박사학위논문
Ph.D. Dissertation

휴리스틱과 바이너리 코드 유사도에 기반한 대규모 IoT 기기 취약점 분석 방법

Improving Large-Scale Vulnerability Analysis of IoT Devices with Heuristics and Binary Code Similarity

2022

김동관 (金東寬 Kim, Dongkwan)

한국과학기술원

Korea Advanced Institute of Science and Technology

박사학위논문

휴리스틱과 바이너리 코드 유사도에 기반한 대규모 IoT 기기 취약점 분석 방법

2022

김동관

한국과학기술원

전기및전자공학부

휴리스틱과 바이너리 코드 유사도에 기반한 대규모 IoT 기기 취약점 분석 방법

김 동 관

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 12월 7일

심사위원장 김 용 대 (인)

심 사 위 원 차 상 길 (인)

심 사 위 원 손 수 엘 (인)

심 사 위 원 유 신 (인)

심 사 위 원 윤 인 수 (인)

Improving Large-Scale Vulnerability Analysis of IoT Devices with Heuristics and Binary Code Similarity

Dongkwan Kim

Advisor: Yongdae Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering

Daejeon, Korea
December 21, 2021

Approved by

Yongdae Kim
Professor of Electrical Engineering

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DEE 김동관. 휴리스틱과 바이너리 코드 유사도에 기반한 대규모 IoT 기기 취약점 분석 방법. 전기및전자공학부 . 2022년. 106+vi 쪽. 지도교수: 김용대.
Dongkwan Kim. Improving Large-Scale Vulnerability Analysis of IoT Devices with Heuristics and Binary Code Similarity. School of Electrical Engineering . 2022. 106+vi pages. Advisor: Yongdae Kim.

초 록

수많은 IoT 기기의 보안성 향상을 위해 대규모 취약점 분석 기술은 필수적이다. 하지만 IoT 기기는 그 하드웨어와 실제 구현 및 실행환경이 매우 다양하고, 개발 표준이 없이 폐쇄적으로 개발되며 세부 정보가 공개되지 않아, 확장성 있는 분석 기술을 개발하기가 쉽지 않다. 대규모 분석을 위해 실제 기기 없이 기기 펌웨어를 분석하는 연구가 다수 존재해왔으나, 간단한 소수의 기기만을 대상으로 삼거나 분석 성공률 또한 미비하였다. 본 연구는 확장성 있는 IoT 기기 취약점 분석 기술을 제시한다. 이를 위해 다양한 기기의 펌웨어를 직접 분석하여 펌웨어 에뮬레이션 및 펌웨어 구조 분석을 위한 휴리스틱을 개발한다. 그 결과 많은 기기들이 유사한 코드를 공유한다는 점을 발견하였으며, 개발된 휴리스틱을 활용해 95개의 최신 무선공유기와 IP 카메라로부터 23개의 0-day와 주요 스마트폰 베이스밴드로부터 3개의 0-day를 발견하였다. 또한 기기의 코드 유사도를 바탕으로 더욱 확장성 있는 취약점 분석 기술을 제시한다. 현재 대규모 분석을 수행할 수 있는 바이너리 코드 유사도 분석 시스템은 존재하지 않기 때문에, 우선 현존하는 유사도 분석 기술을 체계화하고 이를 바탕으로 대규모 유사도 분석 시스템을 구축한다. 이를 활용하여 앞서 분석한 펌웨어들에서 442개의 취약점을 발견하였다. 마지막으로 본 연구에서 사용한 데이터와 코드를 공개하여 IoT 생태계 보안성 향상에 기여한다.

핵 심 낱 말 IoT 보안, IoT 취약점 분석, IoT, 취약점 분석, 바이너리 코드 유사도, IoT 기기 분석, 임베디드 기기

Abstract

To secure numerous Internet of Things (IoT) devices globally, conducting a large-scale vulnerability analysis is essential. However, developing a scalable analysis approach that is applicable to various devices is not straightforward because 1) IoT devices have a wide variety of hardware configurations, implementations, and execution environments, and 2) their vendors often withhold information about their products. To address the scalability issue, several studies have attempted to analyze device firmware rather than physical devices. However, these approaches are currently limited to a few simple/small devices, resulting in low analysis success rates.

In this thesis, we present a practical approach towards scalable vulnerability analysis of IoT devices. We began by conducting an empirical analysis of various IoT devices and discovered that many of them share a common codebase. We leveraged this similarity to develop several heuristics that enable successful firmware emulation and firmware structure analysis, which are essential for vulnerability analysis. Using these heuristics, we discovered 23 0-day vulnerabilities in wireless routers and IP cameras, as well as three 0-days in smartphone baseband devices.

Following that, we present another approach that extends the vulnerability analysis by utilizing binary code similarity analysis (BCSA). There have been several BCSA approaches, but none are easily applicable because they often 1) do not share their source code or datasets and 2) employ uninterpretable machine learning techniques that make the results difficult to comprehend. To address this, we first conducted a comprehensive study of existing BCSA techniques, which revealed several insights. For instance, a simple model with a few basic features can achieve results comparable to those obtained using deep learning techniques. Based on the findings, we developed a BCSA framework and two heuristic features. We demonstrated our system's effectiveness by analyzing over 53M functions in 1,142 IoT firmware images and successfully identifying 442 vulnerabilities. We make our source code and datasets publicly available to encourage further research.

Keywords IoT Security, IoT Vulnerability Analysis, IoT, Vulnerability Analysis, Binary Code Similarity, IoT Analysis, Embedded Device

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Motivation

We are surrounded by billions of Internet of Things (IoT) devices, from smart speakers to internet-connected power outlets and light bulbs. Not only simple and small devices are rapidly deployed worldwide, but an increasing number of complex devices, such as smartphones, self-driving cars, and unmanned aerial vehicles. Because IoT devices are always connected to the Internet, security flaws in these devices are critical for the entire Internet [1, 2]. Recent distributed denial of service (DDoS) attacks, which originate from a massive number of those devices, have demonstrated that such attacks are a real and serious threat. These attacks can generate unprecedented traffic volumes greater than 1 Tbps, which can temporarily shut down critical Internet services such as DynDNS (in 2016) [3] or GitHub (in 2018) [4]. Additionally, numerous backdoors have been discovered on a variety of devices [5, 6], and malware, such as Mirai and Satori, has infected millions of such devices [7, 8, 9, 10].

Scaling up vulnerability analysis is critical for eradicating such threats not only in widespread devices but also in convoluted IoT ecosystems comprised of numerous manufacturers, devices, and applications. Due to the impossibility of physically obtaining numerous devices, existing studies have attempted to analyze device firmware using advanced security analysis techniques, such as symbolic execution [11, 12], fuzzing [13, 14, 15], or automated pentesting [16, 17, 18, 19, 20].

However, conducting vulnerability analysis on a large number of IoT devices remains challenging due to the opacity (*i.e.*, obscurity) and diversity underlying the IoT ecosystem. Each IoT device is accompanied by a specific set of peripheral hardware devices from a variety of manufacturers. As a result, IoT devices have a wide range of hardware configurations and software implementation practices. Additionally, vendors frequently withhold information about their devices, obstructing in-depth analysis of them.

In order to conduct scalable vulnerability analysis, several studies have attempted to address the complexity and opacity of the IoT ecosystem. They are, however, still limited to a few simple devices. For instance, Firmadyne [17], the leading firmware emulation framework, could emulate only 183 of 1,124 (16.3%) firmware images in wireless routers and IP cameras that we collected from the top eight vendors.

What could account for such a low success rate despite the fact that many studies have been conducted? Throughout our literature review of IoT vulnerability analysis, we noticed a prominent research trend in this field: many studies place a higher value on the novelty and freshness while ignoring engineering effort and hands-on analysis when developing analysis approaches. More precisely, they disregard the value of empirical analysis of various device types. Additionally, they avoid developing heuristic workarounds that could have solved analysis problems because they appear to be neither scientific nor novel, but rather simple and trivial. In this regard, we hypothesize that the immaturity (*i.e.*, low success rate) of existing IoT vulnerability analysis approaches may be a result of a trend away from manual analysis and away from the development of heuristic workarounds.

1.2 Research Direction and Contribution

We argue in this thesis that *well-systematized heuristic workarounds* can address the opacity and diversity issues inherent in IoT vulnerability analysis. Despite their diversity, we believe that many IoT devices, particularly similar device versions or families, may share similar implementations. Thus, a small amount of additional engineering effort and hands-on analysis can address the issues in vulnerability analysis and significantly increase

the success rate of the analysis. Additionally, we could leverage the similarity of IoT devices; if we develop heuristics that meet the requirements of vulnerability analysis, the heuristics developed can be applied to other devices, enabling large-scale analysis. Therefore, we explore the following hypothesis in the paper:

> *"While heuristics may appear trivial and not technically novel, developing and systematizing 'dirty' heuristics is critical, and it is the last-mile effort required to enable large-scale vulnerability analysis on the IoT ecosystem."*

To test this hypothesis, we looked at two different device categories, namely Linux-based IoT devices and smartphone baseband chipsets. Linux-based IoT devices have a relatively straightforward functionality and a well-known firmware structure. On the other hand, smartphone baseband chipsets incorporate sophisticated functionalities such as real-time signal/protocol processing, and their firmware structure is largely unknown. We will discuss these characteristics more in detail in §2.2. We collected 1,124 firmware images from the top eight wireless router and IP camera vendors for the former category. For the latter, we gathered 18 firmware images from one of the top three smartphone baseband manufacturers.

By analyzing the collected firmware images empirically, we discovered that similar types of devices, particularly similar device versions or models, in practice share a common codebase. Based on this observation, we explore our hypothesis for three fundamental steps in IoT vulnerability analysis: (1) firmware emulation, (2) firmware structure analysis, and (3) known vulnerability analysis. We conduct empirical investigations into failures of each technique and develop heuristics to address them. Then, we 1) systematize the developed heuristics, 2) integrate them as plugins into existing approaches, and 3) quantify the degree of performance improvement.

Notably, the systematized heuristics demonstrated significant gains. To begin, they increased the success rate of the leading firmware emulation framework from 183 (16.3%) to 892 (79.4%). This enabled us to discover 306 (≈23 times) more vulnerabilities than the state of the art through 1-day exploit testing, as well as 23 new vulnerabilities affecting 95 of the latest devices through the use of a simple custom fuzzer. Second, the heuristics discovered more than 200 times the number of function boundaries in smartphone baseband firmware than the state-of-the-art binary analysis framework. Additionally, the heuristics identified our target functions responsible for decoding baseband protocol messages, resulting in the discovery of 72 functional bugs and 56 memory-related vulnerabilities.

By leveraging the similarity of IoT devices, developing and systematizing heuristics increased the scalability of vulnerability analysis. More precisely, we were able to rapidly identify previously known vulnerabilities by leveraging binary code similarity analysis (BCSA). Notably, there is no BCSA framework that is easily applicable to large-scale IoT vulnerability analysis; thus, we set out to create the first BCSA framework for IoT vulnerability analysis. To begin, we conducted a comprehensive study on existing BCSA approaches and gleaned a wealth of information. For example, proper feature engineering on simple features and models can achieve promising results that are comparable to those obtained using state-of-the-art approaches. Following that, we conducted an empirical analysis of firmware images and developed two heuristic features. By systematizing this heuristic knowledge, we developed a BCSA framework that detects vulnerability candidates in the firmware image for a given vulnerable function. Using this system, we analyzed 53,492,954 functions from 1,142 firmware images and discovered 442 vulnerabilities, of which 189 were not discovered by previous studies.

Our study uncovered common issues in the recent research on IoT vulnerability analysis and underscored the critical nature of conducting hands-on analysis and developing/systematizing heuristic workarounds. To foster further research, we make our source code and datasets publicly available.

In summary, we make the following contributions:

- We conducted empirical research into firmware emulation failures and developed heuristics for successful emulation. By systematizing the heuristics, we increased the success rate of the state-of-the-art firmware

emulation framework from 183 (16.3%) to 892 (79.4%) for 1,124 firmware images of wireless routers and IP cameras. This resulted in the discovery of 343 vulnerabilities including 23 0-day vulnerabilities.

- We conducted an empirical analysis of smartphone baseband firmware in order to decipher its structure. We developed several heuristics for determining the boundaries of firmware functions. For 18 baseband firmware images, the heuristics successfully identified over 200 times more functions than the state-of-the-art binary analysis framework. Additionally, they identified message decoder functions, enabling us to discover a total of 72 functional bugs and 56 vulnerabilities, including critical 0-day vulnerabilities.
- We extracted useful insights and effective features for BCSA through a systematic study of BCSA and empirical analysis of IoT firmware images. Based on this heuristic knowledge, we developed the first BCSA system that is easily applicable to large-scale IoT vulnerability analysis. The system analyzed 53,492,954 functions contained in 1,148 firmware images and discovered 442 vulnerabilities.
- We emphasize the critical nature of performing hands-on analysis and developing/systematizing heuristic workarounds. To encourage further research, we make our source code and datasets publicly available.

1.3 Thesis Structure

The remainder of the thesis is organized around our previous publications [21, 22, 23, 24]. Chapter 2 provides context for the IoT vulnerability analysis procedure and introduces our target devices. In Chapter 3, we describe our heuristic approach for successfully emulating the firmware of Linux-based IoT devices, such as wireless routers and IP cameras. Chapter 4 describes our heuristics for analyzing the firmware structure of major smartphone baseband devices. Following that, Chapter 5 systematizes existing BCSA techniques and our insights. In Chapter 6, we describe how we use BCSA to scale up vulnerability analysis. Finally, we conclude the thesis in Chapter 7.

Chapter 2. Background

IoT devices frequently communicate with the cloud or a mobile phone via the Internet in order to maintain control of the device or to allow the user to control the device. These IoT devices are frequently purpose-built; examples include wireless routers, IP cameras, and smart speakers. IoT devices are composed of specialized hardware peripherals and software to accomplish their intended functions. Such hardware is managed by *firmware*, which consists of a customized bootloader, operating system kernel, and filesystem, as well as programs to perform required tasks. Alternatively to multiple files, firmware can run as a single executable, enabling close collaboration between multiple functionalities, such as a real-time operating system. To conserve power and resources, IoT device firmware is frequently based on RISC architectures, such as ARM, MIPS, or PowerPC, rather than CISC architectures, such as x86 or x86-64. In summary, because IoT devices are specialized embedded computer systems, their vulnerability analysis methodologies differ slightly from those used for general desktop/server computer programs. The remainder of this chapter describes the typical IoT vulnerability analysis procedures and target device categories on which this thesis focuses.

2.1 IoT Vulnerability Analysis Procedure

Figure 2.1 is a representative illustration of a typical IoT vulnerability analysis procedure. This analysis procedure varies according to the methodologies used; in this case, we followed a recent trend in IoT vulnerability analysis. Because physically obtaining numerous devices is infeasible, researchers have attempted to analyze device firmware rather than actual devices. They create a virtual environment that resembles a real device and emulate firmware images in order to apply dynamic analysis techniques (see Figure 2.1 for the yellow parts). On the other hand, some firmware images may be impossible to emulate due to their unknown structure or their interaction with a large number of peripherals. Researchers analyze the structure of these firmware images first and then apply static analysis techniques to them (the red parts in Figure 2.1). Both approaches can also be used in conjunction with one another, which is referred to as a hybrid approach. If a vulnerability is discovered on one device, it may also exist on other devices. To quickly identify such vulnerabilities, one can extract distinct patterns from the discovered vulnerability and then search for those patterns in other firmware images, which we refer to as known vulnerability analysis. In the following, we will detail each step of the IoT vulnerability analysis procedure.

2.1.1 Acquiring Firmware Images

A firmware image can be obtained directly from a device; for example, one can dump a firmware image from a device's flash memory. However, this approach necessitates the use of a proprietary interface accessible only to manufacturers, such as a debug interface. Typically, such an interface is not made available to regular users in order to prevent unintentional firmware access. Rather than that, firmware images can be obtained from manufacturers' websites via a firmware update routine, or from third-party servers that archive previously released firmware images. To easily collect firmware images from these websites and servers, researchers frequently use automated scraping tools, such as Scrapy [25].

Firmware images are typically compressed, and thus must be unpacked prior to further analysis. A firmware image typically contains a bootloader, kernel, and filesystem, containing the device's applications. To compress the firmware image's internal contents, various compression algorithms, such as LZMA, ZIP, or Gzip, are frequently

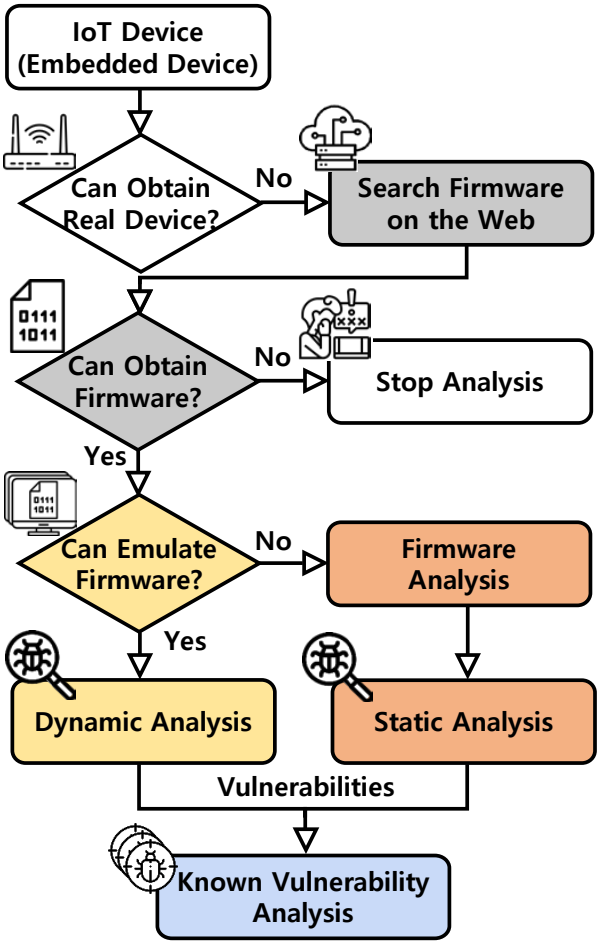


Figure 2.1: A typical example of an IoT vulnerability analysis procedure. The colored logic indicates our focus. (yellow: firmware emulation, red: firmware structure analysis, blue: known vulnerability analysis

used. To unpack firmware images, tools, such as Binwalk [26] or Firmware-Mod-Kit [27], can be used. These tools scan a given image for pre-defined signatures for a variety of file types. If they find a signature match, they extract the matching file from the image and continue scanning until all files in the image are collected. Additionally, there are encrypted or customized images that cannot be used with signature matching, and they are beyond the scope of this thesis.

2.1.2 Emulating Device Firmware and Conducting Dynamic Analysis

Once the files have been extracted from the firmware image, they can be analyzed statically or dynamically. Due to the absence of runtime information, static analysis often produces many false positives. In contrast, dynamic analysis directly executes target programs, resulting in fewer false positives. Therefore, dynamic analysis has been frequently used in IoT vulnerability analysis [28, 29, 30, 31, 32, 33, 15, 34, 35, 36].

To conduct dynamic security analysis, one must either possess a physical device or create an emulation environment in which the firmware can be run and controlled. Because the latter approach does not require any physical devices, it enables large-scale dynamic analysis, preferably using elastic cloud services. Therefore, recent studies [16, 17, 14, 13, 15, 18, 19, 20] have concentrated on the latter approach, which emulates device firmware and dynamically evaluates its security. The system that performs firmware emulation is denoted as the *host* system, and the emulated system (*i.e.*, the running firmware) is referred to as the *guest* system.

Typically, there are two levels of emulation: user-level and system-level. User-level emulation emulates only the target program in the firmware, making optimal use of the host system. An example would be emulating a web interface. A web interface is a representative service used in IoT devices to administer or maintain the device. It serves a variety of static content types, such as HTML, as well as dynamic contents generated by CGI programs. While static contents can be served with the host environment, dynamic contents may not. This is because they may cause a conflict with the host system because they rely on custom libraries and device drivers that are not included in the host system. On the other hand, system-level emulation completely emulates the guest system, including the kernel. Because it provides an isolated execution environment, it can also emulate various kernel and device driver features. Nonetheless, firmware emulation is extremely difficult, as vendor-specific hardware issues or peripherals that are memory-mapped must be considered. Without properly handling them, programs running on the emulated firmware would crash.

Several studies [16, 17, 14, 13, 15, 18, 19, 20] have attempted to overcome these obstacles by developing an emulation environment that is physically similar to that of physical devices. QEMU [37] is a widely used emulator that supports various processor/hardware types and peripherals. Costin *et al*. [16] presented a framework for scalable dynamic analysis, as well as several case studies involving various embedded web interfaces. Chen *et al*. [17] introduced Firmadyne, the first large-scale firmware emulation framework. Notably, Firmadyne emulates non-volatile random access memory (NVRAM), which is used to store various configuration values for programs running in the emulated firmware. Gustafson *et al*. [18] modeled memory-mapped I/O (MMIO) operations in peripheral communication. Feng *et al*. [19] adopted machine learning to resolve this MMIO issue; their model returns predicted values for MMIO access. Clements *et al*. [38] recently proposed using a hardware abstraction layer to decouple the hardware from the firmware. While these approaches are encouraging, they are still insufficient to cover the vast majority of IoT devices in practice. We will more discuss their limitations and how we overcame challenges in firmware emulation in Chapter 3.

Meanwhile, it is worth noting that additional research [39, 40, 41, 42, 43] has been conducted on utilizing physical devices to address emulation challenges. Zaddach *et al*. [41] and Marius *et al*. [42] proposed analyzing only a subset of firmware code by relaying the execution of other code or peripheral access to a physical device. They accomplished this by utilizing a debug interface known as JTAG. Similarly, Kammerstetter *et al*. [39, 40] created a proxy environment that redirects access to a character device to a physical device. While these approaches are advantageous, they require physical devices and thus do not scale well for IoT vulnerability analysis.

After successfully running all required firmware programs, dynamic analysis can be used to identify potential vulnerabilities in a target device. The most frequently used techniques for this type of analysis are 1) advanced fuzzing techniques, or 2) manual/automated pentesting based on previously known vulnerabilities. Existing research that took the former approach [13, 14, 15] frequently made use of a well-known fuzzer, American fuzzy lop (AFL) [44]. Notably, Zheng *et al*. [15] proposed an optimization technique for running a fuzzer effectively in an emulated environment; their approach switches the context between system- and user-level emulation. By contrast, studies that took the latter approach [17] frequently used well-known Proof-of-Concept (PoC) scripts or automated pentesting tools, such as Metasploit [33] or RouterSploit [28].

2.1.3 Analyzing Firmware Structures and Conducting Static Analysis

To perform static or dynamic analysis, one must first understand the firmware's structure. The firmware structure of Linux-based IoT devices, such as wireless routers or IP cameras, is widely known. As a result, there are many useful tools [26, 27, 45] that scan pre-defined signatures for various types of files in such devices. However, many IoT devices (*e.g*., a smartphone baseband) have a unique firmware structure that is not yet publicly known.

These devices often have a convoluted firmware structure and communicate with many peripherals. As a result, emulating the firmware of such devices and performing dynamic analysis is exceedingly difficult.

To resolve this issue, one must manually analyze the firmware structure of the device and then conduct appropriate analysis. To conduct a successful analysis of the firmware structure, one must first understand the firmware's file format, then analyze the memory layout into which the firmware is loaded, and finally, identify function boundaries from a stream of byte codes in the firmware. Nonetheless, the majority of studies [46, 47, 48, 49, 50, 51] have concentrated on identifying function boundaries (*i.e.*, the last step) without examining the other two steps. Even so, these approaches are constrained to simple IoT devices or a few architectures (mostly x86, a few ARM). We will more discuss the challenges inherent in firmware structure analysis and how we overcame them in Chapter 4.

After revealing the firmware structures of IoT devices, static analysis is frequently used to identify potential vulnerabilities [52, 12]. Costin *et al*. [52], for example, showed determined the vulnerability of devices by by analyzing easily crackable passwords or backdoor strings. Using symbolic execution, Shoshitaishvili *et al*. [12] discovered authentication bypass vulnerabilities. Due to the absence of runtime information, static analysis may generate more false positives than dynamic analysis. However, static analysis has several advantages; for example, because it does not directly execute instructions, it is largely immune to the emulation difficulties associated with dynamic analysis. Additionally, it can be applied directly to the files contained within a firmware image. As a result, static analysis is more scalable than dynamic analysis when it comes to vulnerability analysis.

2.1.4 Propagating Vulnerability Discovery with Known Vulnerability Analysis

Numerous IoT devices share similar vulnerabilities and are frequently exploited by them [7, 8, 9, 10]. This could be a result of the trend toward software reuse and code sharing. To ease the device development process, programmers often reuse existing code to create new software. Some even copy and paste code samples from the Internet. This trend has significant implications for the security and privacy of software. When a programmer copies a vulnerable function from another project, the vulnerability is retained even after the original project resolves it. Regrettably, such issues are also prevalent in the IoT ecosystem.

However, detecting known vulnerabilities in binary code is not straightforward, especially when the source code is not available. This is because binary code is devoid of high-level information, such as data types or function names. For instance, it is not immediately apparent from binary code whether a memory cell represents an integer, a string, or another data type. Furthermore, defining precise function boundaries is inherently difficult [46, 47, 48, 49, 50, 51].

As a result, researchers executed known PoC exploits or automated pentesting tools, such as Metasploit [33] or RouterSploit [28] after emulating target device firmware images. This is advantageous because it avoids the need for precise binary analysis. However, this approach may not be suitable for scalable security analysis, as it still requires successful firmware emulation, as well as sufficient time for emulation and dynamic testing.

On the other hand, several studies have focused on exploiting binary code's similarity [53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64] in order to rapidly detect known vulnerabilities. Notably, BCSA has been a critical area of research in a variety of fields, including malware detection [65, 66], plagiarism detection [67, 68], and authorship identification [69]. Recent studies on IoT vulnerability analysis [70, 54, 71, 55, 60, 57, 56] have also attempted to assess the security of numerous IoT devices using advanced BCSA techniques.

Nonetheless, applying BCSA techniques to IoT devices is not trivial due to the fact that existing studies 1) do not share their source code or datasets and 2) use uninterpretable machine learning that complicate understanding the results. In Chapter 5, we will more discuss these points and how we addressed them. In Chapter 6, we also present a scalable BCSA system for IoT security testing, as well as the results of a large-scale analysis.

2.2 Target Device Category

In this thesis, we analyze two distinct categories of IoT devices: 1) IoT devices powered by Linux, such as wireless routers and IP cameras, and 2) smartphone baseband. Malware frequently targets Linux-based IoT devices [7, 8, 9, 10] in order to launch DDoS attacks [3, 4]. On the other hand, the smartphone baseband is critical to mobile communication; it is responsible for managing all packets exchanged between the smartphone and cellular networks. Thus, their implementations may contain multiple critical vulnerabilities [72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83].

Additionally, we chose these two device categories due to their distinctive characteristics, which are detailed in Table 2.1. Active analysis of Linux-based IoT devices has resulted in widespread knowledge of their firmware structure. On the other hand, the smartphone baseband's firmware structure has not been revealed yet. We investigate investigate the importance and effectiveness of heuristics by examining each device category. That is, we 1) investigate firmware emulation issues using Linux-based IoT devices (in Chapter 3) and 2) leverage the smartphone baseband to investigate issues with firmware structure analysis (in Chapter 4). Then, utilizing both devices, we examine issues in known vulnerability analysis (in Chapter 6).

Table 2.1: Characteristics of our target devices.

	Wireless Routers and IP Cameras	Smartphone Baseband
Functionality	Simple	Complex (Real-Time)
Firmware Structure	Well-Known	Unknown
Operating System	General Purpose OS (*i.e.*, Linux)	No OS Abstraction
# of Vendors	Numerous	A Few (Oligopoly)
# of Files in Firmware	Multiple Files	Monolithic
# of Peripherals	A Few	Multiple
Emulation	Feasible	Difficult

Chapter 3. Towards Large-Scale Firmware Emulation

To assess the security of a large number of IoT devices, researchers have focused on large-scale firmware analysis. Many existing studies [16, 17, 14, 13, 15, 18, 19, 20] have taken the approach of running device firmware in an emulated environment that closely mimics the behavior of real hardware/peripherals and then performing dynamic analysis on the running firmware. This approach enables not only dynamic firmware analysis without the need for hardware, but also the use of cloud infrastructure to scale up the security analysis. Among others, Firmadyne [17] is the current state-of-the-art firmware emulation framework that enables large-scale emulation of IoT devices by providing a full-system emulation environment.

However, this approach is not foolproof in practice, as running firmware in a full-system emulation environment frequently fails due to the inconsistencies between the real and the virtual, emulated environments. Any inconsistencies in the emulated environment can cause the running firmware to enter an unexpected state, resulting in the failure of emulation and dynamic security analysis. Resolving this emulation discrepancy is difficult due to the the wide variety of IoT device hardware and configurations. Each IoT device is equipped with a unique set of hardware components from a variety of manufacturers. Additionally, firmware typically relies on configuration vectors, such as data in NVRAM, which an emulated environment may miss due to the data being only available in hardware. Such complicated circumstances are incompatible with Firmadyne's emulation environment. QEMU [37], its emulator, only supports a few common devices and configurations. Without a concerted effort to emulate each device, the issue will never be resolved.

To investigate the impact of this issue in practice, we collected 1,124 firmware images from the top eight wireless router and IP camera vendors and ran them through Firmadyne. The result is alarming as Firmadyne is only capable of emulating 183 of them (see Table 3.2), leaving the vast majority portion of firmware images (83.72%) unanalyzed. Such a low success rate for emulation implies that, while Firmadyne is intended to be generic by providing a full-system emulation environment for firmware, this approach may not be sufficient in practice, necessitating additional manual efforts to resolve emulated environment inconsistencies.

Building a precise emulation environment may not be the only way to run firmware for dynamic security analysis. To counter the hardware complexity inherent in the IoT ecosystem, we propose that *well-systematized heuristic workarounds* could be a viable alternative approach for achieving a higher success rate of firmware emulation in practice. By empirically analyzing Firmadyne's failure cases, we noticed that simple changes in device or software configurations could allow the firmware to continue to run without failures. By systematizing such heuristic workarounds and incorporating them as plugins, the system we developed, FIRMAE [21], increased the success rate of firmware emulation from 183 (16.3%) to 892 (79.4%). By running 1-day exploit testing, the system discovered 306 (≈23 times) more vulnerabilities than Firmadyne due to the increased emulation success rate. Additionally, it discovered 23 0-day vulnerabilities from 95 of the most latest devices using a simple fuzzer.

This chapter 1) discusses the novel possibilities of well-systematizing heuristic workarounds for enabling large-scale firmware emulation in practice, and 2) summarizes how we discovered and systematized such heuristic workarounds.

