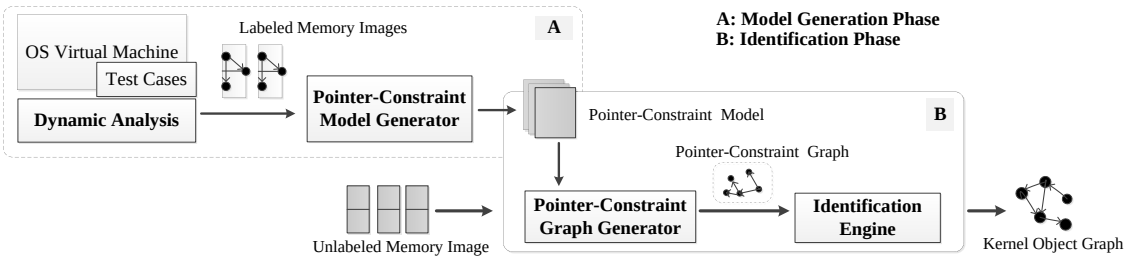


Fig. A.1.: **System Overview.** The model generation phase A outputs the pointer-constraint model. The identification phase B detects the kernel object graph on the unknown memory image.

A.2.2 System Overview

I propose to take a probabilistic inference approach to label kernel objects based on their pointer constraints. Figure A.1 depicts this workflow. Essentially, I propose a supervised learning technique. In the model generation phase, I perform dynamic binary analysis on the OS kernel to label kernel objects and learn a pointer-constraint model. Then in the identification phase, I will use this model to identify kernel objects in an unknown memory image.

Model Generation Phase For a closed-source operating system (like Windows), I perform dynamic binary analysis to label kernel objects while the OS is running inside a virtual machine. These labeled kernel objects are then used to generate the pointer constraint model, which captures the probabilistic type constraints between pointer fields. To ensure the training is well-rounded, I conduct a set of test cases to exercise different components of the operating system, such as filesystem, network, IO, process/module/thread management, etc. Consequently, the recorded memory images (with labeled kernel objects) capture diverse system states under these workloads.

If the source code of an OS is available, I could generate this pointer-constraint model in two ways. I could perform points-to analysis on the source code directly to generate extended type graph (as done in KOP [30] and MAS [40]), and then simply derive a model from the extended type graph. For a generic pointer, I would have to assign equal probability to each possible target type. To generate a model that better reflects the system states at runtime, I could also perform dynamic analysis described above. It means that if a target type for a generic pointer appears more often than the others at runtime, it would have higher probability. The generated model would lead to better classification results.

Identification Phase Given the pointer-constraint model for one OS version and an arbitrary memory image of the same OS version, in the identification phase, MACE tries to identify kernel objects and their pointer relationships.

The problem of labeling pointers in the memory image based on the pointer constraints is equivalent to searching the optimal type assignment for each pointer under the given constraints. A plausible solution to this problem is Maximum Likelihood Estimation (MLE), wherein every possible assignment solution is enumerated and evaluated in terms of likelihood, i.e., the number of satisfied constraints. However, this solution proves to be NP-hard. Therefore, it is too expensive to iterate through all possible types for tremendous amount of pointers in the memory.

I approach this problem by using the *random surfer model* [70], which has been commonly used for complex networks, such as page ranking on the web [70]. Intuitively, in random surfer model, a score associated with each node in the graph is equivalent to the likelihood of this node being visited by the "random surfer". The likelihood of a node be-

ing visited is determined by how likely its neighbors are visited and how likely the "surfer" travels from a neighbor to this node. The random surfer model allows for effectively evaluating the scores for all nodes in the graph such that it is scalable even for very large graphs (e.g., the internet). To the best of our knowledge, I are the first to apply the random surfer model to the problem of memory analysis.

In our problem domain, a node represents a labeled pointer, and an edge from one node to another dictates a confidence level that the source pointer has on the target pointer. In other words, the confidence level is a conditional probability on how likely the target pointer is correctly labeled given the source pointer is correctly labeled. I call this graph a *pointer-constraint graph*. I then apply the random surfer algorithm to calculate a nearly optimal score for each node (i.e., a pointer with a particular label). Finally, I compute object-level scores based on these pointers' scores and identify true kernel objects.

A.3 Model Generation

For a closed-source operating system, I generate the pointer-constraint model in two steps: 1) I conduct dynamic analysis to label kernel objects in the training data set; and 2) I learn the model by conducting statistical analysis on the training data.

A.3.1 Labeling Kernel Objects

I monitor the execution of the OS kernel and observe how kernel objects are allocated and de-allocated, and how these kernel objects are connected with each other. As we ob-

serve the actual binary execution of the OS kernel, we can obtain the ground truth, which is typically hard to get otherwise.

I leverage the dynamic analysis framework DECAF [4] to monitor the execution of an OS and construct the kernel data structure graph on the fly. In general, I monitor and label three kinds of kernel objects. I monitor kernel modules (e.g., `ntoskrnl.exe` and device drivers) by hooking `MmLoadSystemImage`. This is important because global data variables are located in these kernel modules. I hook `ObCreateObject` to monitor and label documented kernel objects (e.g., `EPROCESS`). Windows uses this function to create managed kernel objects (which are all documented). I hook `ExAllocatePoolWithTag` and `ExFreePoolWithTag` to obtain a view of live memory objects in the dynamic memory pools for other objects. While there are other functions to allocate and free memory regions in the kernel, these two are the root functions. All the functions to be hooked are located in the main kernel component `ntoskrnl.exe`, and these functions' offsets can be obtained from the public symbol information.

In this way, I can precisely label kernel modules and documented kernel objects. However, I rely on their pool tags obtained from the `ExAllocatePoolWithTag` function call for undocumented objects. This pool tag labeling mechanism is fairly common in the modern OSes. For example, SLAB in UNIX-like systems is a similar mechanism. Of course, several problems may arise if I simply label undocumented objects by their pool tags: 1) an object allocated with a pool tag may consist of multiple inner objects, which become invisible; and 2) objects of the same type may be allocated using several different pool tags. As reverse engineering undocumented objects is not only main goal, I accept

these limitations and leave a better labeling approach as future work. For example, we could leverage the calling context of the memory allocation routine to label the object.

Certainly, these function hooks are specific to Windows. For another closed-source operating system, we will need to rely on its public documentation and public symbol information to find a set of functions to hook and label objects properly. The general principle should remain the same.

To recognize links between these kernel objects, I examine each double-word within each object and see if the value in the double-word falls in the memory region of any kernel object. If this is true, I treat this value as a pointer field and we establish a link between these two kernel objects. Note that in the kernel space, it is common for a pointer to point to the middle of a kernel object. This approach may lead to an overestimation in our study because a non-pointer field may happen to have a pointer-like value and thus be treated as a pointer. In practice, these pointer-like data fields will not affect the detection accuracy of MACE, because these noises are filtered out in the statistical analysis (described in Section A.3.3). Moreover, pointer fields may not be 4-byte aligned in certain packed data structures, so I have to search double-words in all byte locations.