3.1 Preliminaries for Firmware Emulation

Figure 3.1 depicts a typical firmware emulation procedure for dynamic analysis. After unpacking a firmware image, the emulation framework boots the guest system from the extracted filesystem, which includes a bootloader

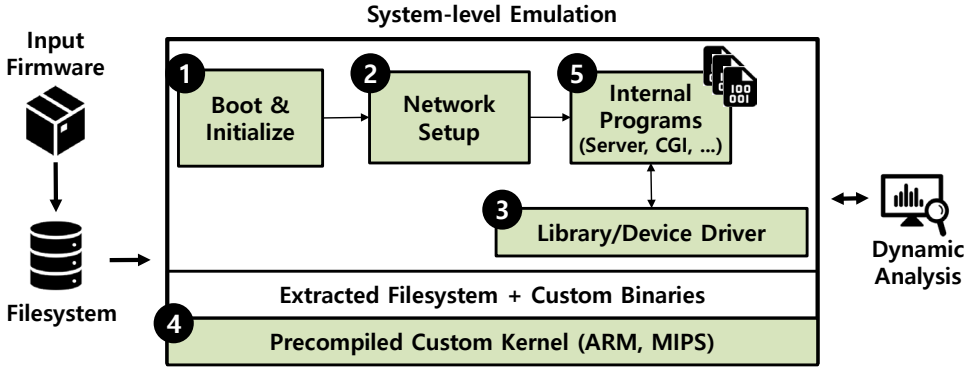


Figure 3.1: An example of typical firmware emulation procedure for dynamic analysis. The colored components involve potential emulation issues.

and kernel. Following that, the guest system performs initialization and configuration tasks for its network functionality. Finally, the guest system runs applications, such as web servers or CGI programs, that interact with the libraries or device drivers in the emulated system.

3.1.1 Challenges in Firmware Emulation

The emulation-based analysis is advantageous to conduct dynamic security analysis on elastic cloud (in terms of scalability). However, successfully emulating a variety of firmware images is exceedingly difficult due to the discrepancy between the real and emulated environments. Resolving such inconsistencies is not straightforward owing to the convoluted IoT ecosystem's wide diversity of hardware configurations and software implementations, which result from the absence of standardized software development practices. Libraries, device drivers, and even kernels vary between vendors, resulting in emulation failure. Unless these issues are resolved properly, internal programs may crash, rendering further emulation and dynamic analysis impossible.

As previously noted [18, 19], devices that access hardware interfaces, such as LED sensors or cameras, exhibit greater diversity. Memory-mapped IO (MMIO) operations are frequently used in such devices to communicate between the main device and its peripherals via pre-defined memory addresses. These addresses, however, vary significantly between devices. Consequently, scaling this approach to multiple devices is challenging. Chen *et al*. [17] attempted a large-scale emulation of one such piece of hardware, NVRAM. Muench *et al*. [84] emphasized the device-specific difficulties associated with performing a dynamic analysis to identify memory corruption vulnerabilities. However, such approaches are still too early to implement on a large scale.

Fully emulating a firmware image may not be feasible unless all functions are implemented exactly as they are in physical devices. Nonetheless, our goal is not to perfectly emulate a firmware image, but rather to perform dynamic analysis. Therefore, there may be opportunities to accomplish the goal, which we will discuss in greater detail below.

3.1.2 Firmadyne (Automatic Emulation Framework) and Its Limitations

Existing research projects approach such emulation challenges as a hardware emulation problem, *i.e*., emulating hardware and peripheral devices as precise as the real one. Among these approaches, Firmadyne [17] is the current state-of-the-art framework (which has been superseded by FirmAE [21]) for large-scale analysis of Linux-based IoT devices. Many subsequent studies [13, 14, 15] have used it to conduct dynamic analysis.

Firmadyne emulates hardware and peripheral devices by utilizing a pre-built customized Linux kernel and

libraries. For precise emulation, it emulates a target firmware image twice after unpacking the image. In the first emulation, its customized kernel with a driver intercepts major system calls and records useful information. Then, the second uses the logged information to configure the guest system properly. For example, Firmadyne records the name and IP address of the network interface that the emulated firmware accesses by hooking `inet_ioctl()` and `inet_bind()` in the first emulation. Then, it configures the guest system with this information in the second emulation. Firmadyne also makes use of pre-built custom libraries to emulate peripherals, particularly flash memory called non-volatile RAM (NVRAM). For instance, a library called `libnvram` stores and retrieves NVRAM values using hard-coded default values.

While Firmadyne's precise hardware emulation appears promising, in reality, its emulation success rate is considerably low. Among 1,124 IoT firmware images that we collected from the top eight vendors of wireless routers and IP cameras, Firmadyne can emulate only 330 of 1,124 (29.4%) firmware images for networking functionality. It has a much lower success rate when it comes to running applications (*e.g.*, web servers) in the firmware. More precisely, it is capable of emulating web applications for only 183 (16.3%) firmware images.

3.1.3 Motivating Example using Firmadyne

To understand how we can handle inconsistencies, we conducted a case study using Firmadyne on two firmware images. To begin, we used Firmadyne to run the firmware for the `D-Link DIR-505L` in order to test CVE-2014-3936 [85]. Because the vulnerability is a stack-based buffer overflow in a web service running on the firmware, exploiting it requires sending HTTP requests via the network interface of the guest system. However, when we used Firmadyne to run the firmware, we were not able to connect to the web service. Our analysis revealed that the guest system's network is not configured properly, preventing the exploitation even though the web server is running normally. The root cause may be that a process responsible for network configuration was blocked due to access to unsupported peripherals or unknown/incorrect values. We were able to test the vulnerability, however, by forcibly configuring the network to its default setting. Second, we used Firmadyne to test the firmware of the `NETGEAR R6250` for CVE-2017-5521. In this case, the emulation failed during the booting procedure due to a kernel panic. We were able to run the firmware and test the vulnerability without analyzing the root cause after slightly modifying the booting and kernel-related configuration to match the virtual environment.

From these two examples, we observed that minor changes in configuration or device settings can enable firmware emulation to run without addressing the root causes, *i.e.*, emulation discrepancy problems. In this regard, we believe that Firmadyne *may have passed up many opportunities to emulate and analyze IoT firmware images, despite the fact that its failure cases are easily handled.*

3.2 Scaling up Firmware Emulation for Security Testing

Perfect hardware compatibility at the emulation layer may not be required for dynamic software security testing, such as applying human pentesting and fuzzing. Rather than that, it may be sufficient to meet the bare minimum requirements for properly running target applications in firmware. For example, with a wireless router, we need to 1) boot the operating system of its firmware, 2) configure the network interfaces for communication, and 3) run a web server that serves as the administrative interface of the router. Several factors may come into play with completing these steps, some of which may be beyond the capabilities of general emulators such as QEMU. These factors include the following: 1) the absence of correct values in NVRAM (*e.g.*, boot parameters) or the absence of devices (*e.g.*, some hardware devices required for booting) necessary to satisfy boot conditions; 2) the absence of network interfaces (*e.g.*, different NIC interfaces) or connections (*e.g.*, the absence of an internet or

intranet connection) required to communicate with the applications in the emulated system; and 3) the absence of conditions necessary to launch a web server (*e.g.*, an IP address and a port).

Unlike previous approaches that sought to accurately emulate hardware behaviors, we take a different approach based on this hypothesis. That is, rather than resolving all hardware emulation dependencies, we aim to build an abstraction environment that meets the bare minimum requirements to run target applications for dynamic security testing. Specifically, the following two properties illustrate our abstraction emulation goal for dynamic security testing on IoT devices:

- Network accessibility: the network in the guest system must be accessible from the host system.
- Service availability: the target program in the guest system must be available for dynamic analysis.

Attaining these goals may not resolve fundamental emulation issues, such as reproducing 100% identical device behaviors. However, we believe that developing heuristic workarounds for setting up the network functionality and running applications can suffice for dynamic security testing, even if the emulated environment is not 100% identical to the target device. Here, we emphasize that the key concept is to deal with high-level properties in order to meet the requirements for running target applications in firmware, not to accurately emulate the underlying hardware. As such heuristics deal with high-level properties rather than hardware issues, they can be transferred across multiple devices and may mitigate failures caused by distinct root causes.

3.2.1 Wireless Routers as a IoT Firmware Case Study

We selected wireless routers as our analysis target for IoT firmware emulation on a large scale. This is because 1) wireless routers have typical characteristics of IoT devices, such as networking and web server functionality; 2) since their introduction in the early 2000s, their models and hardware/software configurations have become extremely diverse; and 3) their firmware images are available for large-scale analysis (we could collect 1,079 images).

Additionally, wireless routers are critical components of IoT and home security because they serve as gateways to the home network and manage other IoT devices via internal networking. Indeed, numerous botnets [8, 7] specifically target them in order to launch DDoS attacks. To this end, we focus on emulating wireless routers in order to test our hypothesis. We specifically focus on emulating their web services because these web services include the administrative interfaces of devices that have been found to contain a variety of vulnerabilities [86, 16, 87, 8, 7].

3.3 Developing Heuristics from Emulation Failures

As a first step toward learning and systematizing heuristics for avoiding emulation failures, we investigate firmware emulation failure cases using the state-of-the-art framework, Firmadyne [17]. We collected 1,079 firmware images for wireless routers from the websites of the top eight vendors [88]. We emulated 526 old versions of images using Firmadyne; it was only able to emulate 16.9% of the images. For the results, we classified the identified failure cases by the place where the emulation failed (See Figure 3.1): boot (§3.3.1), network (§3.3.2), library (§3.3.3), kernel (§3.3.4), and programs (§3.3.5). Through the analysis of failure cases, we were able to develop several heuristics for dealing with and avoiding failures. Note that some of the heuristics have also been proposed previously [17, 20]. Table 3.1 summarizes our systematized heuristics, which we detail in the following.

Table 3.1: Summary of heuristics for resolving emulation issues in order to run web services in wireless routers.

Where	Emulation Problem	Heuristics
❶ Boot	Incorrect booting sequence Missing files or directories	Utilize the original kernel's booting sequence Prepare files and directories prior to the emulation
❷ Network	No support for IP aliasing and VLANs No network interface Filtering rules in `iptables`	Correct routing rules and network interface settings Forcibly set up the default network interface Set the default policy to accept all incoming packets
❸ Library	Unknown NVRAM values Invalid return of `NULL` values	Search the filesystem for key-value pairs Return a valid string pointer instead of `NULL`
❹ Kernel	Insufficient support of kernel modules Incompatible kernel version	Emulate functions such as `ioctl` using a shared library Upgrade the kernel to v4.1 with a proper option
❺ Programs	Inactive web servers No support for extra commands Short timeout for emulation	Forcibly execute the web server Install full-featured busybox Increase the timeout to 240 s

3.3.1 Heuristics for Handling Boot Failures

The booting procedure involves the execution of several programs that initialize the system environment; thus, it fails if the emulated environment is unable to execute any (or even a part) of the programs in the boot process. Many boot failures have been observed to result in a kernel panic.

By analyzing these cases, we identified two distinct types of issues. First, the kernel of the emulator, which is different from the kernel running on the device, failed to locate the correct initializing program, which is configured specifically for the device by the device manufacturer. While program paths may vary between firmware images, the kernel image used in the emulator searches only for pre-defined paths, such as `/sbin/init`, `/etc/init`, or `/bin/init`, that are included by default in the Linux kernel. As a result, programs on alternate paths, such as `/etc/preinit`, cannot be executed properly, resulting in the kernel crashing. This failure frequently occurs in NETGEAR firmware images. By conducting an in-depth analysis, we discovered that they use `preinit` as the path of the initializing program, which is frequently used by an open-source embedded device project, called OpenWrt [89]. We verified that these NETGEAR devices are indeed implemented based on it. Additionally, we discovered that some TP-Link images also utilize `preinit`. To address this issue, Firmadyne built a script that searches for and executes a hard-coded list of files frequently accessed for initializing programs. However, these hard-coded candidates do not account for the diverse paths of initializing programs in the wild.

The second issue is that the boot process fails owing to the absence of files or directories, which are required by the `init` program. If the program attempts to access such non-existent paths, it will crash and eventually halt the booting process. Firmadyne attempted to address this by creating and mounting hard-coded paths, such as `proc`, `dev`, `sys`, or `root`, at the beginning of the custom booting script. Certain hard-coded paths worked flawlessly; for example, creating `/etc/TZ` or `/etc/hosts` resolved several cases of this failure. This approach, however, is unable to account for diverse cases. Additionally, because it forcibly creates files and directories before the firmware initializing itself, it collides with internal programs that create and mount other files or directories in the same paths.

To address these issues, we developed heuristics that extract useful information from the kernel and files in a firmware image. Note that a firmware image typically consists of a kernel image and a collection of programs stored in a filesystem. For the first issue, we utilized the kernel that came with the target device firmware. Specifically, we searched the kernel image for the string literals used in kernel configuration, *i.e.*, kernel's command line string. These strings are pre-defined by the device manufacturer during the device's development stage, and thus are naturally embedded in the kernel image. This information may include the path to the initializing program, the type of console, the root directory, the type of root filesystem, or the memory size. As an illustration of these heuristics, we searched for the string, "`init`", in one of NETGEAR's kernel images and obtained the string,

"`console=ttyS0,115200 root=31:08 rootfstype=squashfs init=/etc/preinit`", which appears to be a kernel boot argument. We can deduce from this string that the initializing program is located at `/etc/preinit`. By configuring the emulated environment with the information obtained, we could boot the kernel appropriately.

Similarly, we addressed the second issue by employing a string literal search strategy on the firmware filesystem. As a preprocessing stage for booting a firmware image, we extracted strings from the programs in the firmware filesystem that are highly likely to be program/directory path names. More specifically, we obtained several strings that start with common Unix system paths, such as `/var` or `/etc`. Then, we created directories or files based on those path values. As a result, we were able to successfully boot many firmware images without encountering any boot errors.

3.3.2 Heuristics for Handling Network Failures

The next step is to develop heuristics for handling network failures. After the boot procedure is complete, the network must be configured properly to allow the emulation host to communicate with the emulated firmware. Any failure in the network configuration will result in the failure of dynamic analysis, even if internal programs such as web services are running normally. This is because the host system is unable to communicate without networking capabilities.

During our investigation, we observed several networking failures while emulating firmware images. First, existing emulation frameworks are incapable of appropriately handling critical network operations, such as IP aliasing and virtual local area network (VLAN). IP aliasing enables the assignment of multiple IP addresses to a single network interface, while VLAN enables to network to be logically segmented, creating an isolated network environment. Both of these features are common in modern wireless routers.

IP aliasing failures are often discovered in D-Link images during network configuration on the host system. Specifically, when configuring the IP aliases, Firmadyne adds static routing rules for each IP alias to connect the guest network to the host network. However, it augments a single interface with multiple routing rules, causing the network to collide. In cases of VLAN failures, Firmadyne disregarded setting the host network, although the VLAN should be set to group the host and guest networks under the same VLAN id. To properly handle IP aliasing and VLAN, we developed dedicated routines that automatically configure routing rules and interface settings for IP aliasing and VLAN. Then, we applied these heuristics to the emulation runtime.

Additionally, we examined cases wherein existing emulation frameworks failed to retrieve information about network interfaces for certain firmware images. We discovered that some images are unable to obtain an IP address from external dynamic host configuration protocol (DHCP) servers in order to connect their wide area network (WAN) interface to the Internet. Because the emulated environment lacks a DHCP server, the emulated firmware is unable to retrieve an IP address and configure a network interface. Moreover, we discovered that a large number of ARM-based images are not emulated. Although we were not able to pinpoint the exact cause, we deduced that these cases resulted from a failure during the boot procedure prior to reaching the network setup phase. To address these issues, we developed a heuristic that forces the emulated system to set up a default network interface (e.g., eth0, a Realtek device), with an IP address of `192.168.0.1`, similar to the previous approach [20]. Consequently, we could avoid such failures.

Lastly, some firmware images in our dataset include a firewall (*i.e.*, `iptables`) that is configured by default to prevent unauthorized remote access; this is a common feature of modern wireless routers. As a result, the guest kernel drops all packets coming from the host. We discovered the majority of these cases in TP-Link, where the guest is unreachable despite proper configuration of the host and guest networks. This is not an emulation failure, as setting iptables follows the behavior of real devices. However, such filtering precludes analysis of their potential

vulnerabilities and threats. In practice, numerous device owners or administrators mistakenly configure these rules, thereby making the device publicly accessible [87, 2, 90]. Therefore, we developed a heuristic that forcibly flushes all `iptable` policies and sets the default policy to accept all incoming packets for further security analysis.

3.3.3 Heuristics for Handling NVRAM Failures

NVRAM is one of the most popular peripheral hardware devices for IoT devices including wireless routers. NVRAM is a type of flash memory that functions as a simple key/value storage. It stores various information for running devices, such as configurations of the device itself or other peripheral devices. In essence, the information stored in the NVRAM is required for peripherals and the target device to operate properly.

Because internal programs in firmware store/fetch configuration values to/from the NVRAM, it must be emulated correctly. Internal programs in IoT firmware frequently communicate with the NVRAM via libraries in their firmware. To take advantage of this feature, other emulation frameworks developed an additional library that simulates the NVRAM's interaction in order to run the firmware without actually having the real NVRAM. For instance, Firmadyne [17] implements a custom library called `libnvram` to emulate the NVRAM. This custom library is loaded on top of other libraries using the environmental variable called `LD_PRELOAD`, which forces internal programs to use it instead of the original library that interacts with the physical NVRAM. To emulate fetching and storing key-value pairs, the custom library intercepts NVRAM-related functions, such as `nvram_get()` and `nvram_set()`. To initialize the key-value pairs, it searches for and uses values from a hard-coded list of default files in the firmware. These default files are frequently present in recent IoT devices for *factory reset* functionality. For unknown keys, the custom library naively returned the `NULL` value.

However, relying on a hard-coded list may not suffice to cover diverse devices. We discovered many failures in D-Link and NETGEAR images, which use alternative default file paths not included in the hard-coded list. `/etc/nvram.default` and `/mnt/nvram_rt.default` are two examples of such default file paths. Additionally, we found that several key-value pair patterns are distinct from those used in the custom library (*e.g.*, `OBJ`, `ELM`).

To address these issues, we developed a heuristic strategy that automatically searches the firmware filesystem for the requested values. Specifically, we ran the firmware in our emulation environment to monitor the `nvram_get()` and `nvram_set()` functions for all fetched (required) keys during the first stage of emulation. Note that emulating such keys in the first stage would almost certainly fail. Then, we scanned the firmware filesystem for files that contained multiple instances of the recorded keys and extracted their corresponding values from those files. By performing this iteratively, we were able to obtain the majority of the required keys to emulate the NVRAM of the target device.

The heuristic may fail if we are unable to find the matching values for the requested keys to the NVRAM. To handle these unknown key/value pairs, we extended the custom library to return a valid pointer to an empty string as a value instead of a `NULL` pointer. This heuristic is based on the fact that many programs pass the returned value as an input to string-related functions, such as `strcpy()` or `strtok()`. Thus, returning a `NULL` value results in the program crashing immediately, whereas returning a zero-length valid string executes such functions successfully. As a result of this strategy, NVRAM-related crashes were significantly reduced, and programs were able to run properly even when the correct configuration values were unknown.

3.3.4 Heuristics for Handling Failures in Kernel

In addition to NVRAM, internal programs can co-operate with peripheral devices via kernel modules, also known as device drivers. They typically communicate with peripherals using `ioctl` commands. Unfortunately, emulating this procedure is not straightforward, as each device driver has distinctive characteristics, depending

on its developers and corresponding device. For example, if the kernel or its modules are not compatible with the actual device, the firmware programs will be unable to interact with the peripheral devices and may subsequently crash.

We investigated these failures and discovered that Firmadyne is running kernel version `v2.6.32`, which is incompatible with recent features used in target devices. In this case, simply upgrading the kernel to a newer version, *e.g.*, `v4.1.17`, resolved the majority of failure cases and successfully emulated more firmware images.

However, some firmware images, particularly those from the past, were not emulated by the new kernel version. These images failed due to a crash in the `libc` library. We investigated these cases and determined that the address space layout randomization (ASLR) of Linux kernel v4.1.17 is incompatible with older versions of `libc`. To address this, we compiled the new kernel with a compatibility option. Specifically, we set the `CONFIG_COMPAT_BRK` option, which disables the randomization of the `brk` area in heap memory. Using this new kernel, we were able to handle the aforementioned cases. Other compatibility issues may exist that our experiment did not detect. To address these issues, multiple kernel versions with various compiling options should be investigated further, which is one of the objectives of our future research.

Apart from kernel version compatibility issues, firmware emulation may fail if the kernel drivers are unable to communicate with the firmware programs. This issue is similar to that of the NVRAM; the device driver should return values that correspond to specific requests. Firmadyne addressed the issue similarly to the NVRAM. That is, they implemented a dummy kernel module that mimics the interaction between the kernel modules in firmware and peripheral devices by hard-coding device names and `ioctl` commands. This approach results in a high number of emulation failures, as the values passed to and returned from the call vary significantly depending on the firmware and device architecture. Such instances were discovered in NETGEAR images that make use of a module called `acos_nat` to communicate with a peripheral device mounted on `/dev/acos_nat_cli`. In those images, Firmadyne's module returns incorrect values and fails to run a web service, `httpd`. Additionally, we confirmed that `ioctl` commands vary by firmware architecture.

To resolve this issue, we developed a heuristic strategy similar to the one used to address NVRAM issues (§3.3.3). Instead of creating a dummy kernel module, we implemented a shared library with wrapper functions that return pre-defined values regardless of the `ioctl` interface/parameter variant. Because these wrapper functions operate as a high-level abstraction, each `ioctl` command does not need to be emulated for each device architecture. While we focused on `acos_nat` in this example, other peripheral accesses via shared libraries can be handled similarly. As a result of this heuristic, firmware programs can continue to execute without encountering system call errors.

3.3.5 Heuristics for Handling Internal Program Failures

Apart from the booting, networking, NVRAM, and kernel issues, executing applications can be disturbed during the emulation. Running applications is the most critical step in dynamic security testing because applications in firmware contain the device's core logic, which is the actual target of security testing. We discovered several issues, particularly from the web interfaces.

First, web server applications in some images failed to run despite a successful network setup. We expect that the network device will be set up after the web server has been run during emulation; thus, the web server was unable to bind to the network device. In this case, we developed a heuristic that forcibly runs the web server after the entire initialization step is complete. Specifically, the heuristic searches the firmware filesystem for a widely used web server program (*e.g.*, `httpd`, `lighttpd`, `boa`, `goahead`) and its corresponding configuration file. The heuristic then executes the web server with the configuration file.

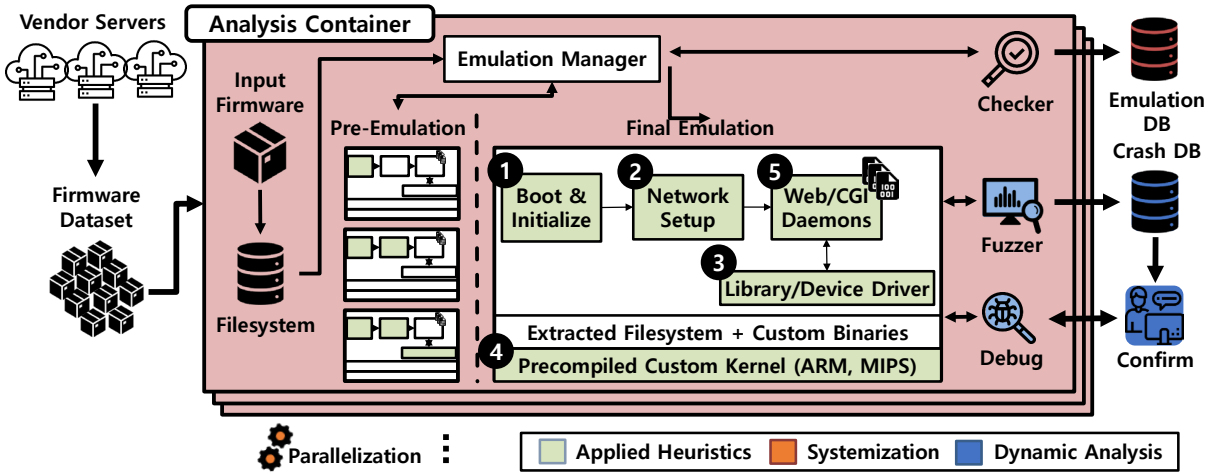


Figure 3.2: FIRMAE architecture overview

Additionally, missing files in the firmware filesystem that are required for an emulation environment can result in emulation failure. The emulated environment can be missing these files because many developers of IoT devices remove unnecessary programs from the filesystem to reduce the size of device firmware. However, in an emulation environment where the heuristics are used, we may require multiple configuration tools, such as `mount`, `ln`, `ifconfig`, and `ip`. Without such tools, emulation will fail and the applied heuristics will be ineffective. To address this issue, we installed the latest version of `busybox`, a swiss-army knife for the Unix box, in order to supply the emulation environment with the necessary command-line tools.

Lastly, emulation may take an inordinate amount of time. Firmadyne set a 60-second timeout period. However, firmware images, particularly those from NETGEAR, take a longer time to complete their booting procedure. As a result of Firmadyne's short timeout, successful emulation was prevented. We investigated such cases and empirically determined that 240 s is an appropriate timeout. Although this change was straightforward, it resulted in the successful emulation of over 60 firmware images.

3.4 FirmAE: Systematizing Heuristics Learned

We systematized the heuristics learned from our in-depth failure analysis. We implemented a prototype system, named FIRMAE, based on Firmadyne [17]. For this, we wrote a total of 3671 lines of code (LoC) in Python and shell scripts. Figure 3.2 depicts a component-wise overview of FIRMAE. The key difference of FIRMAE from Firmadyne is that, in addition to the techniques developed by Firmadyne, FIRMAE employs a variety of systematized heuristics to increase the success rate of emulation. From steps ❶ through ❺ of Figure 3.2, FIRMAE applies corresponding heuristics to prevent detected emulation failures. Additionally, we added interfaces for dynamic security analysis to FIRMAE, which we will detail in §3.5.3.

To facilitate practical large-scale firmware emulation, FIRMAE provides two functionalities: automation and parallelization. To fully automate FIRMAE, we automated all user interactions and added an evaluation procedure that periodically checks the emulation requirements (*i.e.*, network accessibility and web service availability). To accomplish this, we built a checker module that periodically runs the `ping` and `curl` commands and reports on whether the emulation environment meets the requirements.

Additionally, we parallelized the emulation procedure to run multiple firmware images effectively. For this, we leveraged Docker [91], which is one of the most popular containerization frameworks. Containerization enables each firmware image to be emulated independently within a container that contains all necessary packages and

dependencies. Moreover, containerization helps us to take advantage of abstracting the network connection between the host and the guest systems. QEMU [37], the base emulator of FIRMAE, creates an additional network interface (*i.e.*, *TAP*) in the host system. This interface is linked to one of the guest system's network interfaces. Thus, each emulated firmware should have a separate interface with a unique IP address in the host system; otherwise, the network will collide. Because containerization isolates the network environment of each container, packets from the host system can be routed properly to the guest even during parallel emulation. We also placed our checker module for automation and dynamic analysis engine inside each container. Consequently, firmware images can be emulated and analyzed quickly and reliably; FIRMAE can emulate a large number of firmware images in parallel by running the corresponding number of container instances.

3.5 Evaluation

In the following, we evaluate FIRMAE, which systematizes our heuristics, for its effectiveness in firmware emulation and dynamic security testing.

3.5.1 Experimental Setup

To assess the effectiveness of FIRMAE, we first established our dataset from the top eight wireless home router vendors [88]. We collected 1,306 firmware images from vendors' websites and unpacking them using Binwalk [26] to extract filesystems. Then, we filtered them by determining whether the operating system of each image is ARM little-endian (ARMel), MIPS little-endian (MIPSel), or MIPS big-endian (MIPSeb). These architectures account for more than 97% of our initial collection. To further evaluate our approach, we also prepared firmware images for IP cameras in the same manner.

Our final dataset contains a total of 1,124 firmware images, including 1,079 images of wireless routers and 45 images of IP cameras. We categorize them as three datasets: `AnalysisSet`, `LatestSet`, and `CamSet`. Their summary is presented in Table 3.2 along with the emulation result, while a more detailed version is shown in Table 3.5. The `AnalysisSet` contains 526 outdated images from 3 vendors, whereas the `LatestSet` and `CamSet` contain only the latest firmware images as of December 2018. The `LatestSet` has 553 latest images from 8 vendors, including the vendors covered by the `AnalysisSet`, and the `CamSet` includes 45 latest images from 3 vendors. Accordingly, the `AnalysisSet` may contain multiple firmware versions per device, whereas the `LatestSet` and `CamSet` have only one image per device. There is no overlap between the datasets, *i.e.*, they do not share an identical image. We analyzed emulation failure cases using the `AnalysisSet`. As a result of our analysis, we developed several heuristics to address the failures and increase the emulation rate. We systematized these heuristics into FIRMAE and evaluated it against the `LatestSet` and `CamSet`.

All of our experiments were conducted on a server powered by four Intel Xeon E7-8867v4 2.40 GHz processors, 896 GB DDR4 RAM, and a 4 TB SSD. We installed Ubuntu 16.04 with PostgreSQL v9.5.14 [92] and Docker v18.09.4 [91] on the server.

3.5.2 Effectiveness of Heuristics in IoT Firmware Emulation

To evaluate the effectiveness of systematized heuristics in firmware emulation, we emulate each dataset using FIRMAE and Firmadyne. In this experiment, we evaluate three questions:

- *Q*1: How do well-systematized heuristics emulate firmware images more successfully than the existing framework?

Table 3.2: Emulation results of running firmware images using Firmadyne and FIRMAE.

Dataset	Vendor	Images	Firmadyne		FIRMAE	
			Net	Web	Net	Web
AnalysisSet	D-Link	179	55	54 (30.17%)	177	167 (93.30%)
	TP-Link	73	26	5 (6.85%)	73	59 (80.82%)
	NETGEAR	274	86	30 (10.95%)	259	257 (93.80%)
Sub Total		526	167	89 (16.92%)	509	483 (91.83%)
LatestSet	D-Link	58	18	17 (29.31%)	54	48 (82.76%)
	TP-Link	69	33	10 (14.49%)	69	54 (78.26%)
	NETGEAR	101	30	7 (6.93%)	92	79 (78.22%)
	TRENDnet	106	35	23 (21.70%)	91	63 (59.43%)
	ASUS	107	27	25 (23.36%)	63	62 (57.94%)
	Belkin	37	2	2 (5.41%)	30	22 (59.46%)
	Linksys	55	13	8 (14.55%)	48	44 (80.00%)
	Zyxel	20	3	0 (0.00%)	18	10 (50.00%)
Sub Total		553	161	92 (16.64%)	465	382 (69.08%)
CamSet	D-Link	26	0	0 (0.00%)	19	17 (65.38%)
	TP-Link	6	0	0 (0.00%)	6	0 (0.00%)
	TRENDnet	13	2	2 (15.38%)	10	10 (76.92%)
Sub Total		45	2	2 (4.44%)	35	27 (60.00%)
Total		**1,124**	**330**	**183 (16.28%)**	**1,009**	**892 (79.36%)**

Net: network accessibility, Web: web service availability.

- *Q*2: Are the heuristics learned from older firmware images transferrable to newer firmware versions?
- *Q*3: Are the heuristics transferrable to other IoT devices besides wireless routers?

As our goal is to emulate web services for dynamic analysis, we verify the network accessibility and web service availability for each emulated firmware. Henceforth, we will refer to the web service availability as the emulation rate. The final results are listed in Table 3.2. As FIRMAE is fully automated and parallelized, the total emulation time for all datasets took less than 4 h (14289 s).