A.3.2 Test Cases

In order to ensure that the generated model has a diverse set of kernel objects, the test programs used for dynamic analysis need to activate different OS functionalities that are as diverse as possible. I include both standard OS benchmark and common software programs to be run in the guest OS to maximize the variety and number of kernel objects created. For

(a) Labeled Memory Image

Addr	Value	Label	Addr	Value	Label	Addr	Value	Label	Addr	Value	Label
0x8000	0x80B0	(A,0)	**0x8034**	0x80A8	(A,4)	**0x8068**	0x80BC	(A,8)	**0x809C**	0x1	(A,c)
0x8004	0x80A8	(A,4)	**0x8038**	0x80BC	(A,8)	**0x806C**	0x1	(A,c)	**0x80A0**	0x1234	(C,0)
0x8008	0x80BC	(A,8)	**0x803C**	0x1	(A,c)	**0x8070**	0x80B0	(A,0)	**0x80A4**	0x80B0	(C,4)
0x800C	0x1	(A,c)	**0x8040**	0x80B0	(A,0)	**0x8074**	0x80BC	(A,4)	**0x80A8**	0xff	(D,0)
0x8010	0x80B0	(A,0)	**0x8044**	0x80A8	(A,4)	**0x8078**	0x80BC	(A,8)	**0x80AC**	0x80B0	(D,4)
0x8014	0x80A8	(A,4)	**0x8048**	0x80BC	(A,8)	**0x807C**	0x1	(A,c)	**0x80B0**	0x1234	(B,0)
0x8018	0x80BC	(A,8)	**0x804C**	0x1	(A,c)	**0x8080**	0x80B0	(A,0)	**0x80B4**	0x1	(B,4)
0x801C	0x1	(A,c)	**0x8050**	0x80B0	(A,0)	**0x8084**	0x80BC	(A,4)	**0x80B8**	0x8000	(B,8)
0x8020	0x80B0	(A,0)	**0x8054**	0x80BC	(A,4)	**0x8088**	0x80BC	(A,8)	**0x80BC**	0xff	(E,0)
0x8024	0x80A8	(A,4)	**0x8058**	0x80BC	(A,8)	**0x808C**	0x1	(A,c)	**0x80C0**	0xff	(E,4)
0x8028	0x80BC	(A,8)	**0x805C**	0x1	(A,c)	**0x8090**	0x80A0	(A,0)	**0x80C4**	0x8008	(E,8)
0x802C	0x1	(A,c)	**0x8060**	0x80B0	(A,0)	**0x8094**	0x80BC	(A,4)	**0x80C8**	0	(D,0)
0x8030	0x80B0	(A,0)	**0x8064**	0x80BC	(A,4)	**0x8098**	0x80BC	(A,8)	**0x80CC**	0x80B0	(D,4)

(b) Generated Pointer Constraint Model

KOB	Offset Constraint	Target Constraint
		TC[(A,0)] = {(B, 0, 0.9), (C, 0, 0.1)}
A: 0x10	OC[A] = [0, 4, 8]	TC[(A,4)] = {(D, 0, 0.5),(E, 0, 0.5) }
B: 0x0c	OC[B] = [8]	TC[(A,8)] = {(E, 0, 1)}
C: 0x08	OC[C] = [4]	TC[(B,8)] = {(A, 0, 1)}
D: 0x08	OC[D] = [4]	TC[(C,4)] = {(B, 0, 1)}
E: 0x0c	OC[E] = [8]	TC[(E,8)] = {(A, 8, 1)}
		TC[(D,4)] = {(B, 0, 1)}

Fig. A.2.: An example of pointer-constraint model: (a) the labeled memory image for an OS version; (b) is the pointer-constraint model inferred from the labeled memory image. The first column of (b) means the object type and the size.

the standard OS benchmark, I choose `lmbench` [6], as it performs several diverse actions in networking (TCP, UDP, RPC, and pipe), filesystem (file creation and deletion, cached file read, etc.), signal handling, memory access, etc. I also select several common and complex programs to further increase the training coverage, including web browsers, media players, word processors, and PDF readers.

A.3.3 Statistical Analysis

Without source code, I conduct statistical analysis to learn kernel objects and their relationships based on labeled memory images. Specifically, I utilize the pointer constraint model to represent the kernel objects and their relationships. The pointer constraint model includes offset constraints and target constraints.

Offset Constraints: An offset constraint dictates the pointer offset in the kernel object. For example, the offset constraint OC(A) represented in Figure A.2(b) shows that Object A with 12 bytes has three pointers at offset 0, 4, and 8.

To learn offset constraints for each object type t, I go over all the instances of that object type and examine the pointer fields in them. An offset o appears in the offset constraints of t if and only if all the instances of t have valid pointers at offset o. For example, I can learn that OC[A]=[0,4,8], since all instances of A in Figure A.2(a) have pointer-like values at offset 0, 4 and 8.

Target Constraints: A target constraint is imposed on the target of a pointer field. It includes the target type and the probability indicating how likely the pointer target is of the particular type. The target constraint is also can be learned through statistical analysis. By iterating through all labeled pointer fields and their targets in the training memory dumps, I can compute these target constraints. For example, the statistical analysis on instances of object A in Fig. A.2(a) learns that the pointer at offset 0 in object A has two target labels, $(B, 0)$ and $(C, 0)$. By counting the numbers of instances of A with different target types, I also compute the probabilities of $(B, 0)$ and $(C, 0)$ to be 0.1 and 0.9 respectively. The target constraint ($TC(A, 0)$) is shown in Fig. A.2(b).

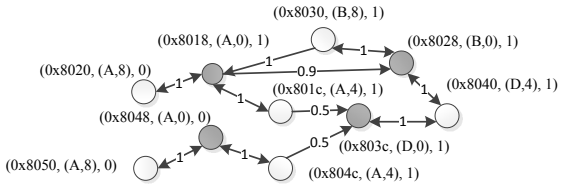
Variable-Length Arrays The variable-length array is handled in the different manner, since its size is not a constant. I discover variable-length arrays using two conditions. (1) the size of a variable-length array is variable; (2) each entry of the array should have the same target type or a NULL pointer. This means that I only focus on object types with the variable size. For each object type with the variable size, I can determine the variable-length array by checking whether all entries of its instances share the same target type or zero. The arrays in our model will be labeled as "array" without the specific size. Its target offset constraint will be normalized to be relative to the start address of their hosting elements, instead of the base of the array.