Overall, the emulation rate increased significantly from 16.28% to 79.36% (by 487%), corroborating Q1. Because our investigation is based on the `AnalysisSet`, it shows the highest emulation rate of 91.83%. The emulation rates for the `LatestSet` and `CamSet` also demonstrate a significant improvement over those obtained by Firmadyne, supporting both Q2 and Q3. In `AnalysisSet`, the emulation rate of NETGEAR images increased the most, from 10.95% to 93.80% (by 857%), owing to the heuristics that forcibly set up the default network interface. TRENDnet, ASUS, Belkin, and Zyxel all have emulation rates of less than 60% in the `LatestSet`. These lower rates are attributed to the increased number of kernel modules in these images and the use of custom hardware interfaces. We detail this in the following section (§3.6).

Additionally, we investigate the effectiveness of each heuristic category by omitting a specific category from the final version of FIRMAE, which incorporates all heuristics. This is because heuristics should cooperate to address failures; thus, omitting a specific category of heuristics has a direct effect on the emulation rate. Figure 3.3 depicts these results, and a more detailed version is included in Table 3.6.

The heuristics used to address NVRAM issues appear to be the most effective, decreasing the emulation rate by 35% on average across all datasets. This result is consistent with the Firmadyne approach, which focuses on NVRAM emulation. Omitting heuristics for boot and network issues also significantly drops the emulation rate by ≈30%. Without the heuristics for kernel issues, only 4.88% of firmware images across all datasets were affected. The heuristics for resolving internal program issues affected 22.35% of firmware images. These results demonstrate that each of the proposed heuristic categories is indeed effective and scalable for successful firmware emulation.

The emulation rates for `CamSet` indicate that resolving failure issues of wireless routers can also aid in

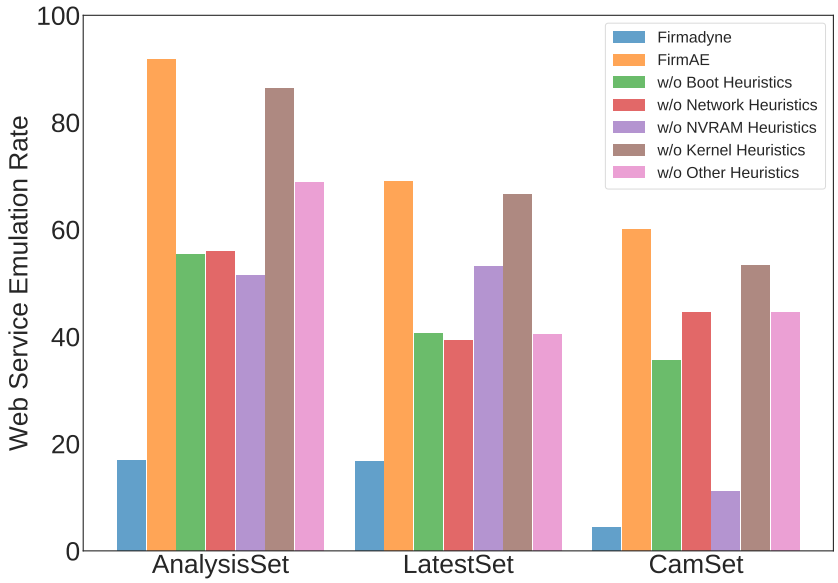


Figure 3.3: Emulation rates obtained by omitting a specific heuristic category from the final version of FIRMAE.

emulating IP cameras. In particular, Firmadyne was unable to emulate any of the D-Link images, whereas FIRMAE was able to emulate more than 65% of the images. Nevertheless, FIRMAE is incapable of emulating all TP-Link images. We looked into these failed cases and discovered that they lacked web servers. The results of `CamSet` demonstrate that many IP cameras share similar characteristics to wireless routers, implying that heuristics for wireless routers can be applied to IP cameras as well.

3.5.3 Dynamic Analysis Capabilities of Firmware Emulation with Heuristics

FIRMAE relies on an imperfect emulation of firmware devices based on heuristics from empirical observations. Thereby, its capability of applying dynamic security analysis would be questionable, *i.e.*, whether FIRMAE can be used for discovering security vulnerabilities or not. To demonstrate that systematized heuristics are indeed effective for bug discovery, we evaluate the following two questions:

- *Q4*: Can exploits for known vulnerabilities (*i.e.*, 1-days) be used against firmware running on FIRMAE?
- *Q5*: Is it possible for dynamic analysis to discover new vulnerabilities (*i.e.*, 0-days) against firmware running on FIRMAE?

The primary step in the dynamic analysis is to initialize web services unless they do not receive any other requests. A large portion of the web services in our dataset requires a network and security configuration (*e.g.*, admin or AP password) in the admin pages, and this initialization procedure differs in each firmware. Web servers in most firmware images in D-Link, TP-Link, Belkin, Linksys, and ZyXEL automatically initialize themselves after successful emulation, whereas those in ASUS and TRENDnet must be initialized in person. Fortunately, many of the web servers have a skip button to configure default options. Some web services do not explicitly have a skip button, but have internal JavaScript functions that behave identically. Meanwhile, some require a manual admin password. To automate the initialization process, we empirically analyzed each web service and extracted representative patterns of buttons and menus from their initial web pages. Then, we utilize the extracted patterns and automate the process. For this, we leveraged Selenium [93], which is an open-sourced tool that can provide an interface like a real browser.

Table 3.3: Number of 1-days discovered on the outdated firmware images (`AnalysisSet`)

		Firmadyne	FirmAE
Vulnerability Category	# of PoC	# of Images (Unique)	# of Images (Unique)
Information leak	2	0 (0)	17 (17)
Command injection	9	10 (6)	152 (65)
Password disclosure	2	4 (3)	146 (99)
Authentication bypass	2	0 (0)	5 (5)
Total	15	14 (9)	320 (128)

Table 3.4: Number of new vulnerabilities discovered on the latest firmware images (`LatestSet` and `CamSet`)

Type	Vulnerability Category	# of Vulns	# of Devices	# of Vendors
1-day	Information leak	2	32	2
	Command injection	5	28	2
	Backdoor	2	3	1
	Path traversal	2	9	2
0-day	Command injection	7	16	2
	Buffer overflow	5	7	4
Total		23	95	6

To test Q4, we launched 1-day exploits using RouterSploit [28] on each emulated firmware image in `AnalysisSet` by FIRMAE and Firmadyne, respectively, and Table 3.3 demonstrates the result. Note that the web service of each image is already initialized as described above. While 14 exploits worked on firmware emulated without applying any heuristics (*i.e.*, Firmadyne), 320 exploits worked on firmware emulated by applying all the proposed heuristics (*i.e.*, FIRMAE), supporting Q4. Here, a working 1-day exploit indicates that it can discover such a vulnerability from the image. Consequently, dynamic analysis on firmware images emulated with FIRMAE can discover significantly more vulnerabilities than that on images emulated with Firmadyne. Note that all the identified vulnerabilities are located in web services, such as SOAP CGI, UPnP, and HNAP.

To test Q5, we ran images from `LatestSet` and `CamSet`, which contains only the latest firmware images, to check if FIRMAE can help discover new vulnerabilities via dynamic security testing. In this regard, we define vulnerabilities as 1) known but remaining unpatched on the latest version or the different models (1-days), or 2) new (0-days). Table 3.4 lists the unique number of newly identified vulnerabilities, affected devices, and their corresponding vendors. First, we launched 1-day exploits from RouterSploit [28] as the same approach described above and discovered 11 1-day vulnerabilities affecting 72 unique devices. To discover 0-day vulnerabilities, we implemented a simple fuzzer with 880 LoC in Python; the fuzzer generates crafted requests and sends them to the emulated web services, 12 new 0-day vulnerabilities affecting 23 unique devices. These results support Q5. In the following, we detail how our fuzzer operates.

Our fuzzer first searches the filesystem of the target firmware and generates a list of web page candidates by checking the extension of files, such as `.html`, `.aspx`, or `.xml`. Then, it extracts possible parameters from the candidates and generates requests to detect vulnerabilities. For example, for the `.htm` and `.html` candidates, the fuzzer parses the HTML tags, such as `script`, `form`, and `input`, to extract target URLs, methods, and parameter information. This approach is particularly helpful in building requests for devices that use the home network administration protocol (HNAP). The HNAP request is based on the XML format, and the default value is set up in the javascript code of a `.html` page. By utilizing the extracted information, the fuzzer can construct a valid request template for fuzzing. Because it searches for candidates from the filesystem, it could also check web services that are not reachable by crawling.

Among the various types of vulnerability, we focus on command injection and buffer overflow, as they are

often found in embedded devices. To detect command injection vulnerabilities, our fuzzer sends payloads, which are essentially a combination of candidate characters, such as `';'`, `'"'`, or `'&'`, followed by a shell command executing our own program. We place this program to log useful information, such as time and environment variables, thereby checking if the vulnerability is triggered. We also hook the `execve` system call, to easily detect if our inputs are injected in the command. For buffer overflow detection, FIRMAE provides feedback when a crash occurs. We also utilized boundary values, such as a large-sized buffer, when generating fuzzing inputs, as they are more likely to trigger vulnerabilities.

Any bugs reported by the fuzzer must be verified. For this, we added debugging programs, such as `strace`, `gdb`, and `gdbserver` to the filesystem of target firmware. Note we could utilize the `ptrace` system call for debugging as we upgraded the kernel version (§3.3.4). We also added `netcat` and `telnetd` to access the guest shell. With these tools, we manually verified the identified bugs.

By using the fuzzer, we identified 12 new 0-day vulnerabilities affecting 23 unique devices. We reported all discovered 0-days to the corresponding vendors, and these were acknowledged by Dec. 2019. Each fuzzing request took an average of 10–15 s when running 50 images in parallel, and the average time spent for finding each vulnerability was 70 min, with a maximum of 150 min. The fuzzing throughput can vary according to the system spec and the number of parallel emulation instances.

In summary, the dynamic analysis results demonstrate that the heuristic-based emulation approach of FIRMAE is effective for vulnerability analysis, supporting both Q4 and Q5.

3.6 Post-Emulation Analysis

3.6.1 Analyzing Unhandled Failure Issues

While our heuristics significantly increased emulation success rates and facilitated vulnerability discovery, they may not be applied to new types of devices or configurations owing to the hardware/environmental diversity in the IoT ecosystem. After running firmware images with FIRMAE, we investigate unhandled failures that cannot be resolved with simple heuristics but require more complex virtualization.

First, as discussed in previous studies [16, 17, 18, 19, 20], emulating kernel modules is not straightforward because 1) different kernel versions frequently cause compatibility issues and 2) some firmware images may be devoid of a kernel, rendering useful information unavailable. In a few cases, web servers and other programs made use of kernel modules located in the `/proc` directory. Those programs often crashed because such files were not present in the emulated environment. Web servers in TP-Link firmware images, for example, attempted to access a kernel module located at `/proc/simple_config/system_code` and subsequently crash due to the module's absence.

Moreover, some internal programs of the firmware utilize their own dedicated interfaces for peripheral communication, further hardening their emulation. For instance, we emulated an NVRAM using popular library calls. However, certain D-Link firmware programs used the `/bin/flash` command to directly access `/dev/nvram`. Similarly, in a few TP-Link firmware images, `httpd` servers accessed a flash memory located at `/dev/ar7100_flash_chrdev` in order to retrieve device configuration information. Meanwhile, in Linksys firmware, web servers named `webs` directly manipulated the `/dev/mtd` interface. They also verified the firmware's integrity, signature, and version.

Lastly, while web servers are accessible, some of them rarely respond with a server error, such as a *500 Internal Server Error*. This error can be caused by various factors, including syntax/code errors in CGI programs, incorrect web interface configuration, or errors. However, the majority of error cases are a result of backend CGI program crashes. We used reverse engineering to analyze the CGI programs and discovered that they all suffer from the same hardware interface issues. They attempted to access entries under `/proc` or `/dev` to obtain configuration

values and, if not successful, terminate abnormally.

These cases present the difficulty of emulating peripheral communication without physical devices. In this regard, we believe that further empirical investigations to develop additional heuristics is indispensable to deal with the convoluted nature of the IoT ecosystem. However, as shown in Q2 and Q3, the developed heuristics can be transferred to newer device versions and similar device families. Therefore, we encourage further research into developing and systematizing heuristics through hands-on failure analysis.

3.6.2 Lessons Learned from Vulnerability Analysis

We discovered two interesting points during our investigation into vulnerability analysis. First, we discovered that several vendors are affected by the same vulnerabilities. For example, certain D-Link and TRENDnet devices share the same vulnerabilities in UPnP and SOAP CGI programs, *i.e.*, a information leaking vulnerability and a command injection vulnerability. Meanwhile, certain NETGEAR devices share Xiongmai's path traversal vulnerability. From these examples, we can deduce that similarity-based analysis techniques may be effective for vulnerability analysis of IoT devices, which we will describe in Chapter 6.

Another point worth noting is the effectiveness of system-level emulation. During our investigation, we discovered that analyzing a target web service can reveal vulnerabilities in other programs that are associated with it. Specifically, when we sent a large payload to detect buffer overflow, the payload was stored in a file by a target CGI program. Then, as a result of the large payload, another program that reads the written file crashed. This vulnerability is only detectable in a system-level emulation environment, as user-level emulation does not consider filesystem relationships.

3.7 Discussions

In this study, we developed a simple analysis engine for dynamic analysis that automatically initializes, logs into, and analyzes web services. Each step, however, can be further improved by incorporating other promising techniques. For instance, symbolic execution can be used to analyze and bypass the login procedure [12]. Moreover, by adopting other dynamic tools [28, 29, 30, 31, 32, 33, 15, 34, 35, 36], fuzzing strategies [94, 95], hybrid analysis approaches [96, 97], or similarity techniques [57, 63], one may discover additional vulnerabilities. We leave such promising improvements to the dynamic analysis engine as future work.

Additionally, successful firmware emulation can be used to establish a honeypot for the analysis of various attacks targeting IoT devices. In practice, there have been several emulation-based honeypot approaches [20, 98, 99, 100]. Vetterl *et al*. [20], in particular, created a honeypot named Honware by emulating device firmware in a manner similar to that used by FIRMAE. The authors investigated emulation failure cases in order to increase the network accessibility rate. Accordingly, FIRMAE's approach to configuring a default network setting is fairly similar to Honware's. However, FIRMAE includes additional heuristics for running web services, which increased the emulation rate even further and enabled active analysis of web service vulnerabilities. As a result, we believe that empirical analysis and the development of heuristics for successful firmware emulation are also required when building an IoT honeypot.

3.8 Conclusion

Recent advancements in dynamic security testing, such as automated pentesting, fuzzing, and symbolic execution, and their combination, can discover security vulnerabilities automatically in a scalable manner. By

applying such dynamic analysis to the IoT ecosystem, it is possible to enhance its security, particularly by harnessing the testing scalability to deal with numerous available devices. However, emulating device firmware itself is challenging owing to the convoluted nature of IoT device hardware/software implementation practices.

In addition to Firmadyne [17], several approaches [18, 19, 38], have attempted to precisely emulate hardware devices by modeling memory-mapped input/output operations in peripheral communication [18, 19] or by building an abstract layer to deal with hardware emulations [38]. These approaches are essential in the long run to achieve better accuracy in testing; however, such frameworks continue to face limitations in terms of covering firmware across highly diversified IoT devices.

We believe that accumulating heuristic knowledge for circumventing firmware emulation failures is the final step towards overcoming such limitations. Systematizing the heuristics learned from failure cases enables large-scale firmware emulation, as such heuristic knowledge is transferrable to newer device versions and similar device families. We recommend that future research conduct additional empirical investigations, systematize, and share the resulting knowledge for scalable dynamic security analysis of IoT ecosystems.

Table 3.5: Full statistics of firmware dataset.

Dataset	Vendor	# of Images	# of Architecture				# of Web Services								
			arm32el	mips32el	mips32eb	etc	httpd	uhttpd	mini_httpd	lighttpd	alphapd	goahead	boa	jjhttpd	etc
AnalysisSet	D-Link	179	22 (12.29%)	82 (45.81%)	75 (41.90%)	0 (0.00%)	102 (56.98%)	0 (0.00%)	0 (0.00%)	9 (5.03%)	39 (21.79%)	10 (5.59%)	2 (1.12%)	14 (7.82%)	5 (2.79%)
	TP-Link	73	10 (13.70%)	15 (20.55%)	48 (65.75%)	0 (0.00%)	64 (87.67%)	9 (12.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	NETGEAR	274	105 (38.32%)	56 (20.44%)	113 (41.24%)	0 (0.00%)	125 (45.62%)	77 (28.10%)	69 (25.18%)	5 (1.82%)	0 (0.00%)	0 (0.00%)	3 (1.09%)	0 (0.00%)	0 (0.00%)
Sub Total		526	137 (26.05%)	153 (29.09%)	236 (44.87%)	0 (0.00%)	291 (55.32%)	86 (16.35%)	69 (13.12%)	14 (2.66%)	39 (7.41%)	10 (1.90%)	5 (0.95%)	14 (2.66%)	5 (0.95%)
LatestSet	D-Link	58	9 (15.52%)	17 (29.31%)	32 (55.17%)	6 (10.34%)	39 (67.24%)	0 (0.00%)	8 (13.79%)	12 (20.69%)	0 (0.00%)	1 (1.72%)	4 (6.90%)	3 (5.17%)	0 (0.00%)
	TP-Link	69	13 (18.84%)	22 (31.88%)	34 (49.28%)	0 (0.00%)	53 (76.81%)	16 (23.19%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	NETGEAR	101	32 (31.68%)	44 (43.56%)	25 (24.75%)	1 (0.99%)	46 (45.54%)	36 (35.64%)	19 (18.81%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	2 (1.98%)	0 (0.00%)	0 (0.00%)
	TRENDnet	106	18 (16.98%)	29 (27.36%)	59 (55.66%)	0 (0.00%)	28 (26.42%)	9 (8.49%)	13 (12.26%)	6 (5.66%)	0 (0.00%)	11 (10.38%)	11 (10.38%)	3 (2.83%)	21 (19.81%)
	ASUS	107	28 (26.17%)	72 (67.29%)	2 (1.87%)	0 (0.00%)	106 (99.07%)	0 (0.00%)	0 (0.00%)	51 (47.66%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (0.93%)
	Belkin	37	2 (5.41%)	20 (54.05%)	15 (40.54%)	0 (0.00%)	25 (67.57%)	0 (0.00%)	11 (29.73%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (10.81%)
	Linksys	55	15 (27.27%)	30 (54.55%)	10 (18.18%)	1 (1.82%)	23 (41.82%)	1 (1.82%)	6 (10.91%)	26 (47.27%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Zyxel	20	5 (25.00%)	10 (50.00%)	5 (25.00%)	0 (0.00%)	2 (10.00%)	0 (0.00%)	2 (10.00%)	7 (35.00%)	0 (0.00%)	2 (10.00%)	5 (25.00%)	0 (0.00%)	3 (15.00%)
Sub Total		553	122 (22.06%)	244 (44.12%)	182 (32.91%)	8 (1.45%)	322 (58.23%)	62 (11.21%)	59 (10.67%)	102 (18.44%)	0 (0.00%)	14 (2.53%)	22 (3.98%)	6 (1.08%)	29 (5.24%)
CamSet	D-Link	26	8 (30.77%)	15 (57.69%)	3 (11.54%)	0 (0.00%)	6 (23.08%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	13 (50.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (23.08%)
	TP-Link	6	6 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (100.00%)
	TRENDnet	13	1 (7.69%)	10 (76.92%)	2 (15.38%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (46.15%)	0 (0.00%)	2 (15.38%)	0 (0.00%)	2 (15.38%)
Sub Total		45	15 (33.33%)	25 (55.56%)	5 (11.11%)	0 (0.00%)	6 (13.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	19 (42.22%)	0 (0.00%)	2 (4.44%)	0 (0.00%)	14 (31.11%)
Total		**1124**	**274 (24.38%)**	**422 (37.54%)**	**423 (37.63%)**	**8 (0.71%)**	**619 (55.07%)**	**148 (13.17%)**	**128 (11.39%)**	**116 (10.32%)**	**58 (5.16%)**	**24 (2.14%)**	**29 (2.58%)**	**20 (1.78%)**	**48 (4.27%)**

Table 3.6: Full results obtained by omitting each heuristic category from the final version of FIRMAE.

Dataset	Vendor	# of Images	FIRMAE		w/o Boot Heuristics		w/o Network Heuristics		w/o NVRAM Heuristics		w/o Kernel Heuristics		w/o Other Heuristics	
			Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service
AnalysisSet	D-Link	179	177 (98.88%)	167 (93.30%)	162 (90.50%)	145 (81.01%)	100 (55.87%)	90 (50.28%)	176 (98.32%)	129 (72.07%)	154 (86.03%)	144 (80.45%)	173 (96.65%)	146 (81.56%)
	TP-Link	73	73 (100.00%)	59 (80.82%)	53 (72.60%)	36 (49.32%)	27 (36.99%)	13 (17.81%)	73 (100.00%)	56 (76.71%)	73 (100.00%)	60 (82.19%)	55 (75.34%)	31 (42.47%)
	NETGEAR	274	259 (94.53%)	257 (93.80%)	110 (40.15%)	110 (40.15%)	191 (69.71%)	191 (69.71%)	239 (87.23%)	86 (31.39%)	259 (94.53%)	250 (91.24%)	252 (91.97%)	185 (67.52%)
Sub Total		526	509 (96.77%)	483 (91.83%)	325 (61.79%)	291 (55.32%)	318 (60.46%)	294 (55.89%)	488 (92.78%)	271 (51.52%)	486 (92.40%)	454 (86.31%)	480 (91.25%)	362 (68.82%)
LatestSet	D-Link	58	54 (93.10%)	48 (82.76%)	46 (79.31%)	41 (70.69%)	19 (32.76%)	18 (31.03%)	54 (93.10%)	48 (82.76%)	54 (93.10%)	40 (68.97%)	51 (87.93%)	45 (77.59%)
	TP-Link	69	69 (100.00%)	54 (78.26%)	54 (78.26%)	32 (46.38%)	39 (56.52%)	23 (33.33%)	69 (100.00%)	53 (76.81%)	69 (100.00%)	57 (82.61%)	57 (82.61%)	23 (33.33%)
	NETGEAR	101	92 (91.09%)	79 (78.22%)	49 (48.51%)	41 (40.59%)	68 (67.33%)	60 (59.41%)	92 (91.09%)	25 (24.75%)	92 (91.09%)	82 (81.19%)	87 (86.14%)	54 (53.47%)
	TRENDnet	106	91 (85.85%)	63 (59.43%)	55 (51.89%)	41 (38.68%)	49 (46.23%)	37 (34.91%)	91 (85.85%)	56 (52.83%)	87 (82.08%)	52 (49.06%)	84 (79.25%)	44 (41.51%)
	ASUS	107	63 (58.88%)	62 (57.94%)	31 (28.97%)	31 (28.97%)	34 (31.78%)	32 (29.91%)	63 (58.88%)	45 (42.06%)	62 (57.94%)	61 (57.01%)	58 (54.21%)	25 (23.36%)
	Belkin	37	30 (81.08%)	22 (59.46%)	3 (8.11%)	3 (8.11%)	14 (37.84%)	14 (37.84%)	30 (81.08%)	19 (51.35%)	30 (81.08%)	22 (59.46%)	29 (78.38%)	5 (13.51%)
	Linksys	55	48 (87.27%)	44 (80.00%)	34 (61.82%)	34 (61.82%)	31 (56.36%)	31 (56.36%)	47 (85.45%)	42 (76.36%)	48 (87.27%)	44 (80.00%)	44 (80.00%)	27 (49.09%)
	Zyxel	20	18 (90.00%)	10 (50.00%)	7 (35.00%)	2 (10.00%)	8 (40.00%)	2 (10.00%)	18 (90.00%)	6 (30.00%)	18 (90.00%)	10 (50.00%)	18 (90.00%)	1 (5.00%)
Sub Total		553	465 (84.09%)	382 (69.08%)	279 (50.45%)	225 (40.69%)	262 (47.38%)	217 (39.24%)	464 (83.91%)	294 (53.16%)	460 (83.18%)	368 (66.55%)	428 (77.40%)	224 (40.51%)
CamSet	D-Link	26	19 (73.08%)	17 (65.38%)	13 (50.00%)	12 (46.15%)	13 (50.00%)	11 (42.31%)	18 (69.23%)	3 (11.54%)	18 (69.23%)	16 (61.54%)	18 (69.23%)	14 (53.85%)
	TP-Link	6	6 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)
	TRENDnet	13	10 (76.92%)	10 (76.92%)	10 (76.92%)	4 (30.77%)	10 (76.92%)	9 (69.23%)	10 (76.92%)	2 (15.38%)	10 (76.92%)	8 (61.54%)	10 (76.92%)	6 (46.15%)
Sub Total		45	35 (77.78%)	27 (60.00%)	23 (51.11%)	16 (35.56%)	23 (51.11%)	20 (44.44%)	34 (75.56%)	5 (11.11%)	34 (75.56%)	24 (53.33%)	34 (75.56%)	20 (44.44%)
Total		**1124**	**1009 (89.77%)**	**892 (79.36%)**	**627 (55.78%)**	**532 (47.33%)**	**603 (53.65%)**	**531 (47.24%)**	**986 (87.72%)**	**570 (50.71%)**	**980 (87.19%)**	**846 (75.27%)**	**942 (83.81%)**	**606 (53.91%)**

Chapter 4. Towards Large-Scale Firmware Structure Analysis

In the previous chapter, we demonstrated the efficacy of systematized heuristics for successful firmware emulation for dynamic security analysis of IoT devices, particularly wireless routers and IP cameras. Unfortunately, such an emulation-based analysis approach may not be applicable to other types of IoT devices, which exhibit characteristics that are fundamentally different from those of wireless routers and IP cameras. For example, baseband is one of such devices; it performs a variety of complex functions, such as real-time signal processing, and communicates with multiple peripherals. Notably, the baseband's firmware runs as a real-time operating system. Therefore, emulating baseband firmware and conducting dynamic security analysis on it is still an unresolved research problem.

To investigate this issue further in the context of analyzing complex IoT devices, we chose the smartphone cellular baseband as our case study target, as it plays a crucial role in mobile communication. Although users interact primarily with the interfaces of user applications running on an application processor, all application data is transferred via a separate processor called a baseband processor (BP), which is dedicated to radio communication. To provide seamless network services to users, the BP runs its software as a single executable, which is typically a real-time operating system, and communicates continuously with cellular networks via a variety of cellular protocol messages.

The security of baseband software is critical because if it is compromised, any data in transit can be sniffed or even spoofed. Therefore, researchers have attempted to analyze its security, particularly for cellular layer 3 (L3) protocols that manage core operations, such as mobility/session management or cryptographic operations to ensure user privacy. Nonetheless, these approaches to baseband analysis are limited to only a small portion of baseband software or a few baseband models/versions. The key problem here is the obscurity and complexity of baseband firmware; vendors are reluctant to publish details about their baseband. Therefore, existing approaches have relied on black-box approaches [72, 73, 74, 78] or on ad-hoc manual inspection [76, 77, 75].

To conduct a scalable security analysis of baseband software, it is necessary to unveil its firmware structure. This procedure is comprised of three steps: 1) analyzing the memory layout where the firmware will be loaded; 2) identifying function boundaries from unknown byte codes; and 3) detecting a target function (*e.g.*, an L3 protocol message decoder) among the numerous identified functions. To achieve this, we first investigated existing approaches to firmware structure analysis of IoT devices. During our literature study, we discovered that existing studies have focused on relatively simple Linux-based devices, such as wireless routers, IP cameras, and printers, but not on complex devices, such as baseband. Furthermore, the majority of studies [46, 47, 48, 49, 50, 51] have focused on identifying function boundaries (*i.e.*, the second step), but not on the other two steps. Even such approaches to function identification are limited to simple IoT devices or a few architectures (mostly x86, a few ARM).

Analyzing baseband firmware is not trivial because its size is extremely large (*i.e.*, over 30 MB), and it contains a large number of functions (*i.e.*, over 90K) associated with complex cellular protocols. To see the impact of this issue in practice, we obtained 18 baseband firmware images (based on the ARM architecture) from one of the top three smartphone manufacturers and analyzed them using IDA Pro [101], the state-of-the-art binary analysis tool. Notably, IDA Pro identified fewer than 600 functions on average (in 30 MB firmware); thus, no further security testing was feasible.

To tackle these issues, we propose that *well-systematized heuristic workarounds* can enable successful firmware structure analysis of baseband for further scalable security testing. We discovered that by empirically analyzing the

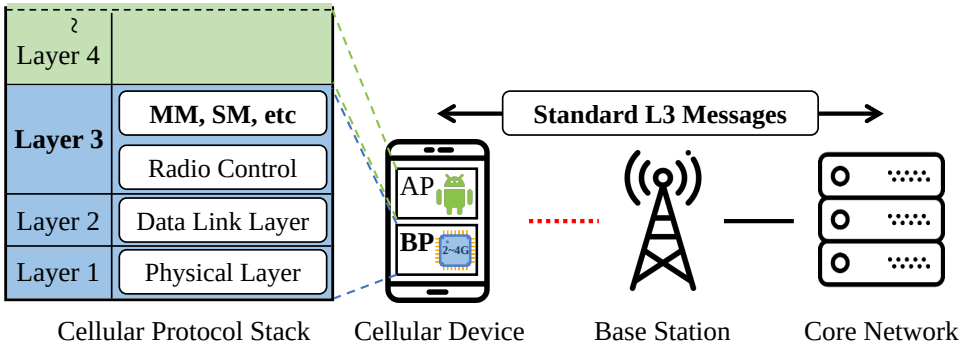


Figure 4.1: High-level overview of a cellular network architecture.

baseband firmware from the beginning (*i.e.*, binary loading), simple heuristics can accomplish the three steps of firmware structure analysis (*i.e.*, memory-layout analysis, function boundary identification, and target function detection). By systematizing these heuristics and integrating them into IDA Pro as plugins, we were able to identify ≈124 times more functions and successfully detect our target function (*i.e.*, L3 message decoder function) from the 18 firmware images we collected. After detecting the target function, we further analyzed its processing routine and discovered nine distinct bugs, including two *critical* 0-day vulnerabilities for remote code execution (RCE). We discovered a total of 78 functional bugs and 56 memory-related vulnerabilities from the 18 firmware images.

This chapter 1) demonstrates the efficacy of well-systematized heuristics for analyzing the firmware structure of complex IoT devices, such as baseband, and 2) summarizes how we discovered and systematized such heuristics.

4.1 Preliminaries for Cellular Baseband Analysis

4.1.1 Cellular Network Architecture

Figure 4.1 illustrates a high-level view of the cellular network architecture. Cellular networks are composed of mainly three components: cellular devices, base stations, and the core network. These components are referred to differently in each cellular generation. For example, NodeB, eNodeB, and gNodeB denote the 3G, 4G, and 5G base stations, respectively. In this paper, we will refer to them as generic terms for simplicity.

A cellular device is any device that is located at the network's edge and enables users to access cellular services. The most common cellular device is a smartphone. A cellular device usually has two separate processors: an application processor (AP), which runs mobile operating systems and user applications, and a cellular BP, which handles radio/digital signal processing. A base station provides cellular devices with a wireless connection. Through the radio interface, it transmits messages from the core network to a cellular device and vice versa. Thus, it is responsible for managing radio resources in order to improve service quality for users. A core network performs core procedures such as mobility and session management, as well as critical user identification and security services, such as encryption and integrity checks.