A.4 Kernel Object Identification

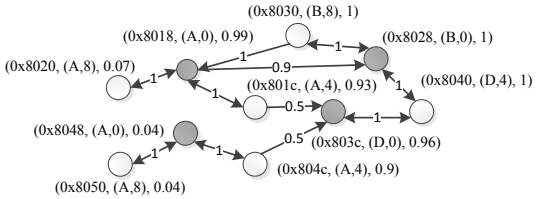
Given an arbitrary memory image, MACE tends to identify its kernel objects based learned constraint model from the training phase in the following steps: (1) it constructs a pointer-constraint graph from the memory image; (2) it applies the Random Surfer algorithm on the pointer-constraint graph until a convergence is reached; and (3) it selects true kernel objects based on the final scores on the pointer-constraint graph.

Address	Value	Address	Value
0x8018	0x8028	0x8038	0x4
0x801c	0x803c	0x803c	0x7
0x8020	0x803c	0x8040	0x8028
0x8024	0x1	0x8044	0x8
0x8028	0x11	0x8048	0x8034
0x802c	0xc	0x804c	0x803c
0x8030	0x8018	0x8050	0x8034
0x8034	0x6	0x8054	0x1

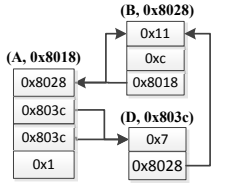
(a) Unlabeled memory image



(b) Initial constraint graph



(c) Converged constraint graph



(d) Identified kernel object graph

Fig. A.3.: An example for random pointer surfing. A solid node in the graph represents a pointer with offset 0, indicating the base of a kernel object.

A.4.1 Pointer-Constraint Graph Construction

Definition A.4.1 *The pointer-constraint graph is a directed graph* $G = (V, E)$. *A node* $v \in V$ *is a tuple* $(a, (t, o), r)$, *where* a *is the address of a pointer,* (t, o) *labels the pointer as the object type* t *and the offset* o *with the object, and* $r \in [0, 1]$ *is the score indicating how likely this pointer is labeled correctly. An edge* $e \in E$, $e = (u, v, r)$ *represents the constraint from node* u *to node* v, *where* $r \in [0, 1]$ *specifies the conditional probability how likely* v *being correctly labeled if* u *is correct.*

As an example, I show a pointer-constraint graph in Figure A.3(b), which is constructed from an unlabeled memory image in Figure A.3(a), using the model presented in Figure A.2. Algorithm 1 describes how to construct a pointer-constraint graph.

I construct a pointer-constraint graph, starting with a number of "root" nodes, and then perform breadth-first traversal to add new nodes and edges into the graph. To find root nodes, I select kernel objects (e.g., `EPROCESS` and `ETHREAD`) that have many pointer

fields inside and thus their offset constraints are fairly unique. I use these offset constraints to scan the kernel memory and find possible kernel objects. Then a root node is created for each pointer field of these objects. Some of these root nodes may be in fact wrong. Their scores may be updated during the evaluation of their target constraints. At last, I will rely on the random surfer algorithm (described in Section A.4.2) to evaluate their authenticity. For example in Figure A.3, I choose object A to find "root" nodes, since it has the most number of offset constraints. If I scan the memory image using A's offset constraint, I find two instances for A. Therefore, I create root nodes for object A. MACE will create a node for each offset constraint of object A. I assign 1 to node $(0x8018, (A, 0), 1)$, $(0x801c, (A, 4), 1)$, $(0x8020, (A, 8), 1)$, $(0x8048, (A, 0), 1)$, $(0x804c, (A, 4), 1)$ and $(0x8050, (A, 8), 1)$ because at this stage, their offset constraints are all met.

To further expand the constraint graph, we need a working queue Q to perform this breadth-first traversal. To begin with, the root nodes are enqueued into Q. Then, on each iteration, a node v is dequeued from Q. I retrieve from the model PCM the target constraints TC for this node v. If v does have target constraints, I go over each target constraint tc in TC. tc tells us a possible target label $(tc.t, tc.o)$ and its likelihood $tc.r$. Then, to check if this target label is compatible with the target memory, I check the offset constraints of the target type $tc.t$ and the memory region starting at the corresponding object base address $M[v.a] - tc.o$. This is done in $PCM.CheckOC()$ function. In other words, I check if the memory words at the offsets specified in the offset constraint are valid addresses. If this is not true, this target type can be not valid, so I check the next target constraint in TC. Otherwise, I extend G into the target object.

Algorithm 1: Pointer-Constraint Graph Construction

```
Input: Memory image M,Pointer-Constraint Model PCM
Output: Pointer-Constraint Graph G
Q ← G.V;
while Q ≠ ∅ do
    v ← Q.dequeue();
    TC ← PCM.GetTC(v.t, v.o));
    if TC ≠ ∅ then
        Matched ← False;
        for each tc ∈ TC do
            if PCM.CheckOC(M, M[v.a] − tc.o, tc.t) = True then
                Matched ← True;
                u ← (M[v.a], tc.t, tc.o, 1);
                if u ∉ G.V then
                    G.AddNode(u);
                    Q.enqueue(u);
                end
                G.AddEdge(v, u, tc.r);
                AddObjtoG(G, M[v.a] − tc.o, tc.t);
            end
        end
        if Matched = False then
            v.r ← 0;
        end
    end
end
return G;
```

To extend the graph G, I first check if the target node u (with the same address and label) has been already created. If not, I validate that its pointer locations are compatible with its offset constraints. For all the nodes that pass the validation check, I create u and set its initial score to 1. Otherwise I will set the score to be 0. I also enqueue u to Q for the subsequent breadth-first traversal. An edge $(v, u, tc.r)$ is added into G, where the edge's likelihood is obtained from the target constraint tc. For example, object A candidate fails the target checking at $0x8020$, $0x8048$ and $0x8050$, MACE has to update the

value 1 to 0 for the node $(0x8020, (A, 8), 0)$, $(0x8048, (A, 0), 0)$ and $(0x8050, (A, 8), 0)$ in Figure A.3(b).

Now I need to add the rest of pointers (if any) in the target object into G. To facilitate subsequent object-level classification, I always create a "base" node (whose offset is 0) for each object, even if the field at offset 0 is not a pointer. I further use this "base" node to bind all the pointers in that object by adding both incoming and outgoing edges between the "base" node. In this way, the scores on the pointers within one objects can flow back and forth to each other until a convergence is reached. In Figure A.3, these "base" nodes are marked as solid circles.

If it turns out that none of the labels in target constraints of v is compatible with the target memory region, I set its score $v.r$ to 0 because v's label $(v.t, v.o)$ may be wrong. It is worth noting that I do not conclude that v is absolutely wrong and remove it immediately from G. Note that a pointer in a true object may occasionally point to invalid target (or a new target that does not exist in our model). I leave it to the Random Surfer algorithm below to decide if this pointer is indeed labeled wrong.

A.4.2 Random Surfer Algorithm

I adopt the random surfer algorithm in [54] to find nearly optimal solution on the pointer-constraint graph. It is also proved to converge. Suppose $\mathbf{r}$ is a $|v|$-dimensional column vector called score vector, where $|v|$ is the number of nodes in G. $\mathbf{r}_i$ is the score of the ith node in G (for the convenience I represent vectors and matrices in bold). Besides, I define a transition matrix $\mathbf{M}$, where $\mathbf{M}_{ij}$ is the transition probability from the i^{th} node to

the j^{th} node in G. If there is no transition from i to j, $\mathbf{M}_{ij}$ is assigned 0. Because the matrix M is a stochastic matrix I normalize $\mathbf{M}$, such that each row of $\mathbf{M}$ sums to 1.