As with the OSI model, the cellular protocol stack is composed of multiple layers. The air interface of cellular networks is located between layers 1 and 2 of the OSI model, and is comprised of the physical layer (PHY) and the medium access control layer (MAC). Then, on layer 3, various core procedure messages are delivered. To properly manage these layers, each cellular device's baseband also implements this cellular protocol stack. Additionally, the latest 4G/5G cellular devices support backward compatibility with earlier 2G/3G cellular technologies for cell coverage and roaming.

Table 4.1: Cellular L3 protocols and their specification documents.

PD	Description	Abbrev.	Spec No.
0	Group Call Control	GCC	44.068
1	Broadcast Call Control	BCC	44.069
2	EPS Session Management	ESM	24.301
3	Call control; call related SS	CS	24.008
4	GPRS Transparent Transport Protocol	GTTP	44.018
5	Mobility Management	MM	24.008
6	Radio Resources Management	RR	44.018
7	EPS Mobility Management	EMM	24.301
8	GPRS Mobility Management	GMM	24.008
9	Short Message Service	SMS	24.011
10	GPRS Session Management	SM	24.008
11	non-call related Supplementary Services	SS	24.080
12	Location Services	LCS	23.271
14	Reserved for extension	-	-
15	Tests procedures	-	36.509

PD stands for the 4-bit protocol discriminator.

4.1.2 Cellular Layer 3 Protocols

Among various cellular protocols, layer 3 (L3) protocols are responsible for complex core functions, such as mobility management, session management, and even cryptographic operations to protect users' private information. These protocols are defined in the cellular specification documents. The L3 protocols are listed in Table 4.1, along with their protocol discriminators (PDs) and document numbers. Each specification document defines the details of messages in the corresponding protocol, such as message formats or directions. Due to the complexity of their implementations, numerous errors have been observed [72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83]. These L3 protocols are not limited to a single cellular generation, such as GSM or LTE, but rather span generations including several distinct protocols [102]. It is worth noting that an L3 message can be transmitted to a cellular device (*i.e.*, downlink) or to the core network (*i.e.*, uplink). In this paper, we only consider the downlink, as our analysis target is the baseband in a smartphone.

4.1.3 Baseband Processor and Software

A BP is a dedicated processor in a cellular device that is responsible for managing all radio functions associated with cellular communication, including digital signal processing. To meet the real-time requirements of radio communication, it runs a real-time operating system as its firmware. Therefore, its firmware operates as a single executable, and we refer to baseband firmware as a *baseband binary*.

Baseband software is typically proprietary, and manufacturers do not share detailed information about it, such as the source code. For instance, Qualcomm's Snapdragon, MediaTek's Helio, and Samsung's Exynos are the top three system-on-a-chip products that contain a BP [103]. None of these manufacturers, however, provide detailed information about their products. As a result, researchers perform reverse engineering on baseband software in order to analyze and identify security flaws [104, 105, 106, 107, 41]. Additionally, each baseband may have a unique architecture based on the design choice. For example, while Exynos and Mediatek are based on the ARM architecture, Snapdragon is based on their own architecture called Hexagon. Therefore, analyzing baseband requires a tool that is compatible with the target baseband architecture.

When baseband software processes L3 protocol messages, it first classifies the message's PD and identity. Then, using the message structure defined in the specification, it parses and decodes the message. After decoding the message, it takes an appropriate action on each piece of decoded data.

4.1.4 Challenges in Baseband Firmware Analysis

There are several challenges in analyzing baseband firmware. First, cellular baseband firmware remains largely unknown because vendors do not make its details public. The reason for this could be to safeguard their proprietary implementations and make them difficult to be analyzed. Notably, the security impact of baseband vulnerabilities is critical. Because baseband controls radio communication, its vulnerabilities can be remotely exploited using a software-defined radio (SDR). Therefore, this can have a detrimental effect on the users' privacy, as well as the vendors' finances or reputations. Consequently, any information about the baseband firmware remains obscure, such as its structure, memory layout, or initialization procedure. This obscurity significantly complicates firmware analysis, and analysis against it requires significant manual effort.

To reduce such manual efforts, one may use memory dumps [76, 77], which reflect the initialization steps for memory layout and include runtime information. However, this approach is not suitable in practice because obtaining memory dumps requires real devices, as well as a special feature (*e.g.*, a hidden dump menu available only on older Android devices) or vulnerability to trigger it. To obtain memory dumps, one may utilize a hardware debug interface, such as JTAG; however, such an interface is also disabled on recent devices. Therefore, memory dumps may not be suitable for scalable firmware analysis. Notably, even memory dumps require additional analysis to determine function boundaries and target functions, which is fundamental for subsequent security analysis.

Moreover, automating the baseband analysis is essential to achieve scalability and applicability; however, this is not a simple task. Researchers have focused on manual analysis to uncover the obscurity of baseband firmware [75, 76, 77]. However, this method is fundamentally limited in terms of scalability and applicability due to the extremely large size of baseband firmware (tens of MBs) and the presence of numerous non-trivial features to analyze, such as cryptographic operations. Without automation, it is nearly impossible to investigate numerous functions for hundreds of L3 protocol messages. Thus, previous research has heavily relied on a black-box-based dynamic analysis techniques, such as fuzzing, in conjunction with physical devices [72, 73, 74, 75]. These approaches, however, are insufficient because many baseband vulnerabilities are difficult to trigger dynamically owing to the convoluted states of L3 protocols. Additionally, these approaches rely on an explicit oracle to identify bugs, such as a program crash, limiting them to a few bug types. Even similar vulnerabilities frequently remain undetected.

Due to these difficulties, assessing the security of various baseband models or versions, even within a single vendor, remains an open research question. To conduct a scalable security analysis of numerous baseband devices, a scalable firmware analysis approach should be developed.

4.2 Scaling up Firmware Structure Analysis for Security Testing

To conduct a scalable security analysis of baseband software, it is necessary to first analyze its firmware structure. For this, there are mainly three requirements: 1) analyzing the memory layout on which the firmware will be loaded 2) identifying function boundaries in a stream of byte codes, and 3) detecting a target function among the numerous identified functions. If the analysis of the firmware structure satisfies these requirements, further analysis can be performed on the target function. Therefore, we focus on developing heuristics that satisfy these requirements.

For the cellular baseband, we focused on one of the top three mobile processor vendors [103], which we refer to as `Vendor1`. On the vendor's request, we have anonymized the device and vendor names in this thesis. To develop heuristics, we first analyze the firmware image for `Vendor1`'s most recent model, and then determine whether the heuristics developed are transferable to other device models and versions. Note that all baseband firmware images

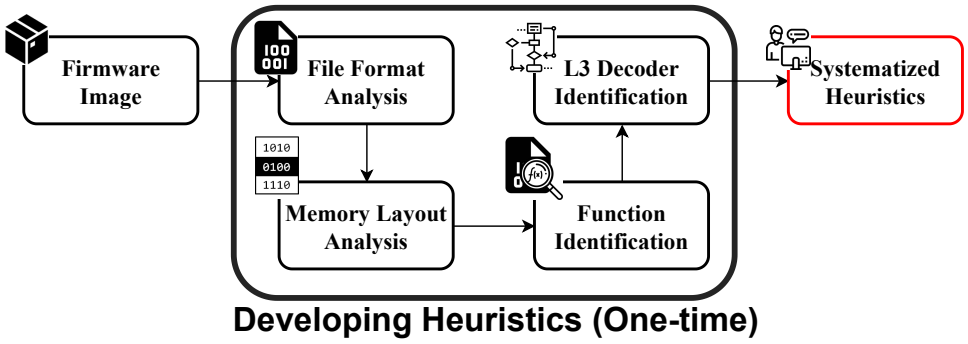


Figure 4.2: Overview of our approach to developing heuristics for baseband firmware analysis.

that we tested are based on the ARM architecture.

Among the various cellular protocols implemented in baseband software, we choose L3 protocol messages as our case study target. More precisely, we are interested in locating a decoder function for L3 protocol messages. As described in §4.1.2, these L3 protocol messages are critical for the cellular core procedures. Due to the complexity of the L3 protocol's logic and data structures, several vulnerabilities have been discovered in their implementations [76, 77, 78]. Therefore, we focus on the L3 protocol messages listed in Table 4.1.

Figure 4.2 illustrates an overview of our approach. Our approach is largely comprised of two components: manual firmware analysis and a fully automated system. The firmware analysis mainly explores where the message decoder are located. This is a manual, yet one-time task, as the decoding logic is rarely changed across device models or versions from the same vendor. The following section describes how we developed our heuristics that extract the L3 decoder function address from the target baseband binary.

4.3 Developing Heuristics by Uncovering Firmware's Obscurity

This section details our approach to uncover the obscurity of baseband firmware. To address this obscurity issue, manual analysis is required as described in §4.1.4; however, the analysis is a one-time task, and the results can be reused for multiple baseband models or versions. This section details our analysis of `Vendor1`'s baseband firmware. More precisely, we describe how we develop heuristics for analyzing file formats (§4.3.2), memory layouts (§4.3.3), identifying function boundaries (§4.3.4), and locating the L3 decoding function (§4.3.5). Table 4.2 summarizes our heuristics.

4.3.1 Collecting Firmware Images

We use 18 firmware images from `Vendor1` in this study. To obtain baseband firmware, there are primarily two methods: memory dumps and firmware images. Previous studies [76, 77] relied on memory dumps because they do not require complicated analyses for firmware initialization; memory dumps contain runtime memory states, a memory layout, and global variables. However, this method requires a physical device to dump memory; thus, it severely limits its scalability and applicability. Additionally, we discovered that in recent devices, the hidden menu for triggering a memory dump or a hardware debug interface, such as JTAG, has been disabled. On the other hand, there is a third-party website [108] that maintains a list of firmware images organized by product model and version. As a result, we chose to utilize a third-party website [108]. We selected 18 firmware images for the latest flagship models from various device models and versions, as listed in Table 4.3.

To begin, we analyzed a single image namely the most recent version of the most recent model (*Model A*

Table 4.2: Summary of heuristics for scalable firmware structure analysis of baseband.

Step	Challenges	Heuristics
❶ File format analysis	Single packed firmware file Unknown firmware format	Parse the file name and extract the baseband firmware Analyze 4-byte integers in the header and extract segments
❷ Memory layout analysis	Invalid memory references	Emulate the scatter-loading procedure
❸ Function identification	Numerous indirect function calls Co-Existence of the ARM and Thumb modes Remaining unknown functions	Build and scan signatures for function prologues Search odd-numbered pointers to Thumb-mode functions Leverage references to debug information
❹ Detecting L3 decoders	Numerous functions (over 90K) Relative addresses to access debug information	Search keywords from each function's debug information Compute the absolute addresses by analyzing program slices

in Table 4.3), in order to break down its obscurity and develop heuristics. Then, we systematize the heuristics and apply them to other firmware images.

4.3.2 Heuristics for File Format Analysis

Prior to analyzing the firmware using software analysis tools, its file format should be unveiled first. As smartphone firmware contains both Android operating system (which runs on an AP) and the baseband firmware (which runs on a BP), we extracted the baseband firmware from the downloaded firmware file. To analyze the baseband firmware's structure, we leveraged our experience with IoT binary analysis. We discovered that many IoT binaries contain a base address, size, or offset for internal segments, which are often represented as 4-byte integers. Based on this heuristic knowledge, we developed a simple heuristic that extracts useful information from the firmware's header by searching for 4-byte integers. As a result, we discovered that the baseband firmware contains a simple header containing multiple sets of the name, offset, size, and base address of each segment. Each segment of the firmware binary has a dedicated routine; for example, we discovered that one segment contains library functions, such as a customized memory-copy function. We extracted each segment based on the header information. Notably, previous research has taken a similar approach [76, 77].

4.3.3 Heuristics for Memory Layout Analysis

Next, we loaded the extracted segments of baseband firmware into IDA Pro [101] using the base addresses obtained from the file format analysis. Notably, IDA Pro is a cutting-edge binary analysis tool that automatically applies several static binary analysis techniques to a loaded binary, including function boundary identification, disassembly, and basic data flow analysis. Nevertheless, it was only able to identify 452 functions in our target firmware, which is 44 MB in size; a large portion of the firmware is not analyzed.

By analyzing this issue, we discovered that the firmware's memory layout is initialized at runtime. This initialization is critical for the firmware analysis; otherwise, data or function pointers in the firmware would point to invalid memory addresses, obstructing further analysis significantly. Indeed, when we opened the firmware using IDA, we discovered that the data or function pointers in the majority of functions attempted to access data or call functions that were located in invalid memory addresses. This has a significant impact on IDA's auto-analysis process, which analyzes addresses detected by pointers in order to identify additional functions.

We discovered that this invalid pointer issue is caused by *scatter-loading*. Notably, IDA was unable to handle scatter-loading. Scatter-loading is an loading mechanism in the ARM architecture that reallocates an initially loaded file into multiple memory regions at runtime. This technique is widely used in ARM-based embedded systems because it enables the compression of data regions, thereby reducing the firmware size. When building firmware, a component in the ARM compiler, named `armlink`, inserts functions for scatter-loading that initialize the firmware

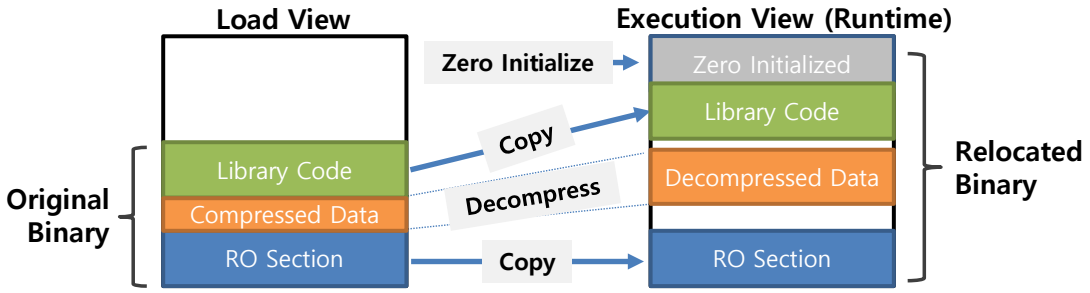


Figure 4.3: Example of scatter-loading.

at runtime. These functions copy, decompress, or zero-initialize memory regions according to a predefined table, as shown in Figure 4.3. As a result, if this scatter-loading is not handled, an entire binary file is loaded into a single continuous memory region, invalidating the data and function pointers. Notably, none of the existing approaches to binary analysis have taken this into account, although scatter-loading is a fundamental feature of ARM-based devices.

In order to address the scatter-loading issue and create a proper memory layout, we developed heuristics that emulate the scatter-loading process. Our heuristics, in particular, mimic the behavior of the scatter-loading functions by copying, decompressing, and zero-initializing memory regions. Because these scatter-loading functions are predefined by `armlink` in highly optimized forms, the majority of recent ARM embedded devices reuse their implementation; thus, these functions have identifiable patterns. As a result, our heuristics detect these signatures in a manner similar to that of IDA's FLIRT [109].

After detecting scatter-loading functions, our heuristics analyze their cross-references to determine the predefined scatter-loading table. This table contains information that indicates the execution sequence and parameters of the scatter-loading functions. More specifically, the table contains a list of tuples, each of which consists of a source address, a destination address, a size, and a scatter-loading function to be applied. Thus, our heuristics parse each tuple and emulate the designated scatter-loading process as stated in the table.

4.3.4 Heuristics for Function Boundary Identification

Although our scatter-loading emulation initializes the memory layout, the majority of the loaded firmware remains unknown. Identifying functions in the baseband firmware presents several difficulties. First, because the baseband firmware is essentially a real-time operating system, it contains numerous functions indirectly called, such as interrupt functions. As a result, IDA Pro's auto-analysis was unable to analyze them correctly. Additionally, baseband firmware is frequently based on the ARM architecture, which is notorious for its difficulty in disassembling. To identify functions in firmware, one must first disassemble its byte code in advance. However, disassembly of unknown bytes in the ARM architecture frequently yields incorrect results [110] due to the ARM architecture's dual instruction set support: ARM and Thumb. The ARM instruction set is the default mode, which executes 32-bit instructions; the Thumb instruction set, on the other hand, supports compact 16-bit instructions, which helps reduce code size. Because the same bytes can be disassembled in two different ways, direct disassembling would result in many incorrectly disassembled codes. There have been several approaches to identifying function boundaries [46, 47, 48, 49, 50, 51]; however, the majority of approaches have not considered binaries that contain both ARM and Thumb mode instructions. Furthermore, in the ARM architecture, switch tables can be embedded within a code segment or even between basic blocks, thereby making them difficult to distinguish from code instructions. Owing to these difficulties, disassembling and identifying the function boundaries of firmware binaries

that contain both ARM and Thumb mode instructions is not straightforward.

To tackle this challenge, we developed three simple heuristics that take advantage of 1) frequently occurring function prologues, 2) the characteristics of function pointers in the Thumb mode, and 3) any remaining debug information from the development stage. First, we develop a heuristic that sequentially scans the firmware binary for function prologue signatures. These prologue signatures include `PUSH` instructions in both ARM and Thumb modes, including `0xe92d` (*i.e.*, `PUSH.W` in the Thumb mode), `0xb500` (*i.e.*, `PUSH` in the Thumb mode), `0xe92d` (*i.e.*, `PUSH` in the ARM mode). Then, we search for those signatures. Here, we employ the linear-sweeping technique that was previously demonstrated to be effective in a study on binary disassembly [48]. If a match is found, it is analyzed in the instruction mode of the matching signature. Because the Thumb mode uses 16 bits whereas the ARM mode uses 32 bits, our heuristic first checks the Thumb mode instructions and then the ARM mode instructions. We discovered empirically that this approach results in fewer false positives. Additionally, in order to minimize false positives in signature-based matching, we developed a heuristic that verifies whether the detected candidate for a function prologue handles registers in the following manner: 1) it should push the `LR` register to the stack; 2) it should not push the `SP` and `PC` registers to the stack; and 3) it should contain at least one normal register, such as `R2`, `R3`, or `R4`. Using these heuristics, we were able to identify significantly more functions (≈71 times) than the IDA Pro's initial analysis.

After detecting function prologues, we further identify functions by analyzing function pointers in the data section. For this, we leverage the characteristics of the Thumb mode; function pointers to Thumb mode functions use odd-numbered addresses. In particular, the least significant bit of the pointer value is always `1`. As the majority of data is aligned with an even address, an odd-numbered address that points to a code section is highly likely a Thumb mode function pointer. Therefore, we can identify ≈2,500 more functions that are indirectly invoked via such pointers.

Lastly, we make use of any remaining debug information from the development stage, such as a logging message. Notably, IoT device developers often include debug information in their production binaries [111, 76, 77]. Note that this debug information is different from that added by compilers via the `-g` option, which is removed when a binary is stripped. Our intuition is that if an instruction candidate has an operand pointing to debug information, it should belong to a function; thus, we can eventually identify the function referencing the debug information. We discovered through manual analysis that the debug information in our target firmware has a distinct structure. Specifically, the structure, which starts with the magic value `DBT`, contains a debug message, as well as the file path and the line number of the file to which it refers. Therefore, we developed another heuristic that searches for all debug information in a baseband binary using the magic value (*i.e.*, `DBT`) and then analyzes function boundaries.

By employing the three aforementioned heuristics, we were able to identify approximately 80 times as many functions as the IDA Pro's initial analysis. After identifying the functions, we then re-run IDA Pro's auto-analysis. If a newly identified function calls other functions, IDA Pro's auto-analysis can be used to recursively identify them. Consequently, we were able to identify over 200 times the number of functions (≈90K functions) identified in the initial analysis.

4.3.5 Heuristics for Detecting Layer 3 Decoder Functions

Our ultimate goal in performing firmware analysis is to locate our target functions (*i.e.*, L3 protocol decoder functions) and further assess the security of their message processing logic. To identify the decoders, we utilize debug information from the development stage, as described in function boundary identification (§4.3.4). We chose this approach over other binary analysis techniques because debug information is commonly used in practice to locate a particular function in a stripped binary when analyzing embedded devices [111, 76, 77]. Baseband

firmware is stripped and extremely large (over 30MB) containing a large number of functions (over 90K), making it extremely difficult to discover a specific function. Therefore, we use debug information, particularly messages for logging, to reduce the analysis effort.

We first developed a heuristic that searches all debug information in the baseband binary. As described in §4.3.4, the debug information has a particular structure that starts with a magic value (*i.e.*, `DBT`); thus, we scanned this magic value from the baseband binary. Then, we analyzed functions that contain references to the relevant debug information. Specifically, the debug information contains a message along with the path and line number of the source file. As functions within the same layer or library may share a file path, we can classify each function based according to the source file path in its debug information. Then, we looked for L3-related functions in debug messages and file paths using keywords, such as `L3`, `SS`, `EMM`, or `NAS`. This approach is advantageous for searching for target functions among a large amount of debug information; approximately 100K cases exist.

Not all functions, however, have direct access to their debug information. During our analysis, we discovered several functions that make use of relative addresses to access debug information. These functions optimize the address of their debug information by computing it at runtime using a base address. To address this, we developed another heuristic that performs lightweight program slice analysis to correctly match debug information within each function. By following the instructions in a target function, this analysis computes the relative address of debug information and obtains its absolute address. After debug information is correctly mapped in each function, we search for functions related to message decoding using keywords such as `decode` or `codec`. Consequently, we were able to identify a decoder function that decodes all L3 protocol messages. Because L3 protocol messages follow a standardized structure [102], they can be handled by a single decoder. After detecting the decoder function, we could conduct further security analysis.

4.4 Evaluation

To evaluate the effectiveness of developed heuristics, we apply them to 18 baseband firmware images that we collected from a third-party website [108], as described in §4.3.1. As of June 2020, we downloaded the latest and oldest firmware images of `Vendor1`, which is one of the top three baseband chipset vendors. We chose the most recent and oldest images because they may exhibit the greatest differences. The most recent images were created within two months, while the oldest were created over four years. Therefore, analyzing them effectively demonstrates the applicability of our heuristics across various baseband versions and models. We ran all experiments on a server equipped with Intel Core i7-6700K running at 4.00 GHz, 64 GB DDR4 RAM, and a 2 TB SSD running Windows 10.

To implement the heuristics, we wrote a total of 1,303 LoC in Python. We used APIs in IDA Pro v7.4 [101] for systematizing the heuristics. We released our source code irrelevant to the vendor to aid in further research.[1] The following sections evaluate our heuristics' capability for identifying function boundaries and detecting target functions (*i.e.*, L3 decoder function). Then, we describe how we discovered 9 bugs from the L3 decoder.

4.4.1 Effectiveness of Heuristics in Baseband Firmware Analysis

To evaluate the effectiveness of the developed heuristics, we test the following questions:

- *Q*1: How do well-systematized heuristics perform better than the existing framework at identifying function boundaries?
- *Q*2: Are well-systematized heuristics effective at detecting target functions?

[1] https://github.com/SysSec-KAIST/BaseSpec

Table 4.3: Analysis results of 18 baseband firmware images from `Vendor1`.

Dataset	Model	Build Date	Size (MB)	# of Funcs Identified by IDA Pro		# of Detected Debug Info.	L3 Decoder Address
				w/o Heuristics	w/ Heuristics		
Latest Version	Model A	May/2020	44	452	91043	127266	0x4113ed5a
	Model B	May/2020	44	3601	89989	126975	0x4117e646
	Model C	May/2020	43.8	446	89893	126926	0x4114ca72
	Model D	Jun/2020	41.3	409	68279	96518	0x40e3f5aa
	Model E	Jun/2020	41.4	409	68621	97671	0x40e3ffca
	Model F	Apr/2020	41.7	417	68080	105733	0x412b6022
	Model G	Apr/2020	41.2	417	67392	104914	0x41226da6
	Model H	Apr/2020	37.4	388	61019	109290	0x4105bbe0
	Model I	Apr/2020	37	386	66663	143473	0x4100b0b4
Oldest Version	Model A	Apr/2019	43.4	457	89789	125284	0x411c03aa
	Model B	Feb/2019	43.3	450	88209	125582	0x4127b8ca
	Model C	Feb/2019	43.1	450	80268	110047	0x4124810e
	Model D	Mar/2018	41.2	402	67749	95796	0x40b469be
	Model E	Mar/2018	41.2	402	66866	94170	0x40b473b6
	Model F	Apr/2017	41.3	441	78709	127517	0x41324ae6
	Model G	Apr/2017	40.8	443	63844	100885	0x41296ede
	Model H	Apr/2016	37.1	376	61611	112231	0x410704e8
	Model I	Apr/2016	36.8	377	61714	112494	0x41019c00

We anonymized the model names upon the request of the vendor. The names are listed alphabetically ascending from the most recent.

- *Q*3: Is it possible to transfer the heuristics learned from a single firmware image to other device models and versions?

In this experiment, we compare the number of functions identified by naive IDA Pro to the number identified by IDA Pro equipped with our heuristics. The results are shown in Table 4.3. As illustrated in the fifth and sixth columns of the table, our heuristics significantly improved the IDA's performance in identifying functions, thereby supporting both Q1 and Q3. The average number of functions identified by naive IDA Pro (*i.e.*, without heuristics) was 595, whereas the average number of functions identified by IDA Pro with our heuristics was 73,874, or ≈124 times more functions. This result is largely due to our heuristics, which demonstrate their efficacy. It should be noted, however, that IDA Pro's auto-analysis is extremely beneficial. After heuristics identified new functions, IDA Pro continued its auto-analysis on each function, particularly for the code references in it, and recursively discovered more functions.

The systematized heuristics can be applied to firmware for other baseband models and versions without requiring additional analysis. In practice, IDA Pro equipped with the heuristics successfully analyzed the remaining latest models and versions listed in Table 4.3, thereby confirming Q3. The average time spent on the firmware analysis, including the IDA's auto-analysis, was approximately 2600 s.

After identifying function boundaries, our heuristics detected the L3 protocol decoder in each baseband binary. Recall that each baseband binary has a single universal decoder for all L3 protocols (§4.3.5). The last column in Table 4.3 represents the address of the detected L3 protocol decoder. This result demonstrates the capability and transferability of the heuristics for detecting the L3 decoder, thereby supporting Q2 and Q3, respectively.

4.4.2 Discovering Bugs from L3 Decoders

After detecting the L3 decoders, we manually analyzed the security of their message processing logic. The baseband typically has two distinct logics for processing received messages. When an L3 protocol message is passed to the decoder function, the message is parsed and the parsed data is passed to the corresponding handler function for processing. Therefore, bugs may exist in both decoder and the handler functions.

Figure 4.4 depicts a high-level overview of the message processing logic and the implications of various

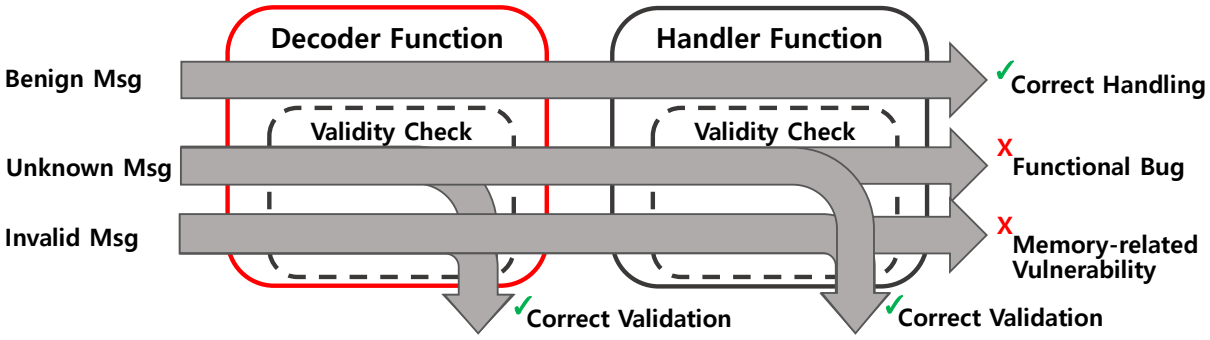


Figure 4.4: Procedure for processing L3 protocol messages. The colored box denotes the focus of our heuristics.

message types. According to their contents, input messages can be classified as benign, unknown, or invalid. We refer to messages that are not defined in the specification documents [102, 112, 113, 114, 115] as *unknown*. We refer to messages with a length field that differs from the one specified in the documents as *invalid*. Note that L3 protocol messages contain multiple elements, referred to as information elements (IEs). Each of these elements may include the following three fields: type, length, and value. The length field indicates the size of the value field in bytes. The length field may vary within a specified range in the documents.

Unknown and invalid messages should be discarded by the decoder function during its validity check routine. However, if developers implement such routines incorrectly or insufficiently, those messages are not discarded, resulting in functional bugs or memory-related vulnerabilities. For instance, if the decoder does not check the length field of an input message, this may result in memory corruption bugs in the handler function, such as a buffer overflow. An additional check may be included in a handler function. However, Analyzing such a routine in handler functions to determine the implications of unknown or invalid messages is not trivial because handler functions manage complex semantics, such as session/mobility management or call control. As a result, we rely heavily on manual analysis to understand the implications. It is worth noting, however, that our heuristics for detecting the decoder function significantly reduced the analysis effort; we analyzed only the functions associated with the decoder function among numerous (over 90K) functions.

To confirm functional bugs, we compare the baseband implementation to the cellular specification. We chose `Release 15` over the other releases of the specification because its has a freeze date of March 22, 2019. That is, the specification's development was complete and remained stable [116]. The next release, `Release 16`, has a freeze date of July 2020; thus, it is highly unlikely to be implemented in our target firmware images. Using these specification documents, we examined the message processing logic implemented in the baseband firmware. Consequently, we discovered several inconsistencies that break the baseband's compliance with the specification. We further investigated those inconsistencies and discovered 78 (5 unique) functional bugs. During this analysis, we also discovered 56 (4 unique) memory-related vulnerabilities, which could result in a denial of service or even remote code execution. Notably, these nine distinct bugs affect 33 distinct messages. We refer to these bugs from B1 to B9 in Table 4.4. Except for B7, all other bugs are newly discovered (*i.e.*, 0-days). We responsibly disclosed all bugs to the manufacturer (*i.e.*, `Vendor1`).

By comparing the bug discovery results of each firmware image, we discovered the following interesting points. When we compared the latest images with the oldest, we noticed that the majority of the discovered bugs had existed in the past. For instance, B8 and B9 are long-lived vulnerabilities from the oldest firmware in our dataset. It is highly likely that earlier models also have these vulnerabilities. Moreover, certain device models exhibit the same vulnerabilities. For example, Models *D*, *E*, *F*, and *G* all produce identical results, while Models *H* and *I* produce identical results; compare the results in the latest version row to those in the oldest version row.

Table 4.4: Bugs discovered from our manual analysis.

			Functional†					Memory-Related			
Dataset	Model	Build Date	B1	B2	B3	B4	B5	B6‡	B7	B8‡	B9
Latest Version	Model A	May/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model B	May/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model C	May/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model D	Jun/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model E	Jun/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model F	Apr/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model G	Apr/2020	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model H	Apr/2020	·	✓	✓	·	·	·	·	✓	✓
	Model I	Apr/2020	·	✓	✓	·	·	·	·	✓	✓
Oldest Version	Model A	Apr/2019	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model B	Feb/2019	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model C	Feb/2019	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model D	Mar/2018	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model E	Mar/2018	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model F	Apr/2017	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model G	Apr/2017	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model H	Apr/2016	·	✓	✓	·	·	·	✓	✓	✓
	Model I	Apr/2016	·	✓	✓	·	·	·	✓	✓	✓

We anonymized the model names upon the request of the vendor. The names are listed alphabetically ascending from the most recent.

† These have been confirmed to be non-compliant to the Release 15 specification.