Equation A.1 describes how to calculate the scores based on the neighbors' scores:

$$\mathbf{r}^{(k+1)} = (1-\alpha-\beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{r}^{(0)} \tag{A.1}$$

where $\mathbf{p} = [\frac{\sum_i^{|v|} r_i}{|v|}]_{|v|\times 1}$ is a constant score vector, where $|v|$ is the number of node in G, and $\mathbf{r}^{(k)}$ indicates the score vector at iteration k. α is the damping factor used to jump out of isolated loops or clusters during surfing. In order to ensure that $\mathbf{r}^{(k)}$ finally converge, β is introduced as another damping factor that controls the frequency jumping to the initial score distribution $\mathbf{r}^{(0)}$. Empirically, following [54] I set $\alpha = 0.7$, $\beta = 0.1$ to guarantee a good rate of convergence.

Algorithm 2: Random Surfer algorithm

input : the transition matrix $\mathbf{M}$, the initial value vector $\mathbf{d}$, damping factor α, β and the vector constant $\mathbf{p}$
output: the converged score vector $\mathbf{r}$

$\mathbf{r}^{(0)} = \mathbf{d}$;
while $\delta \geq \epsilon$ **do**
  $\mathbf{r}^{(k+1)} = (1-\alpha-\beta)\mathbf{M}^T\mathbf{r}^{(k)} + \alpha\mathbf{p} + \beta\mathbf{d}$;
  $\delta = ||\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}||_2^2$;
end

Algorithm 2 details how I update the score of each node in the constraint graph. For each iteration, the algorithm updates the score of each node based on scores of its neighbors and the constraints among them. After several iterations, the score of each node in $\mathbf{r}$ stabilizes. For each iteration, I calculate the mean square error between the current score vector and the previous one. If the error is smaller than the threshold ϵ, I consider it to have

converged. The final score vector will approximate a globally optimal solution that satisfies the constraint graph.

Figure A.3(c) shows the converged pointer-constraint graph after I applied Algorithm 2 on the graph in Figure A.3(b). Although the object A at $0x8018$ fails the target checking at offset 8, its score is 0.99. The overall voting through the constraint graph still considers object A at $0x8018$ is more likely to be true. Although the offset constraints for object A at $0x8048$ show it could be A, the converged constraint graph tells us the score for A at $0x8048$ is 0.04. Therefore, $0x8048$ is impossible to be the base address for object A. This observation verifies that the label decision of an object type judges on overall situation rather the individual pointer constraint.

A.4.3 Kernel Object Labeling

The kernel object labeling utilizes k-means [135] method where $k = 2$ in our scenario to cluster base nodes in the constraint graph. $k = 2$ means that I only split labeled nodes of the same type into two sets including the true set and false set. The set with higher scores as true set means that all nodes in the set are correctly identified. In detail, the kernel object labeling clusters base nodes of same object type by their scores and generates the identified kernel object graph from the cluster with the higher score. For example, k-means splits (0x8018,(A,0),0.99) and (0x8048,(A,0),0.04) into two sets and I considers (0x8018,(A,0),0.99) as the true set. As for the example in Figure A.3(d), I are able to classify the base nodes in the converged graph, and construct a kernel object graph. From

the result, we can see that MACE can still find object A at $0x8018$, even if the constraint checking for the offset 8 at object A failed.

A.5 Implementation and Evaluation

The implementation of MACE comprises 3 components. One component performs an initial scan to recognize pointers on a memory image, which include one plugin (with 570 lines of Python code) to Volatility and one (with 78 lines of c code) to DECAF [4]. Another component is the plugin of DECAF with additional 800 lines of C code to gather the ground truth for kernel objects. The third component is a stand-alone Python program consisting of 6.2K LOC used for learning the pointer-constraint model and kernel object identification.

I evaluated MACE from the following aspects: Section A.5.1 presents the model generation results, including how fast the model converges, how big the model is, and how long it takes to generate the model; Section A.5.2 measures the accuracy and runtime performance of kernel object identification; Section A.5.3 demonstrates MACE's capability of detecting rootkit footprints using realworld rootkit samples; Finally, Section A.5.4 presents synthetic attacks demonstrating the attack tolerance of MACE over memory analysis tools.

Experiment Setup I evaluate MACE on four sets of memory images: 1) 150 memory images from Windows XP Service Pack 3 including 100 images for training and 50 for detection (All these images are of 512 MB RAM); 2) 145 memory images from Windows 7 Service Pack 0 including 100 images for training and 45 for detection (All these images are of 1.5 GB RAM); 3) 8 memory images of 512 MB RAM from kernel malware

analysis; and 4) 1 memory image of 1.5 GB RAM for synthetic attack. The first 295 memory images are derived by using our dynamic analysis component (as discussed in Section A.3.1). All experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2650 ($2.00GHz \times 8$) and 128 GB RAM running 64-bit Ubuntu 11.04.

A.5.1 Model Generation

Model Convergence I generate a model from 100 memory images for Windows XP and Windows 7 respectively. In order to predict how close this model is to a theoretically perfect model, I randomly select k images to generate another model, and compute a "diff" between these two models with respect to their offset constraints and target constraints. In this way, we can see how quickly the model converges when number of training images increases.

Figure A.4 shows the model quality evaluation results for both Windows XP and Windows 7. It illustrates that as the number of images in the training set increases, the missing constraints of the model generated from k images (compared to the model generated from 100 images) decreases exponentially and becomes stable very quickly. Even a model generated from one image is 99.4% similar to the model generated from 100 images.

Model Generation Runtime Obtaining a labeled memory image took nearly 2 minutes for Windows XP and 3 minutes for Window 7. To speed up the image retrieving process, I run 10 virtual machines in parallel and each one conducts 10 different test cases. Finally, it takes 20 minutes on average to obtain 100 Windows XP images and 30 minutes on 100 Windows 7 images. For each image, the model generator extracts the pointer infor-