‡ These are 0-days that allow for remote code execution.

Additionally, Models *C* and *B* produce identical results. This result implies that the manufacturer's codebase for those groups of device models may be identical or similar.

Furthermore, because the build dates of the oldest images span four years, we noticed at least two security changes have been made to the baseband implementation. This is because B6 appeared for the first time between April 2016 and April 2017, and B7 vanished between March 2018 and February 2018. By analyzing them, we determined that B6 was introduced as a result of changes to GMM handlers, and that B7 was removed as a result of the addition of security checks to EMM handlers. Meanwhile, Models *H* and *I* remained unchanged in their most recent versions, with the exception of B7. Due to the one-year build date difference between these two models, they were not affected by B6, a newly introduced bug from Model *G*.

4.5 Discussion

While the heuristics developed in this study are effective and promising, there are several areas for improvement. First, we focused on the cellular L3 protocol messages. Other cellular protocols, however, can be analyzed as well. By design, each message structure used in cellular protocols should be written in a consistent manner in protocol specifications [102]. Such a systematic structure implies that other protocols' decoder and message processing logic may be similar to that of the L3 protocols. Therefore, we believe that similar heuristics can be applied to other protocols, though adjusting the heuristics may require considerable effort.

Automating the discovery of bugs is another critical and promising area of research. We manually analyzed the message processing logic from the detected L3 decoder function to identify bugs. To reduce such manual analysis effort, one could employ other promising analysis techniques such as fuzzing [94, 95, 44] or hybrid analysis [96, 97]. Notably, dynamic analysis techniques can be combined with recent emulation-based approaches on baseband [78, 14, 15]. We reserve such promising improvements for future work.

Lastly, discovering other types of bugs, such as logical or stateful bugs, is another challenging research topic

in this field. It is well established that discovering service-related bugs, such as bypassing security channels [79], is extremely difficult due to the baseband's massive code for numerous cellular protocols. Moreover, these protocols have convoluted states, necessitating the consideration of various stateful information in the analysis. It is is also not trivial to construct a reference for logical bugs from the specifications [117, 118, 119, 120, 121]. To overcome these challenges, one can employ conformance testing [122, 123, 124] or natural language processing techniques [125] to analyze stateful information from the specification. By introducing our heuristics as a starting point for future research in this field, we encourage further investigation.

4.6 Related work

Several studies have focused on assessing the security of cellular network protocols and baseband software. In the early stage, researchers conducted dynamic analysis on a physical device via the over-the-air interface without directly analyzing the baseband firmware. They leveraged open-source cellular stacks [126, 127, 128, 129] and low-cost software-defined radios (SDRs) [130, 131] to access the over-the-air interface. By sending crafted messages to target devices, they analyzed the L3 protocols related to SMS or cell broadcast messages [72, 73, 74], service quality [82, 132], and private information [133, 83]. Recent approaches [134, 81, 80, 79] have attempted to reduce manual efforts involved in dynamic analysis by leveraging abnormal messages. Several other studies took a similar approach, focusing on various layers, protocols, or domains of cellular networks, such as VoLTE [135], SS7/Diameter [136], uplink messages [79, 137], or lower layers [138, 139, 140]. These approaches are advantageous because they do not require an understanding of baseband firmware. They do, however, require physical devices and a thorough understanding of cellular networks and specifications. Additionally, they require a testbed that operates the same as the real environment. Therefore, testing each implemented message across multiple devices is nearly infeasible.

On the other hand, several studies [75, 76, 77] have focused on analyzing baseband firmware for L3 protocols, such as the one we are analyzing. Weinmann [75] showed a practical approach for analyzing memory-related bugs in GSM protocol stacks in baseband using the JTAG debug interface. Golde *et al*. [76] and Cama [77] used memory dumps to analyze recent Exynos firmware. Notably, they discovered RCE 0-days and were rewarded at Mobile Pwn2Own. While these approaches provide promising insights into baseband analysis, they are limited by the fact that they require physical devices to dump memory. To address this limitation, Maier *et al*. [78] recently proposed an emulation-based analysis of the RRC/EMM protocols. They manually analyzed MediaTek firmware and hooked all related functions to emulate functions related to those protocols. They then ran a famous fuzzer, AFL++ [141], to analyze vulnerabilities. These approaches, however, suffer from the fundamental problem of baseband firmware, namely, obscurity and complexity. We address these issues by thoroughly analyzing baseband firmware and developing heuristics that can be transferable to other baseband models or versions. Consequently, by systematizing heuristics, we were able to quickly analyze 18 baseband firmware images (≈2,500 s).

4.7 Conclusion

While security analysis techniques have advanced significantly, they cannot be applied directly to complex IoT devices, such as the smartphone baseband, owing to their obscurity and complexity. Several promising approaches to baseband analysis have been proposed [72, 73, 74, 75, 76, 77, 78], but these approaches are still too early to address the fundamental problem of firmware analysis.

To address this issue, we conducted the first systematic analysis of cellular baseband. During the analysis, we discovered that simple heuristics can satisfy the minimum requirements for firmware analysis when it comes to

assessing its security. We developed heuristics that identify function boundaries and locate target functions. These heuristics significantly improved the performance of the leading binary analysis framework, IDA Pro, in identifying function boundaries, resulting in the discovery of 78 functional bugs and 56 memory-related vulnerabilities, including critical RCE 0-days.

We believe that accumulating heuristic knowledge is the last-mile effort towards resolving the fundamental issues in the firmware analysis of complex IoT devices. If such heuristics are transferrable to a variety of device models and versions, they can enable scalable firmware analysis, as demonstrated by our successful analysis of 18 smartphone baseband firmware images. In this regard, we argue that systematizing and accumulating heuristic workarounds will eventually facilitate scalable security analysis of the IoT ecosystem.

Chapter 5. Systematic Study of Binary Code Similarity Analysis

We have demonstrated in the previous two chapters that simple heuristics are effective and necessary for successful firmware emulation and structure analysis of IoT devices, hence enabling scalable security testing. We have discovered that, despite the diversity of the IoT ecosystem, many devices share a similar codebase, and thereby, heuristics developed for one device are transferrable across various device versions or families. Meanwhile, such code sharing is a serious issue in the IoT ecosystem. Numerous IoT devices share similar/identical vulnerabilities and are continually exploited by them [7, 8, 9, 10]. To rapidly assess such vulnerabilities and secure the IoT ecosystem, we can also leverage binary code similarity, a technique referred to as known vulnerability analysis. Prior to delving into the development of known vulnerability analysis techniques, we conducted a comprehensive investigation on its fundamental, BCSA, which is detailed in this chapter.

5.1 Motivation and Overview

Programmers reuse existing code to build new software. It is a common practice for them to find the source code from another project and repurpose that code for their own needs [142]. Inexperienced developers even copy and paste code samples from the Internet to ease the development process. This trend has deep implications for software security and privacy. When a programmer takes a copy of a buggy function from an existing project, the bug will remain intact even after the original developer has fixed it. Furthermore, if a developer in a commercial software company inadvertently uses a library code from an open-source project, the company can be accused of violating an open-source license such as the GNU General Public License (GPL) [143].

Unfortunately, however, detecting such problems from binary code using a similarity analysis is *not* straightforward, particularly when the source code is not available. This is because binary code lacks high-level abstractions, such as data types and functions. For example, it is not obvious to determine from binary code, whether a memory cell represents an integer, a string, or another data type. Moreover, identifying precise function boundaries is radically challenging in the first place [47, 46].

Therefore, measuring the similarity between binaries has been an essential research topic in many areas such as malware detection [65, 66], plagiarism detection [67, 68], authorship identification [69], and vulnerability discovery [53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64].

However, despite the surging research interest in binary code similarity analysis (BCSA), we found that it is still significantly challenging to conduct new research on this field for several reasons. First, most of the methods focus only on the end results without considering the precise reasoning behind their approaches. For instance, during our literature study in the field, we observed that there is a prominent research trend in applying BCSA techniques to cross-architecture and cross-compiler binaries of the same program [62, 55, 59, 58, 54, 70, 56]. Those approaches aim to measure the similarity between two or more seemingly distinct binaries generated from different compilers targeting different instruction sets. To achieve this, multiple approaches have devised complex analyses based on machine learning to extract the semantics of the binaries, assuming that their semantics should not change across compilers or target architectures. However, none of the existing approaches clearly justifies the necessity of such complex semantics-based analyses. One may imagine that a compiler may generate structurally similar binaries for different architectures, even though they are syntactically different. Do compilers and architectures really matter for BCSA in this regard? Unfortunately, it is difficult to answer this question as most of the existing approaches leverage **uninterpretable** machine learning techniques [56, 71, 55, 144, 62, 145, 63, 64, 146, 147, 148, 149].

Further, it is not even clear why a BCSA algorithm works only on some benchmarks and not on others.

Second, every existing paper on BCSA that we studied utilizes its own benchmark to evaluate the proposed technique, which makes it difficult to compare the approaches with one another. Moreover, reproducing the previous results is often infeasible because most researchers reveal neither their source code nor their dataset. Only 10 of the 43 papers that we studied fully released their source code and *only two* of them opened their entire dataset. Finally, researchers in this field do not use unified terminologies and often miss out on critical citations that appeared in top-tier venues of other fields. Some of them even mistakenly use the same technique without citing the previous literature properly. These observations motivate one of our research goals, which is to summarize and review widely adopted techniques in this field, particularly in terms of generating features.

To address these problems, we take a step back from the mainstream and contemplate fundamental research questions for BCSA. As the first step, we precisely define the terminologies and categorize the features used in the previous literature to unify terminologies and build knowledge bases for BCSA. We then construct a comprehensive and reproducible benchmark for BCSA to help researchers extend and evaluate their approaches easily. Lastly, we design an interpretable feature engineering model and conduct a series of experiments to investigate the influence of compilers, their options, and their target architectures on the syntactic and structural features of the resulting binaries.

Our benchmark, which we refer to as BINKIT, encompasses various existing benchmarks. It is generated by using major compiler options and targets, which include 8 architectures, 9 different compilers, 5 optimization levels, as well as various other compiler flags. BINKIT contains 243,128 distinct binaries and 36,256,322 functions built for 1,352 different combinations of compiler options, on 51 real-world software packages. We also provide an automated script that helps extend BINKIT to handle different architectures or compiler versions. We believe this is critical because it is not easy to modify or extend previous benchmarks, despite us having their source codes. Cross-compiling software packages using various compiler options is challenging because of numerous environmental issues. To the best of our knowledge, BINKIT is the first *reproducible* and *extensible* benchmark for BCSA.

With our benchmark, we perform a series of rigorous studies on how the way of compilation can affect the resulting binaries in terms of their syntactic and structural shapes. To this end, we design a simple *interpretable* BCSA model, which essentially computes relative differences between BCSA feature values. We then build a BCSA tool that we call TIKNIB, which employs our interpretable model. With TIKNIB, we found several misconceptions in the field of BCSA as well as novel insights for future research as follows.

First, the current research trend in BCSA is founded on a rather exaggerated assumption: binaries are radically different across architectures, compiler types, or compiler versions. However, our study shows that this is not necessarily the case. For example, we demonstrate that simple numeric features, such as the number of incoming/outgoing calls in a function, are largely similar between binaries compiled across different architectures. We also present other elementary features that are robust (*i.e.*, have a small effect on the performance of BCSA) across compiler types, compiler versions, and even intra-procedural obfuscation. With these findings, we show that TIKNIB with those simple features can achieve comparable accuracy to that of the state-of-the-art BCSA tools, such as VulSeeker, which relies on a complex deep learning-based model.

Second, most researchers focus on vectorizing features from binaries, but not on recovering lost information during the compilation, such as variable types. However, our experimental results suggest that focusing on the latter can be highly effective for BCSA. Specifically, we show that TIKNIB with recovered type information achieves an accuracy of over 99% on all our benchmarks, which was indeed the best result compared to all the existing tools we studied. This result highlights that recovering type information from binaries can be as critical as developing a novel machine learning algorithm for BCSA.

Finally, the interpretability of a tool helps not only deeply understand BCSA results but also advance the field. For example, we present several practical issues in the underlying binary analysis tool, *i.e.*, IDA Pro, which is used by TIKNIB, and discuss how such errors can affect the performance of BCSA. Since our benchmark has the ground truth and our tool employs an interpretable model, we were able to easily pinpoint those fundamental issues, which will eventually benefit binary analysis tools and the entire field of binary analysis.

In summary, our contributions of this study are as follows:

- We study the features and benchmarks used in previous research on BCSA and clarify less-explored research questions in this field.
- We introduce BINKIT,[1] the first reproducible and expandable BCSA benchmark. It contains 243,128 binaries and 36,256,322 functions compiled for 1,352 distinct combinations of compilers, compiler options, and target architectures.
- We develop a BCSA tool, TIKNIB,[2] which employs a simple interpretable model. We demonstrate that TIKNIB can achieve an accuracy comparable to that of a state-of-the-art deep learning-based tool. We believe this will serve as a baseline to evaluate future research in this field.
- We investigate the efficacy of basic BCSA features with TIKNIB on our benchmark and unveil several misconceptions and novel insights.
- We make our source code, benchmark, and experimental data publicly available to support open science.

In the following, we start by describing the fundamentals of BCSA.

5.2 Binary Code Similarity Analysis

Binary Code Similarity Analysis (BCSA) is the process of identifying whether two given code snippets have similar semantics. Typically, it takes in two code snippets as input and returns a similarity score ranging from 0 to 1, where 0 indicates the two snippets are completely different, and 1 means that they are equivalent. The input code snippet can be a function [150, 54, 151, 62, 59, 152, 144, 64], or even an entire binary image [67, 68]. Additionally, the actual comparison can be based on functions, even if the inputs are entire binary images [71, 58, 153, 55, 154, 155, 56].

At a high level, BCSA performs four major steps as described below:

- **(S1) Syntactic Analysis**: Given a binary code snippet, one parses the code to obtain a disassembly or an Abstract Syntax Tree (AST) of the code, which is often referred to as an Intermediate Representation (IR) [156]. This step corresponds to the syntax analysis in traditional compiler theory, where source code is parsed down to an AST. If the input code is an entire binary file, we first parse it based on its file format and splits it into sections.
- **(S2) Structural Analysis** This step analyzes and recovers the control structures inherent in the given binary code, which is not readily available from the syntactic analysis phase (S1). In particular, this step involves recovering the control-flow graphs (CFGs) and call graphs (CGs) in the binary code [157, 158]. Once the control-structural information is obtained, one can use any attribute of these control structures as a feature. We distinguish this step from semantic analysis (S3) because binary analysis frameworks typically provide CFGs and CGs for free; the analysts do not have to write a complex semantic analyzer.
- **(S3) Semantic Analysis** Using the control-structural information obtained from S2, one can perform traditional program analyses, such as data-flow analysis and symbolic analysis, on the binary to figure out the underlying

[1] https://github.com/SoftSec-KAIST/binkit
[2] https://github.com/SoftSec-KAIST/tiknib

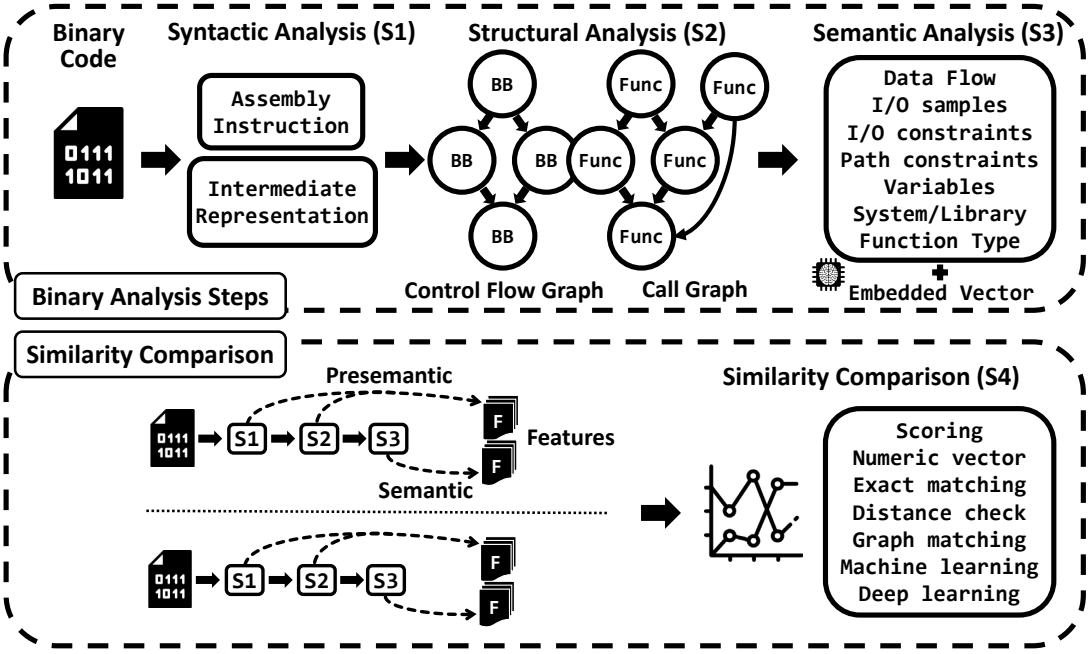


Figure 5.1: Typical workflow of binary code similarity analysis (BCSA).

semantics. In this step, one can generate features that represent sophisticated program semantics, such as how register values flow into various program points. One can also enhance the features gathered from S1–S2 along with the semantic information.

- **(S4) Vectorization and Comparison** The final step is to vectorize all the information gathered from S1–S3 to compute the similarity between the binaries. This step essentially results in a similarity score between 0 and 1.

Figure 5.1 depicts the four-step process. The first three steps determine the inputs to the comparison step (S4), which are often referred to as *features*. Some of the first three steps can be skipped depending on the underlying features being used. The actual comparison methodology in S4 can also vary depending on the BCSA technique. For example, one may compute the Jaccard distance [159] between feature sets, calculate the graph edit distance [160] between CFGs, or even leverage deep learning algorithms [161, 162]. However, *as the success of any comparison algorithm significantly depends on the chosen features, this chapter focuses on features used in previous studies rather than the comparison methodologies.*

In this section, we first describe the features used in the previous papers and their underlying assumptions (§5.2.1). We then discuss the benchmarks used in those papers and point out their problems (§5.2.2). Lastly, we present several research questions identified during our study (§5.2.3).

Our study focuses on recent papers that appeared in top-tier venues to keep the scope manageable. There are, of course, plentiful research papers in this field, all of which are invaluable. Nevertheless, *our focus here is not to conduct a complete survey on them but to introduce a prominent trend and the underlying research questions in this field, as well as answer those questions.* Because of the space limit, we excluded papers [163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173] that were published before 2014 and those not regarding top-tier venues, or binary diffing tools [174, 175, 176] used in industry. Additionally, we excluded papers that aim to address a specific research problem such as malware detection, library function identification, or patch identification. Although our study focuses only on recent papers, we found that the features we studied in this thesis are indeed general enough; they cover most of the features used in the older papers.

5.2.1 Features Used in Prior Works

We categorize features into two groups based on when they are generated during BCSA. Particularly, we refer to features obtained before and after the semantic analysis step (S3) as *presemantic features* and *semantic features*, respectively. Presemantic features can be derived from either S1 or S2, and semantic features can be derived from S3. We summarize both features used in the recent literature in Table 5.1.

(1) Presemantic Features

Presemantic features denote direct or indirect outcomes of the syntactic (S1) and structural (S2) analyses. Therefore, we refer to any attribute of binary code, which can be derived without a semantic analysis, as a presemantic feature. We can further categorize presemantic features used in previous literature based on whether the feature represents a number or not. We refer to features representing a number as *numeric presemantic features*, and others as *non-numeric presemantic features*. The first half of Table 5.1 summarizes them.

(a) Numeric Presemantic Features

Counting the occurrences of a particular property of a program is common in BCSA as such numbers can be directly used as a numeric vector in the similarity comparison step (S4). We categorize numeric presemantic features into three groups based on the granularity of the information required for extracting them.

First, many researchers extract numeric features from each basic block of a target code snippet. One may measure the frequency of raw opcodes (mnemonics) [191, 60] or grouped instructions based on their functionality (e.g., arithmetic, logical, or control transfer) [54, 148]. This numeric form can also be post-processed through machine learning [71, 55, 56, 148], as we further discuss in §5.2.1.

Similarly, numeric features can be extracted from a CFG as well. CFG-level numeric features can also reflect structural information that underlies a CFG. For example, a function can be encoded into a numeric vector, which consists of the number of nodes (i.e., basic blocks) and edges (i.e., control flow) as well as grouped instructions in its CFG [54, 181, 148]. One may extend such numeric vectors by adding extra features such as the number of successive nodes or the betweenness centrality of a CFG [71, 55, 148]. The concept of 3D-CFG [192], which places each node in a CFG into a 3D space, can be utilized as well. Here, the distances among the centroids of two 3D-CFGs can represent their similarity score [61]. Other numeric features can be the graph energy, skewness, or cyclomatic complexity of a CFG [191, 60, 148]. Even loops in a CFG can be converted into numeric features by counting the number of loop headers and tails as well as the number of forward/backward edges [185].

Finally, previous approaches utilize numeric features obtained from CGs. We refer to them as a CG-level numeric feature. Most of these approaches measure the number of callers and callees in a CG [193, 54, 71, 191, 60, 185, 62, 148]. When extracting these features, one can selectively apply an inter-procedural analysis using the ratio of the in-/out- degrees of the internal callees in the same binary and the external callees of imported libraries [58, 61, 63, 148]. This is similar to the coupling concept [194], which analyzes the inter-dependence between software modules. The extracted features can also be post-processed using machine learning [62].

(b) Non-Numeric Presemantic Features

Program properties can also be directly used as a feature. The most straightforward approach involves directly comparing the raw bytes of binaries [66, 195, 173]. However, people tend to not consider this approach because byte-level matching is not robust compared to simple code modifications. For example, anti-malware applications typically make use of manually written signatures using regular expressions to capture similar, but syntactically different malware instances [196]. Recent approaches have attempted to extract semantic meanings from raw binary code utilizing a deep neural network (DNN) to build a feature vector representation [62, 145].

Table 5.1: Summary of the features used in previous studies.

		2014						2015	2016							2017										2018										2019						2020				
		TEDEM	Tracy	CoP	LoPD	BLEX	BinClone	Multi-k-MH	discovRE	Genius	Esh	BinGo	MockingBird	Kam1n0	BinDNN	BinSign	Xmatch	Gemini	GitZ	BinSim	BinSequence	IMF-sim	CACompare	ASE17	BinArm	SANER18	BinGo-E	WSB	BinMatch	MASES18	Zeek	FirmUp	αDiff	VulSeeker	InnerEye	Asm2Vec	SAFE	BAR19i	BAR19ii	FuncNet	DeepBinDiff	ImpOpt	ACCESS20	Patchecko	BinKit
		[53]	[177]	[67]	[68]	[150]	[178]	[70]	[54]	[71]	[179]	[58]	[153]	[152]	[180]	[181]	[59]	[55]	[182]	[183]	[154]	[151]	[155]	[184]	[60]	[185]	[61]	[186]	[187]	[145]	[188]	[57]	[62]	[56]	[144]	[63]	[64]	[146]	[149]	[189]	[147]	[190]	[173]	[148]	★
Presemantic	BB-level Numbers	·	·	·	·	·	·	·	○	◑	·	·	·	·	·	·	·	◑	·	·	·	·	·	·	○	·	·	·	·	·	·	·	◑	·	·	·	·	·	◑	·	·	·	◑	○
	CFG-level Numbers	·	·	·	·	·	○	·	○	◑	·	·	·	·	·	○	·	◑	·	·	○	·	·	·	○	○	○	·	·	·	·	·	·	◑	·	·	·	·	◑	·	·	·	◑	○
	CG-level Numbers	·	·	·	·	·	·	·	○	·	·	○	·	·	·	○	·	·	·	·	·	·	·	·	○	○	○	·	·	·	·	·	○	·	·	◑	·	·	·	·	·	·	·	◑	○
	Raw Bytes	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	◑	·	·	◑	·	·	·	·	·	·	·	·	○	·	·
	Instructions	○	○	·	·	·	○	·	·	·	·	○	·	○	○	·	·	·	·	·	○	·	·	·	·	·	○	·	·	·	·	·	·	·	◑	◑	◑	◑	◑	·	◑	◑	·	·	·
	Functions	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·
Semantic	Symbolic Constraints	·	·	○	○	·	·	·	·	·	○	·	·	·	·	·	○	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	I/O Samples	·	·	·	○	·	·	○	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Runtime Behavior	·	·	·	·	○	·	·	·	·	·	·	○	·	·	·	·	·	·	○	·	○	○	○	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	◑	·
	Manual Annotation	·	·	·	·	·	·	·	·	·	·	○	·	·	·	○	·	·	·	○	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Program Slices, PDG	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	○	○	·	◑	·	·	·	·	·	·	·	·	·	·	·
	Recovered Variables	·	·	·	·	·	·	·	○	◑	·	·	·	·	·	○	·	◑	·	·	·	·	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	◑	·	·	·	◑	·
	Embedded Vector	·	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	○	·	·	·	·	·	·	·	·	·	·	·	○	·	·	○	○	○	○	○	○	○	○	○	○	·	○	·

◑ This mark denotes a feature that is not directly used for similarity comparison but is required for extracting other features used in post-processing.

Another straightforward approach involves considering the opcodes and operands of assembly instructions or their intermediate representations [197, 61]. Researchers often normalize operands [154, 177, 152] because their actual values can significantly vary across different compiler options. Recent approaches [182, 190] have also applied re-optimization techniques [198] for the same reason. To compute a similarity score, one can measure the number of matched elements or the Jaccard distance [58] between matched groups, within a comparison unit such as a sliding window [178], basic block [154], or tracelet [177]. Here, a tracelet denotes a series of basic blocks. Although these approaches take different comparison units, one may adjust their results to compare two procedures, or to find the longest common subsequence [154, 152] within procedures. If one converts assembly instructions to a static single assignment (SSA) form, s/he can compute the tree edit distance between the SSA expression trees as a similarity score [53]. Recent approaches have proposed applying popular techniques in natural language processing (NLP) to represent an assembly instruction or a basic block as an embedded vector, reflecting their underlying semantics [144, 63, 64, 146, 147, 149].

Finally, some features can be directly extracted from functions. Such features may include the names of imported functions, and the intersection of two inputs can show their similarity [181, 62]. Note that these features can collaborate with other features as well.

(2) Semantic Features

We call features that we can obtain from the semantic analysis phase (S3) *semantic features*. To obtain semantic features, a complex analysis, such as symbolic execution [67, 68, 58, 183, 61], dynamic evaluation of code snippets [68, 150, 153, 155, 183, 151, 187, 186, 184], or machine learning-based embedding [56, 71, 55, 144, 62, 145, 63, 64, 146, 147, 148, 149] is necessary. There are mainly seven distinct semantic features used in the previous literature, as listed in Table 5.1. It is common to use multiple semantic features together or combine them with presemantic features.

First, one straightforward method to represent the semantics of a given code snippet is to use symbolic constraints. The symbolic constraints could express the output variables or states of a basic block [67], a program slice [179, 59, 183], or a path [68, 199, 200]. Therefore, after extracting the symbolic constraints from a target comparison unit, one can compare them using an SMT solver.

Second, one may represent code semantics using I/O samples [68, 58, 61, 70]. The key intuition here is that two identical code snippets produce consistent I/O samples, and directly comparing them would be time-efficient. One can generate I/O samples by providing random inputs [68, 70] to a code snippet, or by applying an SMT solver to the symbolic constraints of the code snippet [58, 61]. One can also adopt inter-procedural analysis to precisely model I/O samples if the target code includes a function call [58, 61].

Third, the runtime behavior of a code snippet can directly express its semantics, as presented by traditional malware analysis [201]. By executing two target functions with the same execution environment, one can directly compare the executed instruction sequences [184] or visited CFG edges of the target functions [186]. For comparison, one may focus on specific behaviors observed during the execution [150, 151, 155, 61, 187, 202, 148]: the read/write values of stack and heap memory, return values from function calls, and invoked system/library function calls during the executions. To extract such features, one may adopt fuzzing [151, 203], or an emulation-based approach [187]. Moreover, one can further check the call names, parameters, or call sequences for system calls [153, 155, 187, 183, 61].

The next category is to manually annotate the high-level semantics of a program or function. One may categorize library functions by their high-level functionality, such as whether the function manipulates strings or whether it handles heap memory [58, 61, 181]. Annotating cryptographic functions in a target code snippet [204] is also helpful because its complex operations hinder analyzing the symbolic constraints or behavior of the code [183].

The fifth category is extracting features from a program slice [205] because they can represent its data-flow semantics in an abstract form. Specifically, one can slice a program into a set of strands [182, 57]. Here, a strand is a series of instructions within the same data flow, which can be obtained from backward slicing. Next, these strands can be canonicalized, normalized, or re-optimized for precise comparison [182, 57]. Additionally, one may hash strands for quick comparison [188] or extract symbolic constraints from the strands [179]. One may also extract features from a program dependence graph (PDG) [206], which is essentially a combination of a data-flow graph and CFG, to represent convoluted semantics of the target code, including its structural information [56].

Recovered program variables can also be semantic features. For example, one can compare the similarity of string literals referenced in code snippets [54, 71, 55, 181, 191, 60, 185, 148]. One can also utilize the size of local variables, function parameters, or the return type of functions [54, 181, 148, 189]. One can further check registers or local variables that store the return values of functions [61].

Recently, several approaches have been utilizing embedding vectors, adopting various machine learning techniques. After building an attributed control-flow graph (ACFG) [71], which is a CFG containing numeric presemantic features in its basic blocks, one can apply spectral clustering [207] to group multiple ACFGs or popular encoding methods [208, 209, 210] to embed them into a vector [55]. The same technique can also be applied to PDGs [56]. Meanwhile, recent NLP techniques, such as Word2Vec [211] or convolutional neural network models [212], can be utilized for embedding raw bytes or assembly instructions into numeric vectors [144, 62, 145, 63, 64, 146, 147, 149]. For this embedding, one can also consider a higher-level granularity [144, 63] by applying other NLP techniques, such as sentence embedding [213] or paragraph embedding [214]. Note that one may apply machine learning to compare embedding vectors rather than generating them [180, 188], and Table 5.1 does *not* mark them to use embedded vectors.

(3) Key Assumptions from the Past Research

During our literature study, we found that most of the approaches highly rely on semantic features extracted in (S3), assuming that they should not change across compilers or target architectures. However, none of them clearly justifies the necessity of such complex semantics-based analyses. They focus only on the end results without considering the precise reasoning behind their approaches.

This is indeed the key motivation for our research. Although most existing approaches focus on complex analyses, there may exist elementary features that we have overlooked. For example, there may exist effective presemantic features, which can beat semantic features regardless of target architectures and compilers. It can be the case that those known features have not been thoroughly evaluated on the right benchmark as there has been no comprehensive study on them.

Furthermore, existing research assumes the correctness of the underlying binary analysis framework, such as IDA Pro [101], which is indeed the most popular tool used as shown in the rightmost column of Table 5.2. However, CFGs derived from those tools may be inherently wrong. They may miss some important basic blocks, for instance, which can directly affect the precision of BCSA features.

Indeed, both (S1) and (S2) are challenging research problems by themselves: there are abundant research efforts to improve the precision of both analyses. For example, disassembling binary code itself is an undecidable problem [48], and writing an efficient and accurate binary lifter is significantly challenging in practice [156, 215]. Identifying functions from binaries [46, 47, 48, 49, 50, 51] and recovering control-flow edges [216] for indirect branches are still an active research field. All these observations lead us to research questions in §5.2.3.