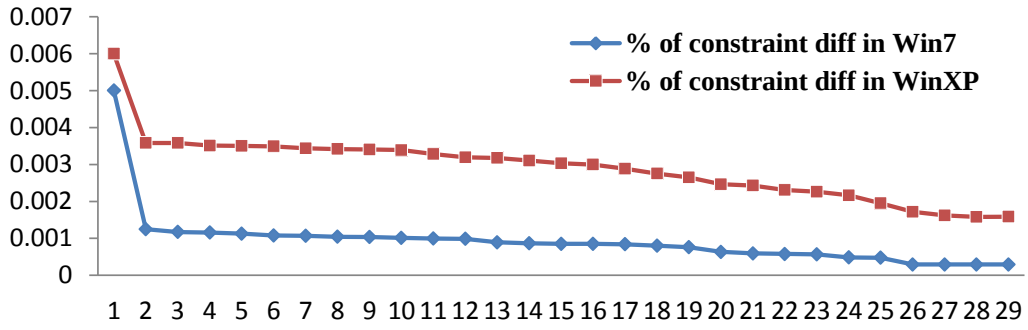
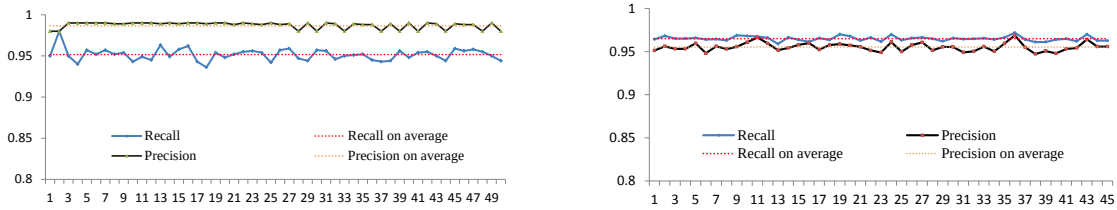


Fig. A.4.: Changes in the model quality 0 (number of new target constraints + new offset constraints) across images. The small number of images can achieve a stable model.



(a) Windows XP SP3 (b) Windows 7 SP0

Fig. A.5.: Precision and Recall.

mation from the memory image. It takes approximately 7 minutes per memory image for Windows XP and 15 minutes for Windows 7. This task can be finished within 40 and 80 minutes respectively for XP and Windows 7 by using 20 processors in parallel. In the end, it takes 20 minutes and 30 minutes to merge the results on 100 images and construct the pointer-constraint model for Windows XP and Windows 7 respectively. Overall, the model generation takes less than 2 and 3 hours for Windows XP and Windows 7 respectively.

A.5.2 Kernel Object Identification

I evaluate MACE's identification capabilities with respect to the accuracy and the runtime performance.

Accuracy I use the two metrics, *Recall* and *Precision*, to measure the accuracy of the detection results. I calculate the recall and the precision using the following formulas:

$$Recall = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ in\ ground\ truth} \tag{A.2}$$

$$Precision = \frac{Correctly\ labeled\ bytes}{Total\ bytes\ labelled\ by\ MACE} \tag{A.3}$$

I measured the above two metrics over 45 memory images for Windows XP and 7 respectively. Figure A.5 shows MACE can achieve good identification results for both Windows XP SP3 and Windows 7 SP0. More specifically, MACE achieves the 95% recall on average and 98% precision on average for Windows XP SP3, and the 96% recall and 95% precision on average for Windows 7 SP0. The detection result of MACE is close to KOP [30] which relies on the source code. Furthermore, I observed zero false negatives and false positives in the kernel objects of high forensic values (the ones extracted by Volatility). 5% false negatives are from the undocumented objects, such as the objects with pool tags 'IoNm' and 'GH0<'. 2% false positives are caused by the undocumented kernel objects of small sizes, such as 'Mmpv'.

Runtime Performance I evaluated MACE's identification runtime performance in two scenarios. In the cloud computing scenario, the virtual machine state has been loaded in memory, so scanning through the virtual machine memory is very fast. I used virtual machine snapshots (50 Windows XP SP3 memory images with 512 MB RAM and 45 Windows 7 SP0 images with 1.5 GB RAM) from KVM/QEMU to evaluate this scenario. On the contrary, in the memory forensics scenario, the memory content is first dumped into

Steps	Time (Sec)	
	Windows XP	**Windows 7**
Initial Scan	3.0 ± 2.1	7.0 ± 4.9
Graph Generation	180 ± 2.0	315 ± 5.6
Kernel Object Inference	22 ± 0.7	55 ± 1.6
TOTAL	205 ± 4.5	377 ± 12.6

Table A.1: MACE's Identification Runtime Performance

a file, then the forensic analysis is performed on the file. The analysis for this scenario will be slower, due to the time for loading the file into memory and other factors. The result for the first scenario is shown in Table A.1.

As we see, MACE finished the kernel object identification in 205 seconds for Windows XP and 377 seconds for Windows 7 on average. The identification for Windows 7 takes longer, as the memory images for Windows 7 are larger and contain more kernel objects. Note that our current implementation is in Python mainly for fast prototyping. The identification performance can be significantly reduced to tens of seconds for a C/C++ implementation.

A.5.3 Detecting Kernel Rootkit Footprints

As a case study, I show how to use the kernel object graph constructed by MACE to detect kernel rootkit footprints. To this end, I developed a tool to analyze memory images infected with kernel rootkits. Using the kernel object graph constructed by MACE, the tool can detect malicious function pointers and hidden objects in the infected images. For the sake of fair comparison, this tool follows the similar logic as SFPD and GHOST that were built on top of KOP [30]. Note that KOP requires the source code of Microsoft Windows

Name	Malicious Location	#	Cat.
Backdoor: W32TDSS	**ntoskrnl.exe:0x7c484**	1	M
	ntoskrnl.exe:0x7c480	1	M
	_GENERIC_CALLBACK.Callback	2	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
	_LDR_DATA_TABLE_ENTRY	1	H
stuxnet.vmem	_GENERIC_CALLBACK.Callback	1	M
	_NOTIFICATION_PACKET. NotificationRoutine	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.MajorFunction[]	3	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Trojan-Spy.Win32. Fakeuinit.a	**NDpp:0x18**	1	M
	NDmo:0x38	1	M
	NDmo:0x50	1	M
	NDmo:0x40	1	M
	NDpb:0x4c	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_ETHREAD.StartAddress	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Backdoor. Win32. ZAccess.dl	_DRIVER_OBJECT.DriverInit	1	M
	_GENERIC_CALLBACK.Callback	1	M
	_ETHREAD.StartAddress	1	M
TrojanPSW. Win32.Papras	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
ds_fuzz_hidden_proc.img	_EPROCESS	7	H
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.MajorFunction[]	4	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
Win32. Haxdoor	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.MajorFunction[]	2	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
RootKit: Futo	_DRIVER_OBJECT.DriverStart	1	M
	_DRIVER_OBJECT.DriverInit	1	M
	_DRIVER_OBJECT.DriverUnload	1	M
	_DRIVER_OBJECT.MajorFunction[]	4	M
	_LDR_DATA_TABLE_ENTRY.EntryPoint	1	M
	_EPROCESS	1	H

Table A.2: **Rootkit Footprints Detected By** MACE. In the column of "Category", **M** means "malicious function pointer", and **H** stands for "hidden object".

to construct the extended type graph and traverse the kernel objects, whereas MACE does not. From the viewpoint of external memory analysts, I would like to see whether the tool built on top of MACE can reach the same detection performance as these built on KOP.

More specifically, to detect malicious function pointers, our tool iterates through all the kernel objects and examine the function pointers in them. MACE can differentiate function

pointers from data pointers, because the target of a function pointer must be located in the text section of a kernel module, which can be determined from parsing the headers of that module. As the pointer-constraint model contains all the valid targets for each function pointer during the training phase, to determine a malicious function pointer, I can check whether the actual target of this function pointer does not belong to any of the valid targets. To be consistent to SFPD, our tool also excludes manipulations in System Service Dispatch Tables (SSDTs) and Interrupt Descriptor Table (IDT).

To detect hidden objects, I use Volatility as the reference system. In particular, I use common commands in Volatility, such as `pslist`. Then I compare these objects obtained from Volatility and the kernel object graph from MACE. If a kernel object of one of the above types appears only in the result from MACE, it is deemed a hidden object.

I collected 8 memory images infected with various real world kernel rootkits. Two images (Stuxnet.vmem and ds_fuzz_hidden _proc.img) were downloaded from the Volatility Google Code website. The rest of memory images (including TDSS, Fakeuinit, ZeroAccess, Papras, Haxdoor, and FuTo) were recorded by running these samples separately in a virtual machine. It demonstrated that MACE can tolerate small changes in the kernel code and the locations of global pointers, making it practical to analyze realworld memory images. I list rootkit detection results in Table A.2.

Malicious Function Pointers It is not surprising that our tool built on MACE can detect malicious function pointers in the common data structures like `_DRIVER_OBJECT`, etc. The other rootkit detection tools would have the same coverage, assuming that they can identify these data structures correctly. More interesting results are found for TDSS and

Fakeuinit, and they are highlighted in the table. For TDSS, I found two malicious function pointers located in the data section of "ntoskrnl.exe". With help of IDA Pro [75], I confirmed that tampering with these two function pointers can effectively hook `IofCompleteRequest` and `IofCallDriver` and thus manipulate the communication between the main kernel and the device drivers. As for Fakeuinit, I found 5 malicious function pointers in undocumented kernel objects, whose pool tags are 'NDpp', 'NDpb and 'NDmo' respectively. Through manual investigation, I determined that these kernel objects are operated by `NDIS.sys` (the central networking module in Windows) to manage the network stack. Manipulating these function pointers can effectively intercept the network communication. Here, note that these pool tags identified from our model are not from the infected memory images. So even if the actual pool tags are modified by the rootkits, our detection results would stay unaffected.

Hidden Objects Our tool also detected hidden processes and modules for several kernel rootkits. It shows that instead of just examining several known linked lists and tables, MACE discovers kernel objects in a global scope. I notice that a recently developed tool (`psxview`) in Volatility can detect hidden processes, by checking other data structures in addition to the active process linked list. In comparison, our tool checks `_EPROCESS` objects in the entire kernel data structure graph, not only the ones publicly known to the memory analysts, not to mention that our tool can also detect other kinds of hidden objects.

A.5.4 Attack Tolerance

To evaluate the attack tolerance of MACE, I devised two synthetic attacks: 1) pool tag manipulation; and 2) deterministic pointer removal.

Pool Tag Manipulation This synthetic attack is as simple as modifying pool tags for the objects like `_KDBG`, `_EPROCESS`, etc. After these modifications, the Windows system continued to run properly, indicating that there is no integrity check on pool tags in Windows. Then I tested the commands in volatility, and none of them output any results. The commands like `psscan` and `thrdscan` rely on pool tag as a constraint to scan particular kinds of kernel objects, so simple modifications on pool tags can easily sabotage these commands. Other commands like `pslist` and `threads` also failed, even though they did not use pool tag as a constraint extensively to scan kernel objects. The failure of these commands is due to the missing `_KDBG`. These commands must scan `_KDBG` to determine the right Windows version and thus locate the right start address of the relevant data structures. For the same reason, the new `psxview` command also failed. In contrast, MACE was not affected by this synthetic attack at all, because by design MACE does not use pool tags and other kinds of soft constraints to identify kernel objects.

Deterministic Pointer Removal I suppose that an attacker can manage to remove a fraction of pointers to hide certain kernel objects, without causing a system crash. In particular, I would like to see how MACE's identification performance degrades while a fraction of deterministic pointers are removed, because all the existing tools only examine deterministic pointers. Furthermore, I simulate a "strawman" system as the theoretical upper-bound

for any memory analysis system that only examines deterministic pointers. This "straw-man" system starts with global variables in the data sections of kernel modules, and only follows deterministic pointers and always makes a right decision whether an object is valid even when some of its pointers or pointer targets are invalid.

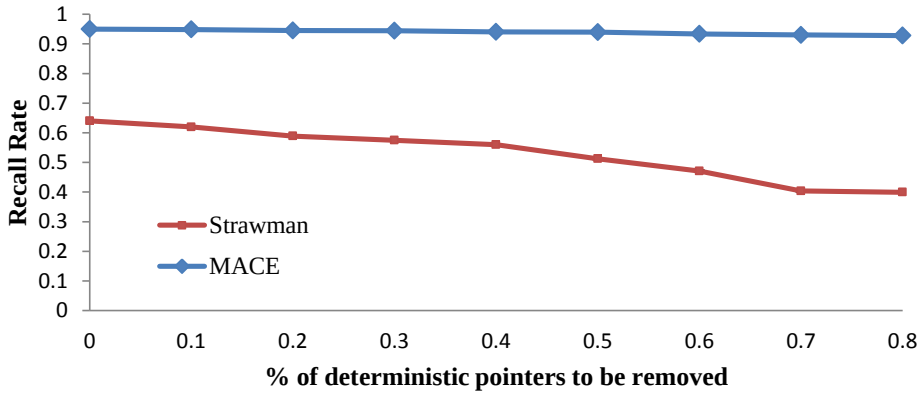


Fig. A.6.: Recall Degradation on Link Sabotage Attacks

To evaluate this attack, I randomly remove a fraction of deterministic links (e.g, 10%, 20%, 30%, etc.) from a labeled memory image, and compute the recall for both MACE and the strawman system. For the strawman system, I consider it would miss a kernel object if there is no deterministic path from a kernel module to it. Figure A.6 presents this result for a Windows XP image. It shows how the recall degrades more or less with the increase of the percentage of sabotaged deterministic pointers.

We can see that even when the attack is absent (0% pointers are removed), the recall for the strawman system is only 65%, demonstrating the necessity of incorporating non-deterministic pointers into the analysis. The performance of the strawman system degrades to about 40% when 80% deterministic pointers are removed. This result appears to be reasonable. However, this is just a theoretical upper-bound. The real memory analysis sys-

tems that only follow deterministic pointers will certainly perform worse than it. Further, the strawman system failed to identify many important kernel objects. For example, out of 22 process objects, it missed 20.

In contrast, the recall degradation for MACE is barely noticeable even when 80% deterministic pointers are removed, thanks to the small-world effect of the kernel object graph and the global evaluation nature of random surfer model.