Table 5.2: Summary of the datasets used in previous studies.

		#Binaries*		Architecture								Optimization					Compiler†										Extra				Info.		
Year	Tool [Paper]	Packages	Firmware	x86	x64	arm	aarch64	mips	mips64	mipseb	mips64eb	O0	O1	O2	O3	Os	GCC 3	GCC 4	GCC 5	GCC 6	GCC 7	GCC 8	Clang 3	Clang 4	Clang 5	Clang 6	Clang 7	misc.	Total #	Noinline	PIE	LTO	Obfus.	Code	Dataset	IDA
2014	TEDEM [53]	14	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	Tracy [177]	(115)	·	△	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
	CoP [67]	(214)	·	△	·	·	·	·	·	·	·	○	○	○	○	○	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	·	○	·	·	○
	LoPD [68]	48	·	○	·	·	·	·	·	·	·	○	○	○	○	○	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	○	○	·	·	·
	BLEX [150]	1,140	·	·	○	·	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	1	·	·	·	·	1	3	·	·	·	·	·	·	○
	BinClone [178]	90	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
2015	Multi-k-MH [70]	60	6	○	·	○	·	○	·	·	·	○	○	○	○	·	·	2	·	·	·	·	1	·	·	·	·	·	3	·	·	·	·	·	·	○
2016	discovRE [54]	593	3	○	·	○	·	○	·	·	·	○	○	○	○	○	·	1	·	·	·	·	1	·	·	·	·	2	4	○	·	·	·	·	◑	○
	Genius [71]	(7,848)	8,128	○	·	○	·	○	·	·	·	○	○	○	○	·	·	2	·	·	·	·	1	·	·	·	·	·	3	·	·	·	·	·	·	○
	Esh [179]	(833)	·	·	○	·	·	·	·	·	·	·	·	·	·	·	·	3	·	·	·	·	2	·	·	·	·	2	7	·	·	·	·	○	○	○
	BinGo [58]	(5,143)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	3	·	·	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	MockingBird [153]	80	·	○	·	○	·	○	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	·	·	·	○
	Kam1n0 [152]	96	·	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	○
	BinDNN [180]	(2,072)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	·	·	·	·	·	1	2	·	·	·	·	·	·	○
2017	BinSign [181]	(31)	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	2	2	·	·	·	○	·	·	○
	Xmatch [59]	72	1	○	·	·	·	○	·	·	·	·	·	·	·	·	·	2	·	·	·	·	1	·	·	·	·	·	3	○	·	·	·	·	·	○
	Gemini [55]	18,269	8,128	○	·	○	·	○	·	·	·	○	○	○	○	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	·	○	·	○
	GitZ [182]	44	·	·	○	·	○	·	·	·	·	○	○	○	○	○	·	3	·	·	·	·	2	1	·	·	·	2	8	·	·	·	·	·	·	·
	BinSim [183]	1,062	·	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	○
	BinSequence [154]	(1,718)	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	IMF-sim [151]	1,140	·	·	○	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	1	3	·	·	·	○	·	·	·
	CACompare [155]	72	·	○	·	○	·	○	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	·	·	·	○
	ASE17 [184]	55	·	○	○	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	·
2018	BinArm [60]	·	2,628	·	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	SANER18 [185]	7	·	○	○	·	·	·	·	·	·	·	·	·	·	·	·	1	1	1	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	BinGo-E [61]	(5,145)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	·	3	·	·	·	·	1	·	·	·	·	1	5	·	·	·	·	·	·	○
	WSB [186]	(173)	·	△	·	·	·	·	·	·	·	·	○	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	·
	BinMatch [187]	(82)	·	○	·	·	·	·	·	·	·	○	·	○	○	·	·	1	·	·	·	·	1	·	·	·	·	·	2	·	·	·	○	·	·	○
	MASES18 [145]	47	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○	·	·	·
	Zeek [188]	(20,680)	·	·	○	·	○	·	·	·	·	○	○	○	○	○	·	3	·	·	·	·	4	1	·	·	·	2	10	·	·	·	·	·	·	·
	FirmUp [57]	·	2,000	△	·	△	·	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	αDiff [62]	(69,989)	2	○	○	○	·	·	·	·	·	○	○	○	○	·	·	2	1	·	·	·	2	·	·	·	·	·	5	·	·	·	·	◑	◑	○
	VulSeeker [56]	(10,512)	4,643	○	○	○	○	○	○	·	·	○	○	○	○	·	·	1	1	·	·	·	·	·	·	·	·	·	2	·	·	·	·	○	·	○
2019	InnerEye [144]	(844)	·	·	○	○	·	·	·	·	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	·	1	·	·	1	·	·	·	·	◑	◑	·
	Asm2Vec [63]	68	·	·	○	·	·	·	·	·	·	○	○	○	○	·	·	1	1	·	·	·	2	·	·	·	·	2	6	·	·	·	○	○	·	○
	SAFE [64]	(5,001)	·	·	○	○	·	·	·	·	·	○	○	○	○	·	1	3	1	1	1	·	2	1	1	1	·	·	12	·	·	·	·	○	◑	○
	BAR19i [146]	(804)	·	·	○	○	·	·	·	·	·	·	○	○	○	·	·	·	·	·	·	·	·	·	·	·	1	·	·	1	·	·	·	·	○	·	·
	BAR19ii [149]	(11,244)	·	○	○	○	·	·	·	·	·	○	○	○	○	·	1	3	1	·	·	·	2	1	1	·	·	2	11	·	·	·	·	·	·	○
	FuncNet [189]	(180)	·	○	·	○	·	○	·	·	·	○	○	○	○	○	·	·	·	1	·	·	·	·	·	·	·	·	1	·	·	·	·	·	·	○
2020	DeepBinDiff [147]	(2,206)	·	·	△	·	·	·	·	·	·	○	○	○	○	·	·	1	·	·	·	·	·	·	·	·	·	·	1	·	·	·	·	○	○	○
	ImOpt [190]	18	·	·	○	·	·	·	·	·	·	○	·	○	○	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	○	·	·	·
	ACCESS20 [173]	12,000	·	△	△	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	Patchecko [148]	2,108	2	○	○	○	○	·	·	·	·	○	○	○	○	○	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	○
	BinKit ★	243,128	·	○	○	○	○	○	○	○	○	○	○	○	○	○	·	1	1	1	1	1	·	1	1	1	1	·	9	○	○	○	○	○	○	○

* We only mark items that are stated explicitly in the paper. Due to the lack of details about firmware images, we were not able to mark optimization options or compilers used to create them. For papers that do not explicitly state the number of binaries in their dataset, we estimated the number and marked it with parentheses.

† This table focuses on two major compilers: GCC and Clang, as other compilers only support a limited number of architectures.

△ We infer the target architectures of the dataset as they are not stated explicitly in the paper.

◑ This indicates that only a portion of the code and dataset is available. For example, discovRE [54] makes available only their firmware images, and αDiff [62] opens transformed function images but not the actual dataset.

5.2.2 Benchmarks Used in Prior Works

It is imperative to use the right benchmark to evaluate a BCSA technique. Therefore, we studied the benchmarks used in the past literature, as shown in Table 5.2. However, during the study, we found that it is

radically difficult to properly evaluate a new BCSA technique using the previous benchmarks.

First, we were not able to find a single pair of papers that use the same benchmark. Some of them share packages such as GNU `coreutils` [58, 150, 151], but the exact binaries, versions, and compiler options are not the same. Although there is no known standard for evaluating BCSA, it is surprising to observe that none of the papers uses the same dataset. We believe this is partly because of the difficulty in preparing the same benchmark. For example, even if we can download the same version of the source code used in a paper, it is extraordinarily difficult to cross-compile the program for various target architectures with varying compiler options; it requires significant effort to set up the environment. Note, however, *only two out of 43 papers we studied fully open their dataset*. Even in that case, it is hard to rebuild or extend the benchmark because of the absence of a public compilation script for the benchmark.

Second, the number of binaries used in each paper is limited and may not be enough for analytics. The *#Binaries* column of Table 5.2 summarizes the number of program binaries obtained from two different sources: application packages and firmware images. Since a single package can contain multiple binaries, we manually extracted the packages used in each paper and counted the number of binaries in each package. We counted only the binaries after a successful compilation, such that the object files that were generated during the compilation process were not counted. If a paper does not explicitly mention package versions, we used the most recent package versions at the time of writing and marked them with parentheses. Note that only 6 out of 43 papers have more than 10,000 binaries, and none reaches 100,000 binaries. Firmware may include numerous binaries, but it cannot be directly used for BCSA because one cannot generate the ground truth without having the source code.

Finally, previous benchmarks only cover a few compilers, compiler options, and target architectures. Some papers do not even describe their tested compiler options or package versions. The *Compiler* column of the table presents the number of minor versions used for each major version of the compilers. Notably, all the benchmarks except one consider less than five different major compiler versions. The *Extra* column of the table shows the use of extra compiler options for each benchmark. Here, only a few of them consider function inlining and Link-Time Optimization (LTO). None of them deals with the Position Independent Executable (PIE) option, although, currently, it is widely used [217].

All these observations lead us to the research questions presented in the next subsection (§5.2.3) and eventually motivate us to create our own benchmark that we call BINKIT, which is shown in the last row of Table 5.2.

5.2.3 Research Problems and Questions

We now summarize several key problems observed from the previous literature and introduce research questions derived from these problems. First, none of the papers uses the same benchmark for their evaluation, and the way they evaluate their techniques significantly differs. Second, only a few of the studies release their source code and data, which makes it radically difficult to reproduce or improve upon existing works. Furthermore, most papers use manually chosen ground truth data for their evaluation, which are easily error-prone. Finally, current state-of-the-art approaches in BCSA focus on extracting semantic features with complex analysis techniques (from §5.2.1 and §5.2.1). These observations naturally lead us to the below research questions. Note that some of the questions are indeed open-ended, and we only address them in part.

RQ1. How should we establish a large-scale benchmark and ground truth data?

One may build benchmarks by manually compiling application source code. However, there are so many different compiler versions, optimization levels, and options to consider when building binaries. Therefore, it is desirable to automate this process to build a large-scale benchmark for BCSA. It should be noted that many of the existing studies have also attempted to build ground truth from source code. However, the number of binaries

and compiler options used in those studies is limited and is not enough for data-driven research. Furthermore, those studies release neither their source code nor dataset (§5.2.2). On the contrary, we present a script that can automatically build large-scale ground truth data from a given set of source packages with clear descriptions (§5.3).

RQ2. Is the effectiveness of presemantic features limited to the target architectures and compiler options used?

We note that most previous studies assume that presemantic features are significantly less effective than semantic features as they can largely vary depending on the underlying architectures and compiler optimizations used. For example, compilers may perform target-specific optimization techniques for a specific architecture. Indeed, 36 out of the 43 papers ($\approx 84\%$) we studied focus on new semantic features in their analysis, as shown in Table 5.1. To determine whether this assumption is valid, we investigate it through a series of rigorous experimental studies. Although byte-level information significantly varies depending on the target and the optimization techniques, we found that some presemantic features, such as structural information obtained from CFGs, are broadly similar across different binaries of the same program. Additionally, we demonstrated that utilizing such presemantic features without a complex semantic analysis can achieve an accuracy that is comparable to that of a recent deep learning-based approach with a semantic analysis (§5.5).

RQ3. Can debugging information help BCSA achieve a high accuracy rate?

We are not aware of any quantitative study on how much debugging information affects the accuracy of BCSA. Most prior works simply assume that debugging information is not available, but how much does it help? How would decompilation techniques affect the accuracy of BCSA? To answer this question, we extracted a list of function types from our benchmark and used them to perform BCSA on our dataset. Surprisingly, we were able to achieve a higher accuracy rate than any other existing works on BCSA without using any sophisticated method (§5.6).

RQ4. Can we benefit from analyzing failure cases of BCSA?

Most existing works do not analyze their failure cases as they rely on uninterpretable machine learning techniques. However, our goal is to use a simple and interpretable model to learn from failure and gain insights for future research. Therefore, we manually examined failure cases using our interpretable method and observed three common causes for failure, which have been mostly overlooked by the previous literature. First, COTS binary analysis tools indeed return false results. Second, different compiler back-ends for the same architecture can be substantially different from each other. Third, there are architecture-specific code snippets for the same function. We believe that all these observations help in setting directions for future studies (§5.7).

Analysis Scope. In this chapter, we focus on function-level similarity analyses because functions are a fundamental unit of binary analysis, and function-level BCSA is widely used in previous literature [150, 54, 151, 62, 59, 152, 144, 64]. We believe one can easily extend our work to support whole-binary-level similarity analyses as in the previous papers [67, 68].

5.3 Establishing Large-Scale Benchmark and Ground Truth (RQ1)

Building a large-scale benchmark for BCSA and establishing its ground truth is challenging. One potential approach for generating the ground truth data is to manually identify similar functions from existing binaries or firmware images [53, 177, 179]. However, this requires domain expertise and is often error-prone and time-consuming.

Table 5.3: Summary of BINKIT.

Dataset Name	# of Packages	# of Architectures	# of Opti-Levels	# of Compilers	# of Binaries	# of Orig. Functions	# of Final Functions*
NORMAL	51	8	4	9	67,680	34,355,824	8,708,459
SIZEOPT	51	8	1†	9	16,920	8,350,442	2,060,625
PIE	46†	8	4	9	36,000	23,090,676	7,766,235
NOINLINE	51	8	4	9	67,680	38,617,186	10,291,001
LTO	29†	8	4	9	24,768	12,279,982	3,375,308
OBFUSCATION	51	8	4	4‡	30,080	15,809,489	4,054,694
Total	51		1,352 Options		243,128	132,503,599	36,256,322

* The target functions are selected in the manner described in §5.3.2.

† The number of packages and compiler options varies because some packages can be compiled only with a specific set of compile options.

‡ We count each of the four obfuscation options as a distinct compiler (§5.3.1).

Another approach for obtaining the ground truth is to compile binaries from existing source code with varying compiler options and target architectures [71, 58, 56, 59]. If we compile multiple binaries (with different compiler options) from the same source code, one can determine which function corresponds to which source lines. Unfortunately, most existing approaches do not open their benchmarks nor compilation scripts used to produce them (Table 5.2).

Therefore, we present BINKIT, which is a comprehensive benchmark for BCSA, along with automated compilation scripts that help reproduce and extend it for various research purposes. The rest of this section details BINKIT and discusses how we establish the ground truth (RQ1).

5.3.1 BINKIT: Large-Scale BCSA Benchmark

BINKIT is a comprehensive BCSA benchmark that comprises 243,128 binaries compiled from 51 package source code with 1,352 distinct combinations of compilers, compilation options, and target architectures. Therefore, BINKIT covers most of the benchmarks used in existing approaches as shown in Table 5.2. BINKIT includes binaries compiled for 8 different architectures. For example, we use both little- and big-endian binaries for MIPS to investigate the effect of endianness. It uses 9 different versions of compilers: GCC v{4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0} and Clang v{4.0, 5.0, 6.0, 7.0}. We also consider 5 optimization levels from `O0` to `O3` as well as `Os`, which is the code size optimization. Finally, we take PIE, LTO, and obfuscation options into account, which are less explored in BCSA.

We select GNU software packages [218] as our compilation target because of their popularity and accessibility: they are real applications that are used widely in Linux systems, and their source code is publicly available. We successfully compiled 51 GNU packages for all our target architectures and compiler options.

To better support targeted comparisons, we divide BINKIT into six datasets: NORMAL, SIZEOPT, NOINLINE, PIE, LTO, and OBFUSCATION. The summary of each dataset is shown in Table 5.3. Each dataset contains binaries obtained by compiling the GNU packages with different combinations of compiler options and targets. There is *no* intersection among the datasets. NORMAL includes binaries compiled for 8 different architectures with different compilers and optimization levels. We did not use other extra options such as PIE, LTO, and no-inline for this dataset. SIZEOPT is the same as NORMAL except that it uses only the `Os` optimization option instead of `O0`–`O3`. Similarly, PIE, NOINLINE, LTO, OBFUSCATION are no different from NORMAL except that they are generated by using an additional flag to enable PIE, to disable inline optimization, to enable LTO, and to enable compile-time obfuscation, respectively.

PIE makes memory references in binary relative to support ASLR. On some architectures, e.g., x86, compilers

inject additional code snippets to achieve relative addressing. As a result, the compiled output can differ severely. Although PIE became the default on most Linux systems [217], it has not been well studied for BCSA. Note we were not able to compile all the 51 packages with the PIE option enabled. Therefore, we have fewer binaries in PIE than NORMAL.

Function inlining embeds callee functions into the body of the caller. This can make presemantic features largely vary. Therefore, we investigate the effect of function inlining on BCSA by explicitly turning off the inline optimization with the `fno-inline` option.

LTO is an optimization technique that operates at link time. It removes unnecessary code blocks, thereby reducing the number of presemantic features. However, it also has been less studied in BCSA. We were only able to successfully compile 29 packages when the LTO option is enabled.

Finally, the OBFUSCATION dataset uses Obfuscator-LLVM [219] to obfuscate the target binaries. We chose Obfuscator-LLVM among various other tools previously used [219, 220, 221, 222, 223, 224] because it is the most commonly used [151, 187, 181, 63, 190], and we can directly compare the effect of obfuscation using the vanilla LLVM compiler. We use Obfuscator-LLVM's latest version with four obfuscation options: instruction substitution (SUB), bogus control flow (BCF), control flow flattening (FLA), and a combination of all the options. We regard each option as a distinct compiler, as shown in the *Comp* column of Table 5.3. One can obfuscate a single binary multiple times. However, we only applied it once. This is because obfuscating a binary multiple times could emit a significantly large binary, which becomes time-consuming for IDA Pro to preprocess. For example, when we obfuscate `a2ps` twice with all three options, the compiled binary reaches over 30 MB, which is 30 times larger than the normal one.

The number of packages and that of compiler options used in compiling each dataset differ because some packages can be compiled only with a specific set of compile options and targets. Some packages fail to compile because they have architecture-specific code, such as inline assemblies, or because they use compiler-specific grammars. For example, Clang does not support both the LTO option and the `Os` option to be turned on. There are also cases where packages have conflicting dependencies. We also excluded the ones that did not compile within 30 min because some packages require a considerable amount of time to compile. For instance, `smalltalk` took more than 10 h to compile with the obfuscation option enabled.

To summarize, BINKIT contains 243,128 binaries and 36,256,322 functions in total, which is indeed many orders of magnitude larger than the other benchmarks that appear in the previous literature. The *Source* column of Table 5.2 shows the difference clearly. BINKIT does not include firmware images because our goal is to automatically build a benchmark with the clear ground truth. One may extend our benchmark with firmware images. However, it would take significant manual effort to identify their ground truth. For additional details regarding each package, please refer to Table 5.11.

Our benchmark and compilation scripts are available on GitHub. Our compilation environment is based on Crosstool-NG [225], GNU Autoconf [226], and Linux Parallels [227]. Through this environment, we compiled the entire datasets of BINKIT in approximately 30 h on our server machine with 144 Intel Xeon E7-8867v4 cores.

5.3.2 Building Ground Truth

Next, we establish the ground truth for our dataset. We first define the criteria for determining the equivalence of two functions. In particular, we check whether two functions with the same name originated from the same source files and have the same line numbers. Additionally, we verify that both functions come from the same package and have the same name in their binaries to ensure their equivalence.

Based on these criteria, we constructed the ground truth by performing the following steps. First, we compiled

all the binaries with debugging information using the `-g` option. We then leveraged IDA Pro [101] to identify functions in the compiled binaries. Next, we labeled each identified function with its name, package name, binary name, as well as the name of the corresponding source file and line numbers. To achieve this, we wrote a script that parses the debugging information from each binary.

Using this information, we then sanitize our dataset to avoid having incorrect or biased results. Among the identified functions, we selected only the ones in the `code` (`.text`) segments as functions in other segments may not include valid binary code. For example, we disregarded functions in the Procedure Linkage Table (`.plt`) sections because these functions are wrappers to call external functions and do not include actual function bodies. In our dataset, we filtered out 40% of the identified functions in this step.

We also disregarded approximately 4% of the functions that are generated by the compiler, but not by the application developers. We can easily identify such compiler intrinsic functions by checking the corresponding source files and line numbers. For example, GCC utilizes intrinsic functions such as `__udivdi3` in `libgcc2.c` or `__aeabi_uldivmod` in `bpabi.S` to produce highly optimized code.

Additionally, we removed duplicate functions within the same project/package. Two different binaries often share the same source code especially when they are in the same project/package. For example, the GNU `coreutils` package contains 105 different executables that share 80% of the functions in common. We removed duplicate functions within each package by checking the source file names and their line numbers. Moreover, compilers can also generate multiple copies of the same function within a single binary due to optimization. These functions share the same source code but have a difference in their binary forms. For example, some parts of the binary code are removed or reordered for optimization purposes. As these functions share a large portion of code, considering all of them would produce a biased result. To avoid this, we selected only one copy for each of such functions in our experiments. This step filtered out approximately 54% of the remaining functions. The last column of Table 5.3 reports the final counting results, which is the number of unique functions.

By performing all the above steps, we can automatically build large-scale ground truth data. The total time spent building the ground truth of all our datasets was 13,300 seconds. By leveraging this ground truth data, we further investigate the remaining research questions (*i.e.*, RQ2–RQ4) in the following sections. To encourage further research, we have released all our datasets and source code.

5.4 Building an Interpretable Model

Previous BCSA techniques focused on achieving a higher accuracy by leveraging recent advances in deep learning techniques [62, 145, 55, 56]. This often requires building a complicated model, which is not straightforward to understand and hinders researchers from reasoning about the BCSA results and further answering the fundamental questions regarding BCSA. Therefore, we design an *interpretable* model for BCSA to answer the research questions and implement TikNib, which is a BCSA tool that employs the model. This section illustrates how we obtain such a model and how we set up our experimental environment.

5.4.1 TikNib Overview

At a high level, TikNib leverages a set of presemantic features widely used in the previous literature to reassess the effectiveness of presemantic features (RQ2). It evaluates each feature in two input functions, based on our similarity scoring metric (§5.4.3), which directly measures the difference between each feature value. In other words, it captures how much each feature differs across different compile options.

Note TikNib is intentionally designed to be simple so that we can answer the research questions presented

Table 5.4: Summary of numeric presemantic features used in TIKNIB.

Category	Features	Count
CFG	# of basic blocks, edges, loops, SCCs, and back edges # of all, arith, data transfer, cmp, and logic instrs. # of shift, bit-manipulating, float, misc instrs. # of arith + shift, and data transfer + misc instrs. # of all/unconditional/conditional control transfer instrs. Avg. # of edges per a basic block Avg./Sum of basic block, loop, and SCC sizes Avg. # of all, arith, data transfer, cmp, and logic instrs. Avg. # of shift, bit-manipulating, float, misc instrs. Avg. # of arith + shift, and data transfer + misc instrs. Avg. # of all/unconditional/conditional control transfer instrs.	41
CG	# of callers, callees, imported callees # of incoming/outgoing/imported calls	6
	Total	47

in §5.2.3. Despite the simplicity of our approach, TIKNIB still produces a high accuracy rate that is comparable to state-of-the-art tools (§5.5.2). We are *not* arguing here that TIKNIB is the best BCSA algorithm.

5.4.2 Features Used in TIKNIB

Recall from RQ2, one of our goals is to reconsider the capability of presemantic features. Therefore, we focus on choosing various presemantic features used in the previous BCSA literature instead of inventing novel ones.

However, creating a comprehensive feature set is not straightforward because of the following two reasons. First, there are numerous existing features, which are similar to one another, as discussed in Chapter 2. Second, some features require domain-specific knowledge, which is *not* publicly available. For example, several existing papers [54, 71, 55, 56, 61, 181, 60, 185] categorize instructions into semantic groups. However, grouping instructions is largely a subjective task, and there is no known standard for it. Furthermore, most existing works do not make their grouping algorithms public.

We address these challenges by (1) manually extracting representative presemantic features and (2) open-sourcing our feature extraction implementation. Specifically, we focus on numeric presemantic features. Because these features are represented as a number, the relationship among their values across different compile options can be easily observed.

Table 5.4 summarizes the selected features. Our feature set consists of CFG- and CG-level numeric features as they can effectively reveal structural changes in the target code. In particular, we utilize features related to basic blocks, CFG edges, natural loops, and strongly connected components (SCCs) from CFGs, by leveraging NetworkX [228]. We also categorize instructions into several semantic groups based on our careful judgment by referring to the reference manuals [229, 230, 231] and leveraging Capstone [232]'s internal grouping. Next, we count the number of instructions in each semantic group per function (i.e., CFG). Additionally, we take six features from CGs. The number of callers and callees represents a unique number of outgoing and incoming edges from CGs, respectively.

To extract these features, we conducted the following steps. First, we pre-processed the binaries in BINKIT with IDA Pro [101]. We then generated the ground truth of these binaries as we described in §5.3.2. For those functions of which we have the ground truth, we extracted the aforementioned features. Table 5.5 shows the time spent for each of these steps. The IDA pre-processing took most of the time as IDA performs various internal analyses. Meanwhile, the feature extraction took much less time as it merely operates on the precomputed results from the pre-processing step.

Table 5.5: Breakdown of the time for extracting features from BINKIT.

Dataset Name	IDA Pre-processing (s)	Ground Truth Building (s)	Feature Extraction (s)	Avg. Feature† Extraction (ms)
NORMAL	14,968.42	3,380.01	661.81	0.08
SIZEOPT	2,171.70	353.13	649.57	0.32
PIE	13,893.92	2,601.74	133.60	0.02
NOINLINE	14,780.06	3,883.88	579.82	0.06
LTO	5,263.97	1,314.94	392.48	0.12
OBFUSCATION	97,723.47	1,766.60	4,189.44	1.03

† The average time spent for extracting features from a function, which is computed by dividing the total time (the fourth column of this table) by the number of functions (the last column of Table 5.3).

5.4.3 Scoring Metric

Our scoring metric is based on the computation of the relative difference [233] between feature values. Given two functions A and B, let us denote a value of feature f for each function as A_f and B_f, respectively. Recall that any feature in TIKNIB can be represented as a number. We can compute the relative difference δ of the two feature values as follows:

$$\delta(A_f, B_f) = \frac{|A_f - B_f|}{|\max(A_f, B_f)|} \tag{5.1}$$

Let us suppose we have N distinct features $(f_1, f_2, \cdots, f_N)$ in our feature set. We can then define our similarity score s between two functions A and B by taking the average of relative differences for all the features as follows:

$$s(A,B) = 1 - \frac{\left(\delta(A_{f_1}, B_{f_1}) + \cdots + \delta(A_{f_N}, B_{f_N})\right)}{N} \tag{5.2}$$

Although each numeric feature can have a different range of values, TIKNIB can effectively handle them using relative differences by representing the difference of each feature with a value between 0 and 1. Therefore, the score s is always within the range of 0 to 1.

Furthermore, we can intuitively understand and interpret the BCSA results using our scoring metric. For example, suppose there are two functions A and B derived from the same source code with and without compiler option X, respectively. If the relative difference of the feature value f between the two functions is small, it implies that f is a robust feature against compiler option X.

In this thesis, we focus only on simple relative differences, rather than exploring complex relationships among the features for interpretability. However, we believe that our approach could be a stepping-stone toward fabricating more improved interpretable models to understand such complex relationships.

5.4.4 Feature Selection

Based on our scoring metric, we perform lightweight preprocessing to select useful features for BCSA as some features may not help in making a distinction between functions. To measure the quality of a given feature set, we compute the area under the receiver operating characteristic (ROC) curve (i.e., the ROC AUC) of generated models.

Suppose we are given a dataset in BINKIT, which is generated from source code containing N unique functions. In total, we have maximum $N \cdot M$ functions in our dataset, where M is the number of combinations of compiler options used to generate the dataset. The actual number of functions can be less than $N \cdot M$ due to function inlining. For each unique function λ, we randomly select two other functions with the following conditions. (1) A true positive (TP) function, λ^{TP}, is generated from the same source code as in λ, with different compiler options, and (2) a true negative (TN) function, λ^{TN}, is generated from source code that is different from the one used to generate

λ, with the same compiler options as for λ^{TP}. We generate such pairs for each unique function, thereby acquiring around $2 \cdot N$ function pairs. We then compute the similarity scores for the functions in each pair and their AUC.

We note that the same methodology has been used in prior works [55, 56]. We chose the method as it can efficiently analyze the tendency over a large-scale dataset. One may also consider top-k [55, 57, 63, 56] or precision@k [55, 63] as an evaluation metric, but this approach takes too much computational overhead: $O((N \cdot M)^2)$ operations.

Unfortunately, there is no efficient algorithm for selecting an optimal feature subset to use; it is indeed a well-known NP-hard problem [234]. Therefore, we leverage a greedy feature selection algorithm [235]. Starting from an empty set $\mathbb{F}$, we determine whether we can add a feature to $\mathbb{F}$ to increase its AUC. For every possible feature, we make a union with $\mathbb{F}$ and compute the corresponding AUC. We then select one that maximizes the AUC and update $\mathbb{F}$ to include the selected feature. We repeat this process until the AUC does not increase further by adding a new feature. Although our approach does not guarantee finding an optimal solution, it still provides empirically meaningful results, as we describe in the following sections.

5.4.5 Experimental Setup

For all experiments in this study, we perform 10-fold cross-validation on each test. When we split a test dataset, we ensure functions that share the same source code (*i.e.*, source file name and line number) are either in a training or testing set, but not in both. For each fold, during the learning phase, *i.e.*, the feature selection phase, we select up to 200K functions from a training set and conduct feature selection, as training millions of functions would take a significant amount of time. Limiting the number of functions for training may degrade the final results; however, when we tested the number of functions from 100K to 1000K, the result remained almost consistent. In the validation phase, we test all the functions in the testing set without any sampling. Thus, after 10-fold validation, all the functions in the target dataset are tested at least once.

We ran all our experiments on a server equipped with four Intel Xeon E7-8867v4 2.40 GHz CPUs (total 144 cores), 896 GB DDR4 RAM, and 8 TB SSD. We set up Ubuntu 18.04.5 LTS with IDA Pro v6.95 [101] on the server. For feature selection and similarity comparison, we utilized Python scikit-learn [236], SciPy [237], and NumPy [238].

5.5 Presemantic Feature Analysis (RQ2)

We now present our experimental results using TikNib on the presemantic features (§5.4.2) to answer RQ2 (§5.2.3). With our comprehensive analysis of these features, we obtained several useful insights for future research. In this section, we discuss our findings and lessons learned.

5.5.1 Analysis Result

To analyze the impact of various compiler options and target architectures on BCSA, we conducted a total of 72 tests using TikNib. We conducted the tests on our benchmark, BinKit, with the ground truth that we built in §5.3. Table 5.6 describes the experimental results where each column corresponds to a test we performed. Note that we present only 26 out of 72 tests because of the space limit. Unless otherwise specified, all the tests were performed on the Normal dataset. As described in §5.4.4, we prepared 10-fold sets for each test. We divided the tests into seven groups according to their purposes as shown in the top row of the table. For example, the *Arch* group contains a set of tests to evaluate each feature against varying target architecture.

Table 5.6: In-depth analysis results of presemantic features obtained by running TIKNIB on BINKIT.

Index	Description	Rand.	Opt Level: O0 vs. O3	Opt Level: O2 vs. O3	Rand.	Compiler: GCC v4 vs. GCC v8	Compiler: Clang v4 vs. Clang v7	Compiler: GCC vs. Clang	Rand.	Arch: x86 vs. ARM	Arch: x86 vs. MIPS	Arch: ARM vs. MIPS	Arch: 32 vs. 64	Arch: LE vs. BE	Rand.	vs. SizeOpt†: O0 vs. Os	vs. SizeOpt†: O1 vs. Os	vs. SizeOpt†: O3 vs. Os	vs. Extra†: PIE	vs. Extra†: NoInline	vs. Extra†: LTO	vs. Obfus.†: BCF	vs. Obfus.†: FLA	vs. Obfus.†: SUB	vs. Obfus.†: All	Bad‡: Norm.	Bad‡: Norm. vs. Obfus.†
①	# of Train Pairs (10^6)	0.40	0.13	0.19	0.36	0.19	0.19	0.19	0.40	0.17	0.16	0.17	0.18	0.20	0.39	0.14	0.17	0.18	6.04↑	0.17	0.19	0.19	0.19	0.20	0.18	3.63↑	4.56↑
	# of Test Pairs (10^6)	1.58	0.26	0.33	1.43	0.16	0.17	0.75	1.57	0.17	0.16	0.17	0.71	0.40	1.55	0.29	0.34	0.32	0.24	1.09	1.37	0.17	0.17	0.18	0.17	0.40↑	0.51↑
②	Train Time (sec)*	60.0	24.6	25.3	77.3	28.9	24.6	35.7	76.9	29.0	28.8	28.0	19.0	22.4	48.2	20.8	24.4	15.6	10.5	12.6	42.0	25.0	28.8	44.1	17.6	5.9	5.4
	Test Time (sec)*	59.1	23.3	23.8	49.7	12.0	11.8	40.9	54.4	12.3	12.5	12.0	39.9	22.8	52.2	24.8	23.2	22.6	32.0	62.5	60.7	11.9	11.3	10.6	11.2	1.6	1.6
③	CFG # of edges per BB	0.42	0.37	0.45	0.42	0.45	0.45	0.41	0.44	0.43	0.41	0.41	0.44	0.46	0.43	0.40	0.43	0.44	0.38	0.48	0.47	0.31	0.37	0.46	0.29	0.38	0.22
	CFG # of edges	0.57	0.50	0.68	0.58	0.68	0.69	0.57	0.64	0.67	0.62	0.63	0.67	0.72	0.64	0.54	0.63	0.60	0.63	0.66	0.72	0.40	0.37	0.72	0.31	0.52	0.25
	CFG # of loops	0.45	0.46	0.49	0.45	0.50	0.47	0.46	0.48	0.50	0.49	0.50	0.50	0.51	0.49	0.48	0.49	0.45	0.51	0.45	0.51	0.46	0.36	0.50	0.29	0.44	0.22
	CFG # of inter loops	0.46	0.47	0.49	0.46	0.50	0.48	0.47	0.48	0.50	0.49	0.50	0.50	0.50	0.49	0.48	0.49	0.45	0.51	0.45	0.50	0.46	0.38	0.50	0.32	0.45	0.27
	CG # of callees	0.57	0.52	0.63	0.58	0.64	0.63	0.57	0.61	0.64	0.57	0.58	0.60	0.64	0.59	0.54	0.62	0.60	0.56	0.61	0.62	0.61	0.61	0.64	0.60	0.52	0.56
	CG # of callers	0.55	0.55	0.59	0.56	0.60	0.58	0.54	0.58	0.58	0.51	0.53	0.56	0.60	0.57	0.58	0.60	0.58	0.54	0.58	0.57	0.56	0.56	0.58	0.55	0.54	0.54
	CG # of imported callees	0.55	0.56	0.63	0.58	0.62	0.59	0.55	0.59	0.63	0.49	0.50	0.59	0.57	0.57	0.58	0.61	0.58	0.58	0.59	0.59	0.56	0.57	0.59	0.56	0.52	0.55
	CG # of imported calls	0.56	0.57	0.65	0.59	0.65	0.62	0.56	0.61	0.67	0.50	0.51	0.62	0.60	0.59	0.59	0.62	0.59	0.61	0.60	0.62	0.53	0.59	0.62	0.52	0.52	0.51
	CG # of incoming calls	0.56	0.55	0.61	0.58	0.63	0.60	0.55	0.59	0.61	0.52	0.54	0.59	0.63	0.59	0.60	0.62	0.59	0.57	0.61	0.60	0.54	0.58	0.61	0.53	0.54	0.50
	CG # of outgoing calls	0.59	0.53	0.67	0.60	0.68	0.67	0.58	0.64	0.69	0.60	0.60	0.63	0.69	0.63	0.56	0.65	0.62	0.60	0.64	0.66	0.57	0.64	0.68	0.55	0.54	0.53
	Inst Avg. # of arith+shift	0.35	0.35	0.55	0.43	0.52	0.52	0.36	0.47	0.39	0.25	0.26	0.40	0.53	0.36	0.36	0.49	0.49	0.43	0.55	0.53	0.35	0.35	0.50	0.30	0.27	0.24
	Inst Avg. # of compare	0.44	0.40	0.52	0.46	0.51	0.52	0.44	0.49	0.45	0.44	0.37	0.50	0.55	0.46	0.43	0.50	0.49	0.46	0.54	0.55	0.33	0.37	0.54	0.29	0.39	0.21
	Inst Avg. # of ctransfer	0.34	0.33	0.45	0.37	0.42	0.42	0.33	0.40	0.39	0.36	0.38	0.39	0.46	0.37	0.35	0.39	0.39	0.38	0.43	0.46	0.30	0.28	0.43	0.24	0.31	0.18
	Inst Avg. # of ctransfer+cond.	0.27	0.25	0.32	0.28	0.32	0.31	0.26	0.30	0.29	0.28	0.29	0.29	0.33	0.28	0.28	0.29	0.29	0.28	0.33	0.34	0.23	0.20	0.32	0.17	0.22	0.11
	Inst Avg. # of dtransfer	0.34	0.30	0.49	0.38	0.46	0.49	0.35	0.43	0.35	0.31	0.33	0.38	0.52	0.36	0.31	0.43	0.43	0.37	0.49	0.49	0.35	0.33	0.47	0.27	0.31	0.19
	Inst Avg. # of dtransfer+misc	0.32	0.28	0.48	0.36	0.44	0.47	0.34	0.41	0.35	0.28	0.30	0.37	0.49	0.34	0.29	0.42	0.42	0.35	0.48	0.48	0.34	0.32	0.45	0.26	0.29	0.18
	Inst Avg. # of float instrs.	0.25	0.30	0.34	0.28	0.25	0.34	0.28	0.26	0.28	0.31	0.29	0.31	0.40	0.26	0.28	0.31	0.31	0.29	0.28	0.33	0.25	0.31	0.35	0.25	0.44	0.20
	Inst Avg. # of total instrs.	0.30	0.27	0.42	0.34	0.40	0.42	0.32	0.38	0.33	0.28	0.28	0.34	0.45	0.32	0.28	0.38	0.38	0.33	0.42	0.43	0.31	0.29	0.40	0.24	0.28	0.17
	Inst # of arith	0.40	0.43	0.64	0.51	0.62	0.61	0.46	0.55	0.43	0.28	0.29	0.48	0.62	0.41	0.44	0.58	0.56	0.53	0.61	0.60	0.40	0.50	0.58	0.35	0.33	0.27
	Inst # of arith+shift	0.40	0.43	0.64	0.51	0.62	0.61	0.46	0.55	0.44	0.27	0.28	0.47	0.63	0.41	0.43	0.58	0.56	0.53	0.61	0.61	0.40	0.50	0.59	0.35	0.32	0.27
	Inst # of bit-manipulating	0.26	0.29	0.35	0.28	0.33	0.32	0.26	0.30	0.19	0.13	0.20	0.33	0.17	0.25	0.27	0.32	0.30	0.33	0.31	0.33	0.25	0.23	0.24	0.18	0.41	0.03
	Inst # of compare	0.56	0.53	0.67	0.61	0.69	0.69	0.60	0.65	0.50	0.60	0.44	0.65	0.72	0.60	0.59	0.64	0.60	0.64	0.65	0.71	0.39	0.40	0.70	0.30	0.50	0.22
	Inst # of ctransfer	0.50	0.45	0.61	0.52	0.60	0.62	0.49	0.56	0.60	0.54	0.57	0.58	0.65	0.56	0.45	0.55	0.54	0.57	0.58	0.63	0.39	0.38	0.64	0.33	0.41	0.26
	Inst # of cond. ctransfer	0.60	0.54	0.68	0.61	0.69	0.70	0.63	0.67	0.69	0.66	0.67	0.69	0.72	0.67	0.60	0.64	0.60	0.64	0.65	0.72	0.39	0.37	0.71	0.31	0.52	0.25
	Inst # of dtransfer	0.42	0.36	0.63	0.46	0.61	0.61	0.48	0.55	0.48	0.37	0.41	0.50	0.64	0.46	0.34	0.57	0.56	0.49	0.61	0.61	0.41	0.39	0.61	0.33	0.38	0.26
	Inst # of dtransfer+misc	0.41	0.35	0.63	0.45	0.60	0.61	0.48	0.55	0.50	0.35	0.37	0.50	0.64	0.45	0.33	0.56	0.55	0.48	0.61	0.62	0.40	0.39	0.61	0.33	0.39	0.25
	Inst # of float instrs.	0.25	0.30	0.34	0.28	0.25	0.35	0.28	0.27	0.28	0.31	0.29	0.31	0.40	0.26	0.28	0.31	0.32	0.29	0.28	0.33	0.30	0.33	0.35	0.26	0.44	0.20
	Inst # of misc	0.30	0.26	0.52	0.34	0.48	0.48	0.33	0.42	0.11	0.31	0.27	0.46	0.67	0.34	0.23	0.39	0.38	0.38	0.48	0.50	0.30	0.31	0.50	0.26	0.32	0.15
	Inst # of shift	0.36	0.38	0.45	0.38	0.45	0.40	0.36	0.41	0.30	0.46	0.47	0.42	0.55	0.38	0.34	0.39	0.38	0.42	0.39	0.47	0.41	0.40	0.44	0.38	0.48	0.49
	Inst # of total instrs.	0.43	0.38	0.62	0.47	0.60	0.61	0.50	0.56	0.54	0.37	0.38	0.52	0.63	0.47	0.35	0.56	0.54	0.50	0.59	0.61	0.39	0.39	0.59	0.32	0.42	0.25
④	Avg. TP-TN Gap	0.42	0.41	0.53	0.45	0.52	0.52	0.44	0.49	0.46	0.42	0.43	0.49	0.55	0.46	0.42	0.49	0.48	0.48	0.51	0.54	0.39	0.38	0.53	0.32	0.42	0.27
	Avg. TP-TN Gap of Grey	0.49	0.44	0.54	0.49	0.59	0.60	0.52	0.56	0.57	0.52	0.50	0.59	0.60	0.57	0.49	0.56	0.54	0.53	0.57	0.53	0.48	0.48	0.57	0.44	0.47	0.45
⑤	Avg. # of Selected Features	8.5	13.9	9.3	12.9	10.1	8.7	11.7	11.0	11.0	12.0	11.0	7.1	8.4	7.3	11.0	10.0	5.7	14.3	5.3	16.5	8.3	9.9	15.7	6.1	11.6	8.1
⑥	ROC AUC	0.95	0.93	0.99	0.96	1.00	1.00	0.97	0.98	1.00	0.98	0.98	0.99	1.00	0.98	0.96	0.99	0.97	0.97	0.98	1.00	0.98	0.98	1.00	0.95	0.93	0.91
	Std. of ROC AUC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
⑦	Average Precision (AP)	0.95	0.93	0.99	0.97	1.00	1.00	0.97	0.99	0.99	0.97	0.98	0.99	1.00	0.98	0.96	0.99	0.98	0.98	0.98	1.00	0.98	0.98	1.00	0.95	0.93	0.90
	Std. of AP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01

All values in the table are 10-fold cross validation averages. We color a cell gray if a feature was consistently selected (*i.e.*, 10 times) during the 10-fold validation. Due to the space constraints, we only display features that have been selected at least once during the 10-fold validation.

† We compare a function from the NORMAL to the corresponding function in each target dataset.

‡ We match functions whose compiler options are largely distant to test for bad cases. Please refer to §5.5.1 for additional information.

↑ These in the first rows (①) are divided by 10^4 instead of 10^6.

* The train and test times in the seconds rows (②) do not include the time for data loading.

For each test, we select function pairs for training and testing as described in §5.4.4. That is, for a function λ, we select its corresponding functions (*i.e.*, λ^{TP} and λ^{TN}). Therefore, N functions produce $2 \cdot N$ functions pairs. The first row (①) of Table 5.6 shows the number of function pairs for each test. When selecting these pairs, we deliberately choose the target options based on the goal of each test. For instance, we test the influence of varying the target architecture from x86 to ARM (*x86 vs. ARM* column of Table 5.6). For each function λ in the x86 binaries of our dataset, we select both λ^{TP} and λ^{TN} from the ARM binaries compiled with the same compiler option as in λ. In other words, we fix all the other options, except for the target architecture for choosing λ^{TP} and λ^{TN} to focus on our testing goal. The same rule applies to other columns. For the *Rand.* columns, we alter all the compiler options in the group randomly to generate function pairs.

The second row (②) of Table 5.6 presents the time spent for training and testing in each test, which excludes the time for loading the function data on the memory. The average time spent for a single function was less than 1 ms.

Each cell in the third row (③) of Table 5.6 represents the average of $\delta(\lambda_f, \lambda_f^{\text{TN}}) - \delta(\lambda_f, \lambda_f^{\text{TP}})$ for feature f, which we call the *TP-TN gap* of f. This TP-TN gap measures the similarity between λ^{TP} and λ, as well as the difference between λ^{TN} and λ, in terms of the target feature. Thus, when the gap of a feature is larger, its discriminative capability for BCSA is higher. As we conduct 10-fold validation for each test, we highlight the cells with gray when the corresponding feature is chosen in all ten trials. Such features show relatively higher TP-TN gaps than the others do in each test. We also present the average TP-TN gaps in the fourth row (④) of the table.

The average number of the selected features in each test is shown in the fifth row (⑤) of Table 5.6. A few presemantic features could achieve high AUCs and average precisions (APs), as shown in the sixth row (⑥) and seventh row (⑦) of the same table, respectively. We now summarize our observations as follows.

(1) Optimization is largely influential

Many researchers have focused on designing a model for *cross-architecture* BCSA [70, 58, 54, 153, 61]. However, our experimental results show that architecture may not be the most critical factor for BCSA. Instead, optimization level was the most influential factor in terms of the relative difference between presemantic features. In particular, we measured the average TP-TN gap of all the presemantic features for each test (*Avg. of TP-TN Gap* row of the table) and found that the average gap of the `O0` vs. `O3` test (0.41) is less than that of the x86 vs. ARM test (0.46) and the x86 vs. MIPS test (0.42). Furthermore, the optimization level random test (*Rand.* column of the *Opt Level* group) shows the lowest AUC (0.96) compared to that of the architecture and compiler group (0.98). These results confirm that compilers can produce largely distinct binaries depending on the optimization techniques used; hence, the variation among the binaries due to the optimization is considerably greater than that due to the target architecture on our dataset.

(2) Compiler version has a small impact

Approximately one-third of the previous benchmarks shown in Table 5.2 employ multiple versions of the same compiler. However, we found that even the major versions of the same compiler produce similar binaries. In other words, compiler versions do not heavily affect presemantic features. Although Table 5.6 does not include all the tests we performed because of the space constraints, it is apparent from the *Compiler* column that the two tests between two different versions of the same compiler, i.e., GCC v4 vs. GCC v8 and Clang v4 vs. Clang v7, have much higher TP-TN gaps (0.52) than other tests, and their AUCs are close to 1.0.

(3) GCC and Clang have diverse characteristics

Conversely, the GCC vs. Clang test resulted in the lowest TP-TN gap (0.44) and AUC (0.97) among the tests in the *Compiler* group. This can be because each compiler employs a different back-end, thereby producing different binaries. Another potential problem is that the techniques inside each optimization level can vary depending on the compiler. We detail this in §5.7.2.

(4) ARM binaries are closer to x86 binaries than MIPS

The tests in the *Arch* group measure the influence of target architectures with the NORMAL dataset. Overall, the target architecture did not have much of an effect on the accuracy rate. The AUCs were over 0.98 in all the cases. Surprisingly, the x86 vs. ARM test had the highest TP-TN gap (0.46) and AUC (1.0), indicating that the presemantic features of the x86 and ARM binaries are similar to each other, despite being distinct architectures. The ARM vs. MIPS test showed a lower TP-TN gap (0.43) and AUC (0.98) although both of them are RISC architectures. Additionally, the effect of the word size (*i.e.*, bits) and endianness was relatively small. Nevertheless, we cannot rule out the possibility that our feature extraction for MIPS binaries is erroneous. We further discuss this issue in §5.7.1.

(5) Closer optimization levels show similar results

We also measured the effect of size optimization (`Os`) by matching function λ in the NORMAL dataset with a function (λ^{TP} and λ^{TN}) in the SIZEOPT dataset. Subsequently, the binaries compiled with the `Os` option were similar to the ones compiled with the `O1` and `O2` options. This is not surprising because `Os` enables most of the `O2` techniques in both GCC and Clang [239, 240]. Furthermore, we observe that the `O1` and `O2` options produce similar binaries, although it is not shown in Table 5.6 due to the space limit.

(6) Extra options have less impact

To assess the influence of the PIE, no-inline, and LTO options, we compared functions in the NORMAL dataset with those in the PIE, NOINLINE, and LTO datasets, respectively. For the no-inline test, we limit the optimization level from `O1` to `O3` as function inlining is applied from `O1`. It was observed that the influence of such extra options is not significant. Binaries with and without the PIE option were similar to each other because it only changes the instructions to use relative addresses; hence, it does not affect our presemantic features. Function inlining also does not affect several features, such as the number of incoming calls, which results in a high AUC (0.97). LTO does not exhibit any notable effect either.

However, by analyzing each test case, we found that some options affect the AUC more than others. For example, in the no-inline test, the AUC largely decreases as the optimization level increases: `O1` (0.995), `O2` (0.981), and `O3` (0.967). This is because as more optimization techniques are applied, more functions are inlined and transformed in the NORMAL, while their corresponding functions in the NOINLINE are not inlined. On the other hand, in the LTO test, the AUC increases as the version of Clang increases: v4 (0.956), v5 (0.968), v6 (0.986), and v7 (0.986). In contrast, GCC shows stable AUCs (0.987–0.988) across all versions, and all the AUCs are higher than those of Clang. This result indicates that varying multiple options would significantly affect the success rate, which we describe below.

Table 5.7: Summary of datasets for comparing TIKNIB to VulSeeker (*i.e.*, ASE datasets).

Name	Package	Architecture	Compiler (GCC)	# of Orig. Funcs	# of Final Funcs
ASE1	OpenSSL v1.0.1{f,u}	{x86,arm,mips}_32	v5.5.0	152K	126K
ASE2	OpenSSL v1.0.1{f,u} BusyBox v1.21 Coreutils v6.{5,7}	”	”	704K	183K
ASE3	”	{x86,arm,mips}_32, {x86,arm,mips}_64	v4.9.4, v5.5.0	2,777K	735K
ASE4	”	Same as NORMAL options		16,799K	4,467K

As the index of the dataset grows, the number of packages, architectures, and compiler options increases ASE1 and ASE3 are the datasets used in VulSeeker [56]. For all datasets, the optimization levels are `O0`–`O3`.

(7) Obfuscator-LLVM does not affect CG features

Many previous studies [151, 187, 181, 63, 190] chose Obfuscator-LLVM [219] for their obfuscation tests as it significantly varies the binary code [63]. However, applying all of its three obfuscation options shows an AUC of 0.95 on our dataset, which is relatively higher than that of the optimization level tests. Obfuscation severely decreases the average TP-TN gaps except for CG features. This is because Obfuscator-LLVM applies intra-procedural obfuscation. The `SUB` obfuscation substitutes arithmetic instructions while preserving the semantics; the `BCF` obfuscation notably affects CFG features by adding bogus control flows; the `FLA` obfuscation changes the predicates of control structures [241]. However, none of them conducts inter-procedural obfuscation, which modifies the function call relationship. Thus, we encourage future studies to use other obfuscators, such as Themida [242] or VMProtect [221], for evaluating their techniques against inter-procedural obfuscation.

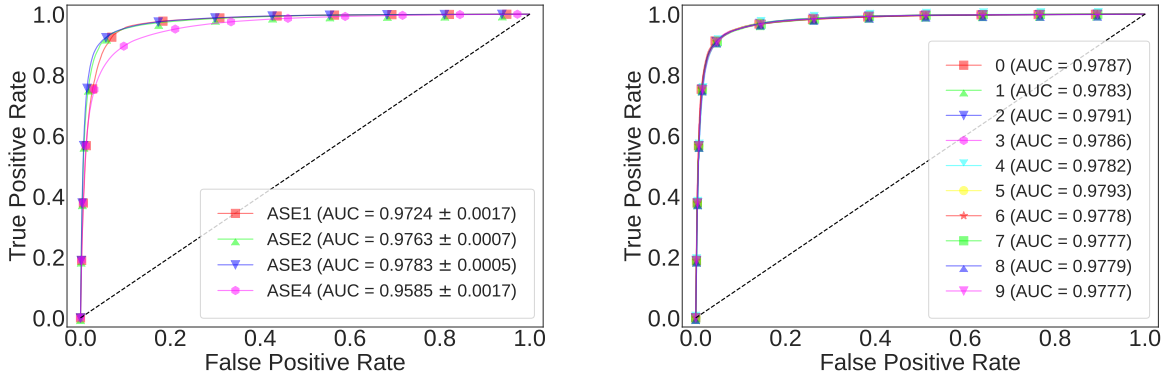
(8) Comparison target option does matter

Based on the experimental results thus far, we perform extra tests to understand the influence of comparing multiple compiler options by intentionally selecting λ^{TP} and λ^{TN} from binaries that could provide the lowest TP-TN gap. In this study, we present two of them because of the space limit. Specifically, for the first test, we selected functions from 32-bit ARM binaries compiled using GCC v4 with the `O0` option, and the corresponding λ^{TP} and λ^{TN} functions from 64-bit MIPS big-endian binaries compiled using Clang v7 with the `O3` option. For the second test, we changed the Clang compiler to the Obfuscator-LLVM with all three obfuscation options turned on. The *Bad* column of the table summarizes the results. The AUC of both cases was approximately 0.93 and 0.91, respectively. Their average TP-TN gaps were also significantly lower (0.42 and 0.27) than those in the other tests. This signifies the importance of choosing the comparison targets for evaluating BCSA techniques. Existing BCSA research compares functions for all possible targets in a dataset, as shown in the *Rand.* tests in this study. However, our results suggest that researchers should carefully choose evaluation targets to avoid overlooking the influence of bad cases.

5.5.2 Comparison Against State-of-the-Art Techniques

From our experiments in §5.5.1, we show that using only presemantic features with a simple linear model (i.e., TIKNIB) is enough to obtain high AUC values. Next, we compare TIKNIB with state-of-the-art techniques.

To accomplish this, we chose one of the latest approaches, VulSeeker [56], as our target because it utilizes both presemantic and semantic features in a numeric form by leveraging neural network-based post-processing. Thus, we can directly evaluate our simple model using numeric presemantic features. Note that *our goal is not*



(a) ROC AUCs for all ASE datasets.

(b) ROC AUC of each fold for ASE3.

Figure 5.2: Experimental results obtained by running TIKNIB on ASE datasets.

to claim that our approach is better, but to demonstrate that the proper engineering of presemantic features can achieve results that are comparable to those of state-of-the-art techniques.

For this experiment, we prepared the datasets of VulSeeker, along with the additional ones as listed in Table 5.7. We refer to these datasets as ASE1 through ASE4. ASE1 and ASE3 are the ones used in VulSeeker, and ASE2 and ASE4 are extra ones with more packages, target architectures, and compilers. Note that the number of packages, architectures, and compiler options increases as the index of the dataset increases. The optimization levels for all datasets are `O0`–`O3`. We intentionally omitted firmware images used in the original paper, as they do not provide solid ground truth. For each dataset, we established the ground truth in the same way described in §5.3.2. The time spent for IDA pre-processing, ground truth building, and feature extracting was 2197 s, 889 s, and 239 s, respectively. We then conducted experiments with the methodology explained in §5.4; note that the same methodology was used in the original paper.

Figure 5.2 depicts the results. Figure 5.2a shows that the AUCs of TIKNIB on ASE1 and ASE3 are 0.9724 and 0.9783, respectively. However, those of VulSeeker were 0.99 and 0.8849 as reported by the authors [56]. Figure 5.2b illustrates that the AUC of each fold in ASE3 ranged from 0.9777 to 0.9793, which is higher than that of VulSeeker (0.8849). Therefore, TIKNIB was more robust than VulSeeker in terms of the size and compile options in the dataset. TIKNIB also exhibits stable results even for ASE2 and ASE4.

From these results, we conclude that presemantic features with proper feature engineering can achieve results that are comparable to those of state-of-the-art BCSA techniques. Although our current focus is on comparing feature values, it is possible to extend our work to analyze the complex relationships among the features by utilizing advanced machine learning techniques [56, 71, 55, 144, 62, 145, 63, 64, 146, 147, 148, 149].

5.5.3 Analysis on Real-World Vulnerabilities

To further assess the effectiveness of presemantic features, we apply TIKNIB to vulnerability discovery, which is a common practical application of BCSA [53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]. We investigate whether TIKNIB can effectively identify a vulnerable function across various compiler options and architectures.

We chose the `tls1_process_heartbeat` function in the `OpenSSL` package as our target function because it contains the infamous Heartbleed vulnerability (*i.e.*, CVE-2014-0160), and has thus been used in many prior BCSA studies to evaluate their approaches [54, 55, 71, 63]. We utilized two versions of `OpenSSL` in the ASE4 dataset shown in Table 5.7: One (*i.e.*, v1.0.1f) contains the vulnerable function, while the other (*i.e.*, v1.0.1u) contains the patched version. As the dataset was compiled with 288 distinct combinations of compiler options and architectures,

each function has 576 samples: 288 (the number of possible combinations) × 2 (the number of available `OpenSSL` versions) ≈ 576.

Notably, testing all possible combinations of options entails a significant computational overhead; it requires 288 (the number of options for our target function) × 287 (the number of options for a function in `OpenSSL`) × 2 (the number of `OpenSSL` versions) × 5K (the number of functions in `OpenSSL`) ≈ 826M operations. Therefore, we focused on architectures and compiler options that are widely used in software packages. Specifically, we chose three 64-bit architectures (ARM, x86, and MIPS) and two levels of optimization (O2 and O3). This setup reflects real-world scenarios, as many software packages use `O2`–`O3` by default: `coreutils` uses `O2`, while `OpenSSL` uses `O3`. Previous studies [63, 179] also used the same setup (`O2`–`O3`) except the architecture; they only tested for x86. Additionally, we selected four compilers (Clang v4.0, Clang, v7.0, GCC v4.9.4, and GCC v8.2.0) to consider extreme cases. Consequently, there were 24 possible combinations of these architectures and compiler options.

We conducted a total of 552 tests on these 24 option combinations: 24 (the number of options for our target function) × 23 (the number of options for a function in `OpenSSL`). For each test, we simply computed the similarity scores for all function pairs using TikNib and checked the rank of the vulnerable function. To reflect real-world scenarios, we assumed in all tests that we were not aware of the precise optimization level, compiler type, or compiler version of the testing binary. On the other hand, we assumed that we could recognize the architecture of the testing binary as it is straightforward. Therefore, when we train TikNib, we chose a feature set that achieved the best performance across all possible combinations of optimization levels, compiler types, and compiler versions, while setting the source and target architectures fixed. For training, we used the Normal dataset (Table 5.3) as it does not include `OpenSSL`; thus, the training and testing datasets are completely distinct.

Table 5.8 summarizes the experimental results, with each column corresponding to the tests for the specified options. We organized the results by option group specified in each column after running all 522 tests. The first row of the table (*# of Option Pairs*) indicates the total number of options pairs, which is the same as that of true positive pairs. The remaining rows of the table show the averaged values obtained by the option pair tests. For example, the *All to All* column represents the averaged results of all possible combinations (24 × 23). The *ARM to MIPS* column, on the other hand, represents the averaged results of all combinations with the source and target architectures set to ARM and MIPS, respectively. That is, we queried the vulnerable functions compiled with ARM and searched for its true positives compiled with MIPS while modifying the other options.

In the majority of the tests, TikNib successfully identified the vulnerable function with a rank close to 1.0 and a precision@1 close to 1.0, demonstrating its effectiveness in vulnerability discovery. Meanwhile, it performed marginally worse in the tests for MIPS. This result corroborates our observation in §5.5.1 that feature extraction for MIPS binaries can be erroneous. We further discuss this issue in §5.7.1. Additionally, the last three rows of Table 5.8 display the ranks of additional functions worth noting. The `dtls` represents the DTLS implementation of our target function (*i.e.*, `dtls1_process_heartbeat`), which also contains the vulnerability. Due to its similarity to our target function, it was ranked highly in all tests. The last two rows of the table present the ranks of the patched versions of these two functions in `OpenSSL` v1.0.1u. Notably, the patch of the vulnerability affects the presemantic features of these functions, particularly the number of control transfer and arithmetic instructions. Consequently, the patched functions had a low rank.

5.6 Benefit of Type Information (RQ3)

To evaluate the implication of debugging information on BCSA, we select type information as a case study on the presumption that they do not vary unless the source code is changed. Specifically, we extract three types of features per function: the number of arguments, types of arguments, and return type of a function. Note that

Table 5.8: Top-k and precision@1 results of analyzing the Heartbleed vulnerability in `OpenSSL` (*i.e.*, CVE-2014-0160) using TIKNIB.

Source option to Target option	**All to All**	**ARM to ARM**	**ARM to MIPS**	**ARM to x86**	**MIPS to MIPS**	**MIPS to ARM**	**MIPS to x86**	**x86 to x86**	**x86 to ARM**	**x86 to MIPS**	**O2 to O3**	**O3 to O2**	**GCC to Clang**	**GCC v4 to GCC v8**	**GCC v8 to GCC v4**	**Clang v4 to Clang v7**	**Clang v7 to Clang v4**
# of Option Pairs	552	56	64	64	56	64	64	56	64	64	144	144	144	36	36	36	36
Rank (tls, vuln)*	1.19	1.14	1.66	1	1	1.62	1	1	1.25	1	1.18	1.19	1	1.44	1.06	1	1
Precision@1 (tls, vuln)*	0.89	0.86	0.66	1	1	0.75	1	1	0.75	1	0.9	0.89	1	0.78	0.94	1	1
Rank (dtls, vuln)†	4.54	9.82	11.81	3.06	2	4.72	2	2.07	1.75	3.62	4.5	4.38	2.72	3.11	5.06	3.61	3.33
Rank (tls, patched)‡	29.16	12.12	57.69	3.56	3.82	51.62	43.94	4.29	6.38	70.59	27.5	28.96	27.68	32.89	40.89	20.22	22.67
Rank (dtls, patched)‡	76.47	46.95	145.75	7.25	8.21	128	128.94	9.57	11.94	181.03	73.04	75.41	87.31	66.28	87.33	68.44	78

All rank and precision@1 values are averaged; because each test involves multiple combinations of options (the first row), their results are averaged.