A.6 Discussion

In this section, I discuss several potential and practical issues and concerns related to MACE. Also, I make clarifications and suggest countermeasures if necessary.

Kernel Patches By design, we need to train one model for each OS version. Kernel patches introduce changes in the main kernel module and certain data structure definitions. We need to train a new model for every single kernel patch. In reality, it is not necessary, because the changes introduced in these patches are usually small. The major kernel data structure definitions remain unchanged. As demonstrated in our experiment, we can still use the model generated for Windows XP Service Pack 3 to analyze two memory images downloaded from the Volatility website and obtain good results. In these two memory images, patches have been applied to Windows XP Service Pack 3.

Third-party Device Drivers A memory image under the analysis may have third-party device drivers loaded, which have not been observed during model generation phase. In this case, MACE will not be able to identify kernel objects defined in these device drivers.

However, MACE will still detect these device drivers, because a number of documented objects (e.g., `DEVICE_OBJECT`) will be created for them and they will be detected by MACE. It is still an open research problem to discover data structures in these third-party device drivers. I leave it as future work.

A.7 Related Work

Memory Analysis Frameworks Memory analysis came into limelight after 2004 work by Carrier et al. [31]. There exist plenty of open-source and commodity memory analysis tools [2, 22, 32, 44, 45, 88, 98, 100, 111, 122, 123, 142].The memory analysis is also extended to the analysis of hypervisors and virtual machine [68].

Robust Signature Schemes To improve robustness for memory analysis [114], two signature schemes [46, 94] have been proposed. These two signature schemes detect kernel objects by relying on invariants that are hard to be manipulated and evaded. Although the work by Dolan-Gavitt et al. [46] is based on data invariants, most of the identified invariants are indeed on pointer fields. In comparison, MACE leverages the insights in pointer invariants and takes a fresh look into the problem of memory analysis.

Source Code based Memory Analysis The knowledge of data structure definitions can be directly obtained from the kernel source code. SigGraph [94] extracts points-to relationships directly from the Linux source code and creates pointer-based signatures. SigGraph only extracts deterministic points-to relationships, and omits generic pointers. In order to obtain a nearly complete data structure graph, KOP [30] and MAS [40] perform points-

to analysis on the Windows kernel source code. They identify points-to relationships for generic pointers and generate extended type graph. In contrast, MACE is designed for external forensic analysts who often do not have access to the kernel source code of an investigated system.

Probabilistic Memory Analysis Several systems also take probabilistic approaches in memory analysis. Laika [39] applies Bayesian unsupervised machine learning algorithm to infer a type graph from a memory snapshot of a user-level program execution. In comparison, the inference algorithm used in MACE is supervised learning. In the training set, the kernel objects are classified and labeled using dynamic analysis. Some assumptions made in Laika do not hold in kernel data structures. For example, Laika assumes a pointer should point to the beginning of an object. This is not true for kernel data structures in common OSes like Windows and Linux.

To identify data structure instances that have been freed, DIMSUM [95] takes a probabilistic inference approach. Given a data structure definition, DIMSUM constructs a factor graph and computes marginal probabilities of all the candidate memory locations that satisfy the data structure constraints. In comparison, MACE tackles a similar problem but in a larger scale. The computational overhead would be too high to compute marginal probabilities for all pointers.

A.8 Summary

In this chapter, I presented MACE, a memory kernel object mining tool that can accurately identify kernel objects in a robust manner. I evaluated MACE on 100 memory

images for Windows XP SP3 and Windows 7 SP0. The experimental results showed that MACE can achieve the recall of 95% and the precision of 98% on average. Furthermore, the experiment also demonstrates the the robustness and the good efficiency. To illustrate the strength of MACE, I also conducted synthetic attacks on a memory image from Window XP SP3. The detection result showed that MACE outperformed other external memory analysis tools with respect to wider coverage and better robustness.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[2] Volatility: Memory Forencis System. https://www.volatilesystems.com/default/volatility/.

[3] DDWRT ftp. http://download1.dd-wrt.com/dd-wrtv2/downloads/others/eko/BrainSlayer-V24-preSP2/.

[4] DECAF: Binary Analysis Platform. Sycurelab, Syracuse University. http://code.google.com/p/decaf-platform/.

[5] The LLVM Compiler Infrastructure. http://llvm.org/.

[6] LMBench – Tools for Performance Analysis. http://www.bitmover.com/lmbench.

[7] mongodb. https://www.mongodb.com.

[8] Nearpy. https://pypi.python.org/pypi/NearPy.

[9] Insight-VMI, A semantic bridge for virtual machine introspection and forensic applications. https://code.google.com/p/insight-vmi/wiki/LinuxDebugSymbols.

[10] Linux memory forensics using Volatility – Prerequisites. https://code.google.com/p/volatility/wiki/LinuxMemoryForensics.

[11] z3. https://z3.codeplex.com/, 2010.

[12] DD-WRT Firmware Image r21676. ftp://ftp.dd-wrt.com/others/eko/BrainSlayer-V24-preSP2/2013/05-27-2013-r21676/senao-eoc5610/linux.bin, 2013.

[13] ReadyNAS Firmware Image v6.1.6. http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip, 2013.

[14] Retargetable decompiler. https://retdec.com, 2013.

[15] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM Commun.*, 51, 2008.

[16] R. Arandjelovic and A. Zisserman. All about vlad. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1578–1585, 2013.

[17] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[18] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[19] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.

[20] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86a platform for analyzing x86 executables. In R. Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer Berlin Heidelberg, 2005.

[21] M.-F. Balcan, A. Blum, and A. Gupta. Approximate clustering without the approximation. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1068–1077, 2009.

[22] N. Beebe. Digital forensic research: The good, the bad and the unaddressed. In *Advances in Digital Forensics V*. 2009.

[23] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, 2012.

[24] M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4, 2013.

[25] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *2006 IEEE Symposium on Security and Privacy*, pages 15–pp. IEEE, 2006.

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

[27] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3):255–259, 1998.

[28] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *IEEE Symposium on Security and Privacy*, 2011.

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

[30] 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 Communication Security (CCS'09)*, 2009.

[31] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[32] A. Case, L. Marziale, and G. G. Richard, III. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[33] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Oakland*, 2015.

[34] K. Chatfield, V. S. Lempitsky, A. Vedaldi, and A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *BMVC*, volume 2, page 8, 2011.

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

[36] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[37] P. Comparetti, G. Salvaneschi, C. Kolbitsch, C. Kruegel, E. Kirda, and S. Zanero. Identifying dormant functionality in malware programs. In *IEEE Symposium on Security and Privacy*, 2010.

[38] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[39] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[40] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of USENIX Security Symposium*, 2012.

[41] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[42] A. DINABURG and A. RUEF. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[43] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security(CCS'08)*, pages 51–62. ACM, 2008.

[44] B. Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digital Investigation*, 4:62–64, 2007.

[45] B. Dolan-Gavitt. Forensic analysis of the windows registry in memory. *Digital Investigation*, 5:S26–S32, 2008.

[46] 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(CCS'09)*, 2009.

[47] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. *2012 IEEE Symposium on Security and Privacy*, 0, 2011.

[48] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[49] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850. ACM, 2013.

[50] T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *Proceeding of CanSecWest*, 2009.

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

[52] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security*, 2014.

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

[54] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM Journal on Scientific Computing*, 27(4):1181–1201, 2006.

[55] Q. Feng, A. Prakash, H. Yin, and Z. Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. Technical Report SYR-EECS-2014-05, Syracuse University, 2014.

[56] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *ASIACCS*, 2016.

[57] 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*, 2016.

[58] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, C. Carmony, and H. Yin. Extracting conditional formulas for cross-platform bug search. In *ASIACCS*, 2017.

[59] H. Flake. Structural comparison of executable objects. In *DIMVA*, volume 46, 2004.

[60] fu. FU Rootkit. `http://www.rootkit.com~/project.php?id=12`, 2005.

[61] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, 2012.

[62] Y. Fu, Z. Lin, and D. Brumley. Automatically deriving pointer reference expressions from executions for memory dump analysis. In *Proceedings of the 2015 ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE'15)*, 2015.

[63] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[64] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. 2008.

[65] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium(NDSS'03)*, 2003.

[66] A. M. Geoffrion. *Lagrangean relaxation for integer programming*. Springer, 1974.

[67] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 2008.

[68] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor memory forensics. In *Proceedings of Symposium on Research in Attacks, Intrusion, and Defenses (RAID'13)*, 2013.

[69] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 5, 2012.

[70] T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4): 784–796, 2003.

[71] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review*, 42(3):74–82, 2008.

[72] A. Henderson, A. Prakash, L. K. Yan, et al. make it work, make it right, make it fast. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.

[73] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[74] ida-decompiler. ida-decompiler. `https://github.com/EiNSTeiN-/ida-decompiler/tree/3bd9ea6a1c073e68fef33e3cf092a34ca7fdd763`.

[75] idapro. The IDA Pro Disassembler and Debugger. `http://www.datarescue.com/idabase/`.

[76] A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *NASA Conference Publication*, pages 137–152. Citeseer, 1997.

[77] J. Jang. *Scaling Software Security Analysis to Millions of Malicious Programs and Billions of Lines of Code*. PhD thesis, CARNEGIE MELLON UNIVERSITY, 2013.

[78] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320, 2011.

[79] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Oakland*, 2012.

[80] J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX conference on Security(USENIX'13)*, pages 81–96. USENIX Association, 2013.

[81] R. Jhala and R. Majumdar. Path slicing. In *ACM SIGPLAN Notices*, volume 40, pages 38–47. ACM, 2005.

[82] L. Jiang, W. Tong, and A. G. Meng, Deyu andHauptmann. Towards efficient learning of optimal spatial bag-of-words representations. In *ICMR*, 2014.

[83] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, 2007.

[84] 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.

[85] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[86] 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*, 2013.

[87] G. Kondrak. N-gram similarity and distance. In *String Processing and Information Retrieval*, pages 115–126. Springer, 2005.

[88] J. D. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4(1):24–29, 2007.

[89] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[90] H. W. Kuhn. The hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008*, pages 29–47. Springer, 2010.

[91] 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.

[92] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, volume 4, pages 289–302, 2004.

[93] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[94] 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 *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[95] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th ISOC Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[96] McCabe. More Complex = Less Secure. Miss a Test Path and You Could Get Hacked. `http://www.mccabe.com/sqe/books.htm`, 2012.

[97] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *the workshop on learning for text categorization*, 1998.

[98] memoryze. MANDIANT Memoryze. `http://www.mandiant.com/resources/download/memoryze`.

[99] J. Ming, M. Pan, and D. Gao. ibinhunt: binary hunting with inter-procedural control flow. In *Information Security and Cryptology*, pages 92–109. Springer, 2012.

[100] S. Mrdovic, A. Huseinovic, and E. Zajko. Combining static and live digital forensic analysis in virtual environment. In *Proceedings of XXII International Symposium on Information, Communication and Automation Technologies, 2009.*

[101] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

[102] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[103] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[104] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[105] M. Newman. *Networks: an introduction*. 2010.

[106] A. Y. Ng, M. I. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 2:849–856, 2002.

[107] B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC'13), 2013 IEEE 37th Annual*, pages 492–501, July 2013.

[108] B. D. Payne, M. De Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference(ACSAC'07)*, pages 385–397. IEEE, 2007.

[109] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy(Oakland'08)*, pages 233–247. IEEE, 2008.

[110] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *CCS*, 2015.

[111] N. L. Petroni, Jr., 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.

[112] 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, 2014.

[113] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Oakland*, 2015.

[114] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[115] G. Qian, S. Sural, Y. Gu, and S. Pramanik. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the symposium on Applied computing*, pages 1232–1237, 2004.

[116] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[117] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., 2015.

[118] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *USENIX Security*, 2014.

[119] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and vision computing*, 27(7):950–959, 2009.

[120] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*, pages 1–12. Springer, 2007.

[121] 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 22th ACM SIGSAC Conference on Computer and Communications Security(CCS'15)*, pages 146–157. ACM, 2015.

[122] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3:10–16, 2006.

[123] A. Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64, 2008.

[124] 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*, page 16, 2013.

[125] M. Shahrokh Esfahani. Effect of separate sampling on classification accuracy. *Bioinformatics*, 30:242–250, 2014.

[126] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, volume 48, pages 391–406. ACM, 2013.

[127] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, 2015.

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

[129] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine, IEEE*, 25(2):128–131, 2008.

[130] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[131] 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 *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, 2008.

[132] N. Stephens, J. Grosen, C. Salls, A. Dutcher, and R. Wang. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[133] S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[134] H. A. Taha. *Integer programming: theory, applications, and computations*. Academic Press, 2014.

[135] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained k-means clustering with background knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML'01)*, 2001.

[136] M. Wall. Galib: A c++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 87:54, 1996.

[137] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[138] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Oakland*, 2015.

[139] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. 2015.

[140] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo. Evaluating bag-of-visual-words representations in scene classification. In *International workshop on Workshop on multimedia information retrieval*, 2007.

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

[142] R. Zhang, L. Wang, and S. Zhang. Windows memory analysis based on kpcr. In *Proceedings of the Fifth International Conference on Information Assurance and Security(IAS'09)*, 2009.

VITA

VITA

Qian Feng was born in Shaan Xi, China. She received her Bachelor of Science degree in Software Engineering at Xian Jiaotong University (Xi'an , Shaan Xi, China). She received her Masters of Science degree from Xi'an Jiaotong University (Xi'an , Shaan Xi, China). She received her PhD in Electrical and Computer Engineering from Syracuse University (Syracuse, New York, USA) in February 2017.