* These are the results of the vulnerable `tls1_process_heartbeat` function in `OpenSSL` v1.0.1f.

† This is the result of the vulnerable `dtls1_process_heartbeat` function in `OpenSSL` v1.0.1f, which is similar to but distinct from the `tls1_process_heartbeat` function.

‡ These are the results of their patched versions in `OpenSSL` v1.0.1u.

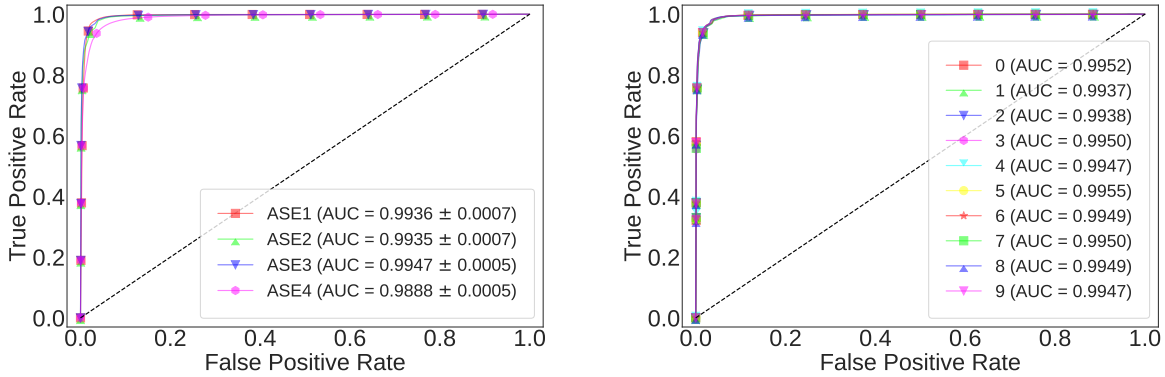
Table 5.9: In-depth analysis results of presemantic and type features obtained by running TIKNIB on BINKIT.

		Opt Level			**Compiler**				**Arch**						**vs. SizeOpt†**			**vs. Extra†**			**vs. Obfus.†**				**Bad‡**	
	Rand.	**O0 vs. O3**	**O2 vs. O3**	**Rand.**	**GCC 4 vs. GCC 8**	**Clang 4 vs. Clang 7**	**GCC vs. Clang**	**Rand.**	**x86 vs. ARM**	**x86 vs. MIPS**	**ARM vs. MIPS**	**32 vs. 64**	**LE vs. BE**	**Rand.**	**O0 vs. Os**	**O1 vs. Os**	**O3 vs. Os**	**PIE**	**NoInline**	**LTO**	**BCF**	**FLA**	**SUB**	**All**	**Norm.**	**Norm. vs. Obfus.** †
Avg. # of Selected Features	4.0	4.0	6.8	5.3	7.1	10.5	6.8	3.0	12.0	6.4	7.1	9.4	7.7	9.0	7.0	7.0	7.0	8.4	6.0	7.6	7.0	7.0	8.3	6.0	4.0	4.5
Avg. TP-TN Gap of Grey	0.53	0.52	0.56	0.53	0.58	0.59	0.53	0.56	0.54	0.54	0.54	0.56	0.56	0.55	0.54	0.57	0.57	0.56	0.56	0.55	0.50	0.52	0.58	0.50	0.55	0.55
ROC AUC	0.99	0.99	1.00	0.99	1.00	1.00	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.99
Average Precision (AP)	0.99	0.99	1.00	0.99	1.00	1.00	0.99	0.99	1.00	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.98

All values in the table are 10-fold cross validation averages.

† We compare a function from the NORMAL to the corresponding function in each target dataset.

‡ We match functions whose compiler options are largely distant to test for bad cases. Please refer to §5.5.1 for additional information.



(a) ROC AUCs for all ASE datasets.

(b) ROC AUC of each fold for ASE3.

Figure 5.3: Experimental results obtained by running TIKNIB on ASE datasets including type features.

inferring the correct type information is challenging and is actively researched [243, 244]. In this context, we only consider basic types: `char`, `short`, `int`, `float`, `enum`, `struct`, `void`, and `void *`. To extract type information, we create a type map to handle custom types defined in each package by recursively following definitions using Ctags [245]. We then assign a unique prime number as an identifier to each type. To represent the argument types as a single number, we multiply their type identifiers.

To investigate the benefit of these type features, we conducted the same experiments described in §5.5, and Table 5.9 presents the results. Here, we explain the results by comparing them with Table 5.6, which we obtained without using the type features. The first row of Table 5.9 shows that the average number of selected features including type features is smaller than that of selected features (⑤) in Table 5.6. Note that all three type features were always selected in all tests. The second row in Table 5.9 shows that utilizing the type features could achieve a large TP-TN gap on average (over 0.50); the corresponding values in ④ of Table 5.6 are much smaller. Consequently, the AUC and AP with type features reached over 0.99 in all tests, as shown in the last two rows of Table 5.9. Additionally, it shows a similar result (*i.e.*, an AUC close to 1.0) on the ASE datasets that we utilized for the state-of-the-art comparison (§5.5.2), and Figure 5.3 illustrates the result.

This result confirms that type information indeed benefits BCSA in terms of the success rate, although recovering such information is a difficult task. Therefore, we encourage further research on BCSA to take account of recovering debugging information, such as type recovery or inference, from binary code [246, 247, 243, 244, 248, 249].

5.7 Failure Case Inquiry (RQ4)

We carefully analyzed the failure cases in our experiments and found their causes. It was possible because our benchmark (*i.e.*, BINKIT) has the ground truth and our tool (*i.e.*, TIKNIB) uses an interpretable model. We first checked the TP-TN gap of each feature for failure cases and further analyzed them using IDA Pro. We found that optimization largely affects the BCSA performance as described in §5.5.1. In this section, we discuss other failure causes and summarize the lessons learned; however, many of these causes are closely related to optimization. We categorized the causes into three cases: (1) errors in binary analysis tools (§5.7.1), (2) differences in compiler back-ends (§5.7.2), and (3) architecture-specific code (§5.7.3).

5.7.1 Errors in Binary Analysis Tools

Most BCSA research heavily relies on COTS binary analysis tools such as IDA Pro [101]. However, we found that IDA Pro can yield false results. First, IDA Pro fails to analyze indirect branches, especially when handling MIPS binaries compiled with Clang using the position-independent code (PIC) option. The PIC option sets the compiler to generate machine code that can be placed in any address, and it is mainly used for compiling shared libraries or PIE binaries. Particularly, compilers use register-indirect branch instructions, such as `jalr`, to invoke functions in a position-independent manner. For example, when calling a function, GCC stores the base address of the Global Offset Table (GOT) in the `gp` register, and uses it to calculate the function addresses at runtime. In contrast, Clang uses the `s0` or `v0` register to store such base addresses. This subtle difference confuses IDA Pro and makes it fail to obtain the base address of the GOT; thus, it cannot compute the target addresses of indirect branches.

Moreover, IDA Pro sometimes generates incomplete CFGs. When there is a *switch* statement, compilers often make a table that stores a list of jump target addresses. However, IDA Pro often failed to correctly identify the number of elements in the table, especially on ARM architecture, where switch tables can be placed in a code segment. Sometimes, switch tables are located in between basic blocks, and it is more difficult to distinguish them.

The problem worsens when handling MIPS binaries compiled for Clang with PIC because switch tables are typically stored in a read-only data section, which can be referenced through a GOT. Therefore, if IDA Pro cannot fully analyze the base address of the GOT, it also fails to identify the jump targets of switch statements.

As we manually analyze the errors, we may have missed some. Systematically finding such errors is a difficult task because the internals of many disassembly tools are not fully disclosed, and they differ significantly. One may extend the previous study [48] to further analyze the errors of disassembly tools and extracted features, and we leave this for future studies.

During the analysis, we found that IDA Pro also failed to fetch some function names if they have a prefix pre-defined in IDA Pro, such as `off_` or `sub_`. For example, it failed to fetch the name of the `off_to_chars` function in the `tar` package. We used IDA Pro v6.95 in our experiments, but we found that its latest version (v7.5) does not have this issue.

5.7.2 Diversity of Compiler Back-ends

From §5.5.1, the characteristics of binaries largely vary depending on the underlying compiler back-end. Our study reveals that GCC and Clang emit significantly different binaries from the same source code.

First, the number of basic blocks for the two compilers significantly differs. To observe how the number changes depending on different compiler options and target architectures, we counted the number for the NORMAL dataset. Figure 5.4 illustrates the number of functions and basic blocks in the dataset for selected compiler options and architectures (see §5.8 for details). As shown in the figure, the number of basic blocks in binaries compiled with Clang is significantly larger than that in binaries compiled with GCC for `O0`. We figured out that Clang inserts dummy basic blocks for `O0` on ARM and MIPS; these dummy blocks have only one branch instruction to the next block. These dummy blocks are removed when the optimization level increases (`O1`) as optimization techniques in Clang merge such basic blocks into their predecessors.

In addition, the two compilers apply different internal techniques for the same optimization level, while they express the optimization level with the same terms (*i.e.*, `O0`–`O3` and `Os`). In particular, by analyzing the number of caller and callee functions, we discovered that GCC applies function inlining from `O1`, whereas Clang applies it from `O2`. Consequently, the number of functions for each compiler significantly differs (see the number of functions in `O1` for Clang and that for GCC in Figure 5.4).

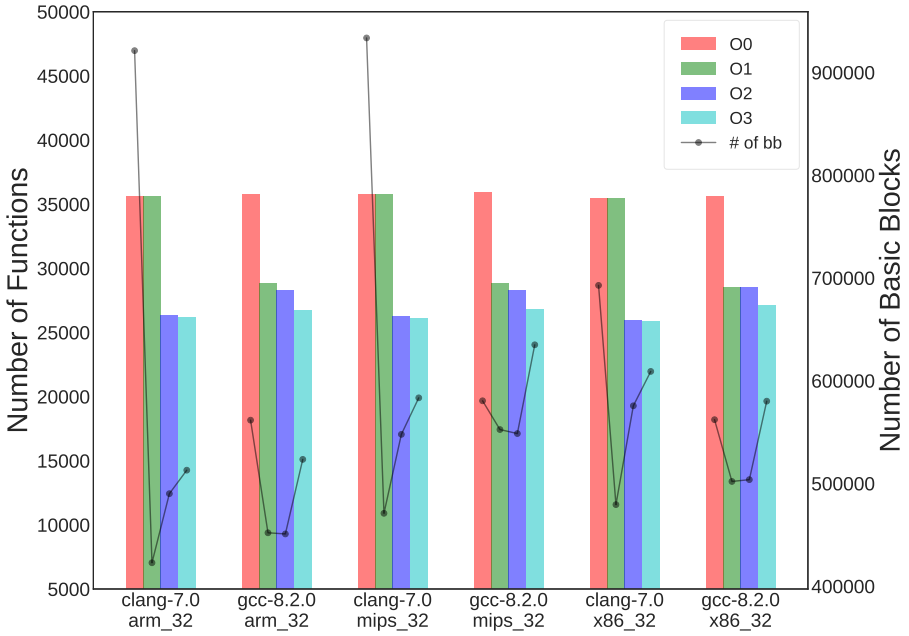


Figure 5.4: Final number of functions and basic blocks in NORMAL.

Moreover, we discovered that two compilers internally leverage different function-level code for specific operations. For example, GCC has functions, such as `__umoddl3` in `libgcc2.c` or `__aeabi_dadd` in `ieee754-df.S`, to optimize certain arithmetic operations. Furthermore, on x86, GCC generates a special function, such as `__x86.get_pc_thunk.bx`, to load the current instruction pointer to a register, whereas Clang inlines this procedure inside the target function. These functions can largely affect the call-related features, such as the number of control transfer instructions or that of outgoing calls. Although we removed these compiler-specific functions not to include them in our experiments (§5.3.2), they may have been inlined in their caller functions in higher optimization levels (`O2`–`O3`). Considering such functions took approximately 4% of the identified functions by IDA Pro, they may have affected the resulting features.

Similarly, the two compilers also utilize different instruction-level code. For example, in the case of move instructions for ARM, GCC uses conditional instructions, such as `MOVLE`, `MOVGT`, or `MOVNE`, unless the optimization level is zero (`O0`). In contrast, Clang utilizes regular move instructions along with branch instructions. This significantly affects the number of instructions as well as that of basic blocks in the resulting binaries. Consequently, in such special cases, the functions compiled using GCC have a relatively smaller number of basic blocks compared with those using Clang.

Finally, compilers sometimes generate multiple copies of the same function for optimization purposes. For example, they conduct inter-procedural scalar replacement of aggregates, removal of unused parameters, or optimization of cache/memory usage. Consequently, a compiled binary can have multiple functions that share the same source code but have different binary code. We found that GCC and Clang operate differently on this. Specifically, we discovered three techniques in GCC, which produce function copies with special suffixes, such as `.part`, `.cold`, or `.isra`. For instance, for the `get_data` function of `readelf` in `binutils` (in `O3`), GCC yields three copies with the `.isra` suffix, while Clang does not produce any of such functions. Similarly, for the `tree_eval` and `expr_eval` functions in `bool` (in `O3`), GCC produces two copies with the `.cold` suffix, but Clang does not. Although we selected only one of such copies in our experiments to avoid biased results (§5.3.2), the other copies can still survive in their caller functions by inlining.

Table 5.10: Number of functions and basic blocks in the NORMAL dataset for each compiler option.

Options			# of Functions				# of Basic Blocks			
Comp.*	Arch	Bit	O0	O1	O2	O3	O0	O1	O2	O3
Clang	arm	32	35,610	35,638	26,325‡	26,210	921,181†	422,807	490,029	512,890
GCC	arm	32	35,798	28,817‡	28,270	26,755	561,575	451,965	450,850	523,404
Clang	arm	64	35,612	35,637	26,106	26,005	897,892	466,108	555,294	584,098
GCC	arm	64	35,790	28,701	28,155	26,605	566,364	490,574	488,070	576,724
Clang	mips	32	35,723	35,741	26,221	26,130	933,529	470,923	547,755	583,302
GCC	mips	32	35,898	28,832	28,311	26,778	580,432	552,430	548,533	634,923
Clang	mips	64	35,679	35,701	26,222	26,095	921,484	460,491	534,209	569,074
GCC	mips	64	35,775	28,732	28,218	26,638	560,243	547,273	533,655	618,634
Clang	mipseb	32	35,721	35,741	26,217	26,136	933,654	470,795	547,752	583,189
GCC	mipseb	32	35,895	28,831	28,305	26,775	580,544	554,791	549,768	635,195
Clang	mipseb	64	35,676	35,700	26,220	26,093	922,077	460,531	534,528	569,063
GCC	mipseb	64	35,772	28,728	28,208	26,635	560,230	547,265	533,678	618,663
Clang	x86	32	35,466	35,484	25,974	25,878	692,755	479,383	575,554	609,008
GCC	x86	32	35,602	28,543	28,476	27,074	562,037	501,925	503,681	580,059
Clang	x86	64	34,127	34,202	25,593	25,482	640,058	444,199	551,809	581,622
GCC	x86	64	35,837	28,803	28,749	27,308	567,578	499,713	503,899	592,185

* We show the numbers for GCC v8.2.0 and Clang v7.0 for clear comparison.

† Clang inserts dummy basic blocks for the `O0` option on ARM and MIPS.

‡ GCC starts function inlining from `O1`, but Clang does from `O2`.

In summary, the diversities of compiler back-ends can largely affect the performance of BCSA, by making the resulting binaries divergent. Here, we introduced the major issues we discovered. We encourage further studies to investigate the implication of detailed options in each optimization level across different compilers.

5.7.3 Architecture-Specific Code

When manually inspecting failures, we found that some packages have architecture-specific code snippets guarded with conditional macros such as `#if` and `#ifdef` directives. For example, various functions in `OpenSSL`, such as `mul_add` and `BN_UMULT_HIGH`, are written in architecture-specific inline assembly code to generate highly optimized binaries. This means that a function may correspond to two or more distinct source lines depending on the target architecture.

Therefore, instruction-level presemantic features can be significantly different across different architectures when the target programs have architecture-specific code snippets, and one should consider such code when designing cross-architecture BCSA techniques.

5.8 Additional Results for Compiler Optimization

One of the popular goals in BCSA is to identify similar functions compiled from the same source but with different architectures or compile options. However, as shown in §5.5.1, compiler optimization is a significant factor that affects the presemantic features of the resulting binary code. Moreover, diversities in the implementation logic in each compiler also increase the implication of optimization (§5.7.2). In this section, we detail the number of functions and basic blocks for each compiler option and further discuss them.

Table 5.10 presents the number of functions and basic blocks in the NORMAL dataset. Here, we show the numbers for the latest version of GCC and Clang, which is v8.2.0 and v7.0, respectively, for comparison. For both compilers, the number of functions significantly decreases for higher optimization levels due to function inlining.

Meanwhile, the number of basic blocks does not decrease, as basic blocks can survive in caller functions although function inlining is applied. The number even increases as the optimization level increases from `O2` to `O3` for both compilers. By analyzing the cases, we confirmed that one possible reason is loop unrolling, which unwinds the loops and generates multiple copies of basic blocks. Consequently, the number of basic blocks for `O3` reaches higher than that for `O2`.

In this chapter, we have described multiple optimization issues that significantly affect the resulting binary code and presemantic features. However, we believe that there exist remaining issues for detailed optimization techniques. Therefore, we conclude by encouraging further studies to investigate the implication of detailed options in each optimization level across different compilers.

5.9 Discussion

Our study identifies several future research directions in BCSA. First, many BCSA papers have focused on building a general model that can result in stable outcomes with any compiler options. However, one could train a model targeting a specific set of compiler options, as shown in our experiment, to enhance their BCSA techniques. It is evident from our experiment's results that one can easily increase the success rate of their technique by inferring the compiler options used to compile the target binaries. There exists such an inference technique [250], and combining it with existing BCSA methods is a promising research direction.

Second, there are only a few studies on utilizing decompilation techniques for BCSA. However, our study reveals the importance of such techniques, and thus, invites further research on leveraging them for BCSA. One could also conduct a comprehensive analysis on the implication of semantic features along with decompilation techniques.

Additionally, we investigated fundamental presemantic features in this study. However, the effectiveness of semantic features is not well-studied yet in this field. Therefore, we encourage further research on investigating the effectiveness of semantic features along with other presemantic features that are not covered in the study. In particular, as many recent studies have been adopting NLP techniques, inspecting their effectiveness would be another essential study.

Our scope is limited to a function-level analysis (§5.4.1). However, one may extend the scope to handle other BCSA scenarios to compare binaries [174, 63, 147] or a series of instructions [154, 177, 152]. Additionally, one can extend our approach for various purposes such as vulnerability discovery [54, 71, 55, 63, 179, 251, 148], malware detection [66, 252, 65, 253, 254, 255, 256], library function identification [204, 257, 258, 191, 259, 260], plagiarism/authorship detection [68, 202, 261], or patch identification [262, 263, 264]. However, extending our work to other BCSA tasks may not be directly applicable. This is because it requires additional domain knowledge to design an appropriate model that fits the purpose and careful consideration of the trade-offs. We believe that the reported insights in this study can help this process.

Recall from Chapter 2, we did not intend to completely survey the existing techniques, but instead, we focused on systematizing the fundamental features used in previous literature. Furthermore, our goal was on investigating underexplored research questions in the field by conducting a series of rigorous experiments. For a complete survey, we refer readers to the recent survey on BCSA [265].

Finally, because our focus is on comparing binaries without source code, we intentionally exclude similarity comparison techniques that require source code. Nevertheless, it is noteworthy that there has been plentiful literature on comparing two source code snippets [266, 267, 268, 269, 270, 271, 272, 273, 274, 195] or comparing source code snippets with binary code snippets [275, 276, 277].

5.10 Conclusion

We studied previous BCSA literature in terms of the the features and benchmarks used. We discovered that none of the previous BCSA studies used the same benchmark for their evaluation, and that some of them required manually fabricating the ground truth for their benchmark. This observation inspired us to develop BINKIT, the first large-scale public benchmark for BCSA, along with a set of automated build scripts. Additionally, we developed a BCSA tool, TIKNIB, that employs an interpretable model. Using our benchmark and tool, we answered less-explored research questions regarding the syntactic and structural BCSA features. Several elementary features have been shown to be robust across multiple architectures, compiler types, compiler versions, and even intra-procedural obfuscation. Further, we proposed potential strategies for enhancing BCSA. We conclude by inviting further research on BCSA using our findings and benchmark.

Table 5.11: Number of binaries, original functions, and filtered (final) functions in BINKIT.

		NORMAL			SIZEOPT			PIE*			NOINLINE			LTO*			OBFUSCATION		
			# of Functions			# of Functions			# of Functions			# of Functions			# of Functions			# of Functions	
Package Name	**Ver.**	**# of Bins**	**Orig.**	**Final**	**# of Bins**	**Orig.**	**Final**	**# of Bins**	**Orig.**	**Final**	**# of Bins**	**Orig.**	**Final**	**# of Bins**	**Orig.**	**Final**	**# of Bins**	**Orig.**	**Final**
a2ps	4.14	576	364K	251K	144	88K	60K	576	364K	251K	576	402K	287K	576	244K	141K	256	166K	116K
binutils	2.30	4K	7,752K	1,032K	1K	1,847K	240K	4K	7,748K	1,033K	4K	9,113K	1,266K	4K	5,388K	1,123K	2K	3,615K	486K
bool	0.2.2	288	39K	15K	72	10K	3K	288	40K	15K	288	42K	16K	288	35K	11K	128	18K	7K
ccd2cue	0.5	288	44K	8K	72	11K	2K	288	44K	8K	288	45K	8K	288	39K	5K	128	20K	4K
cflow	1.5	288	151K	85K	72	36K	20K	288	151K	85K	288	173K	106K	288	109K	51K	128	70K	40K
coreutils	8.29	30K	10,396K	507K	8K	2,535K	105K	-	-	-	30K	11,592K	733K	2K	381K	23K	13K	4,744K	251K
cpio	2.12	576	303K	102K	144	76K	25K	576	303K	102K	576	341K	127K	576	228K	68K	256	138K	48K
cppi	1.18	288	85K	32K	72	22K	8K	288	85K	32K	288	91K	37K	288	56K	12K	128	39K	15K
dap	3.10	1K	105K	26K	288	26K	6K	1K	108K	26K	1K	105K	26K	1K	94K	15K	512	46K	11K
datamash	1.3	288	153K	80K	72	37K	18K	288	153K	80K	288	174K	99K	-	-	-	128	71K	38K
direvent	5.1	288	222K	120K	72	54K	29K	288	222K	120K	288	239K	137K	288	156K	70K	128	101K	55K
enscript	1.6.6	864	225K	59K	216	56K	14K	864	225K	59K	864	235K	66K	864	195K	46K	384	101K	27K
findutils	4.6.0	2K	814K	210K	432	198K	48K	2K	815K	210K	2K	925K	269K	2K	603K	189K	768	373K	100K
gawk	4.2.1	288	406K	252K	72	99K	60K	288	403K	252K	288	488K	332K	288	354K	218K	128	195K	123K
gcal	4.1	1K	341K	155K	288	85K	38K	1K	343K	155K	1K	351K	160K	1K	325K	146K	512	151K	69K
gdbm	1.15	1K	317K	98K	288	78K	23K	1K	318K	98K	1K	331K	110K	-	-	-	512	144K	45K
glpk	4.65	576	596K	399K	144	145K	96K	576	596K	399K	576	643K	445K	-	-	-	256	244K	184K
gmp	6.1.2	288	273K	198K	72	67K	48K	288	273K	198K	288	291K	215K	-	-	-	128	123K	90K
gnu-pw-mgr	2.3.1	576	289K	81K	144	66K	16K	576	291K	81K	576	358K	119K	576	201K	50K	256	139K	42K
gnudos	1.11.4	576	188K	82K	144	47K	20K	576	188K	82K	576	189K	82K	-	-	-	256	83K	36K
grep	3.1	288	237K	133K	72	57K	31K	-	-	-	288	286K	180K	-	-	-	128	110K	64K
gsasl	1.8.0	288	125K	84K	72	31K	20K	288	125K	84K	288	131K	90K	-	-	-	128	56K	38K
gsl	2.5	1K	2,043K	1,694K	288	500K	412K	1K	2,044K	1,694K	1K	2,210K	1,851K	-	-	-	512	921K	770K
gss	1.0.3	576	82K	28K	144	20K	7K	576	82K	28K	576	86K	32K	-	-	-	256	36K	13K
gzip	1.9	288	112K	38K	72	28K	9K	-	-	-	288	122K	47K	-	-	-	128	50K	18K
hello	2.10	288	65K	20K	72	17K	6K	288	65K	20K	288	67K	22K	288	43K	6K	128	29K	9K
inetutils	1.9.4	5K	2,083K	267K	1K	507K	65K	5K	2,086K	267K	5K	2,271K	309K	5K	1,719K	260K	2K	946K	122K
libiconv	1.15	864	164K	89K	216	40K	21K	864	164K	89K	864	175K	101K	-	-	-	384	75K	42K
libidn2	2.0.5	288	71K	23K	72	17K	5K	288	71K	23K	288	78K	31K	-	-	-	128	33K	12K
libmicrohttpd	0.9.59	288	109K	46K	72	27K	11K	288	109K	46K	288	115K	52K	-	-	-	128	50K	22K
libosip2	5.0.0	576	303K	188K	144	76K	46K	576	303K	188K	576	312K	196K	-	-	-	256	135K	85K
libtasn1	4.13	1K	158K	35K	288	39K	8K	1K	160K	35K	1K	167K	42K	-	-	-	512	69K	16K
libtool	2.4.6	288	66K	28K	72	16K	7K	288	66K	28K	288	69K	30K	-	-	-	128	30K	13K
libunistring	0.9.10	288	260K	180K	72	61K	41K	288	260K	180K	288	297K	215K	-	-	-	128	179K	86K
lightning	2.1.2	288	131K	101K	72	29K	22K	288	131K	101K	288	175K	144K	-	-	-	128	85K	48K
macchanger	1.6.0	288	37K	7K	72	9K	2K	288	37K	7K	288	38K	8K	288	32K	4K	128	16K	3K
nettle	3.4	1K	92K	11K	288	23K	2K	1K	95K	11K	1K	95K	13K	-	-	-	512	41K	5K
patch	2.7.6	288	224K	110K	72	54K	25K	288	224K	110K	288	261K	147K	288	168K	71K	128	104K	53K
plotutils	2.6	864	168K	36K	216	42K	9K	864	170K	36K	864	171K	37K	864	145K	24K	384	75K	16K
readline	7.0	576	353K	168K	144	88K	42K	-	-	-	576	375K	186K	-	-	-	256	158K	77K
recutils	1.7	3K	1,776K	241K	720	446K	58K	3K	1,777K	241K	3K	1,984K	277K	-	-	-	1K	812K	111K
sed	4.5	288	188K	96K	72	46K	22K	-	-	-	288	225K	131K	-	-	-	128	88K	46K
sharutils	4.15.2	1K	581K	98K	288	136K	21K	1K	583K	97K	1K	710K	136K	1K	404K	57K	512	278K	49K
spell	1.1	288	35K	3K	72	9K	864	288	36K	3K	288	36K	4K	288	33K	3K	128	16K	2K
tar	1.30	576	556K	300K	144	135K	70K	576	555K	300K	576	661K	388K	576	439K	235K	256	257K	142K
texinfo	6.5	288	113K	47K	72	28K	11K	288	114K	47K	288	131K	63K	288	93K	35K	128	52K	22K
time	1.9	288	37K	6K	72	9K	1K	288	38K	6K	288	39K	8K	288	32K	3K	128	17K	3K
units	2.16	288	102K	37K	72	26K	9K	288	102K	37K	288	103K	37K	288	86K	26K	128	46K	16K
wdiff	1.2.2	288	62K	12K	72	15K	3K	288	62K	12K	288	65K	15K	288	50K	8K	128	28K	6K
which	2.21	288	41K	8K	72	10K	2K	288	42K	8K	288	43K	9K	288	37K	5K	128	19K	3K
xorriso	1.4.8	288	921K	784K	72	226K	190K	288	920K	783K	288	990K	851K	288	590K	473K	128	420K	358K
Total		68K	34,356K	8,708K	17K	8,350K	2,061K	36K	23,091K	7,766K	68K	38,617K	10,291K	25K	12,280K	3,375K	30K	15,809K	4,055K

BINKIT consists of 243,128 binaries from compiling 51 GNU packages with 1,352 distinct options. After building the ground truth (§5.3.2), we sanitized the dataset, which decreased the number of functions from 132,503,599 to 36,256,322.

* Certain packages in the PIE and LTO datasets were unable to be compiled due to architecture-specific code or incompatible dependencies.

Chapter 6. Towards Large-Scale IoT Vulnerability Analysis with BCSA

Detecting previously known vulnerabilities in a different device is a common occurrence in the IoT ecosystem. For example, CVE-2018-10106, a permission bypass vulnerability discovered in 2018 from D-Link wireless routers, has persisted in recent devices over the years (*e.g.*, CVE-2018-10106, CVE-2019-17506, CVE-2019-20213, CVE-2020-9376). Even the same vulnerability can be found in devices from different vendors; for instance, CVE-2018-7034 is found in TRENDnet devices. This *known vulnerability* issue often stems from the absence of secure development standards. To be specific, many IoT vendors lack an internal policy or process for developing secure software. As a result, they often 1) manage the versions of their devices and source code improperly, or 2) copy and paste buggy code without validating it. As a result, many IoT devices share similar/identical vulnerabilities and are continuously exploited by them [7, 8, 9, 10].

To identify such known vulnerabilities, emulated-based analysis can be used. They can identify the same (or similar) vulnerabilities in different devices by emulating firmware images of target devices and running PoC exploits of previously known vulnerabilities. Although this approach does not generate false positives, achieving successful firmware emulation requires overcoming obstacles, such as handling multiple architectures (*e.g.*, ARM, MIPS, PowerPC, Hexagon) or resolving dependency issues in various peripherals (*e.g.*, Camera, LED, MMIO access). Furthermore, it takes a long time to emulate the firmware and send requests for vulnerability testing. On the other hand, similarity-based analysis can be adopted. This approach extracts signatures or distinguishable patterns from previously known vulnerabilities and then searches for them in target firmware images. While this approach may generate false positives, it is significantly more scalable than the former approach (*i.e.*, emulation-based analysis) in IoT vulnerability analysis.

Scalability is critical for addressing known vulnerabilities in a large number of IoT devices. Therefore, we chose the latter approach (*i.e.*, similarity-based analysis) and investigated its effectiveness in analyzing IoT vulnerabilities using BCSA. We began by establishing a ground truth dataset, which we refer to as FirmKit;[1] FirmKit contains 1,142 firmware images of IoT devices, as well as addresses of 323 vulnerable functions. Then, we searched for BCSA tools to run on this dataset for analysis. However, none of the existing tools were readily available for this analysis, as they released neither their source code nor datasets. Therefore, we leveraged our previously developed tool, TikNib, to conduct our analysis and confirmed the effectiveness of BCSA in IoT vulnerability analysis. Additionally, we discovered several findings during our analysis that will aid future research in this field. This chapter details 1) how we used BCSA to conduct practical IoT vulnerability analysis and 2) the results of our experiments.

6.1 Existing Approaches of BCSA-based IoT Vulnerability Analysis

6.1.1 Problems of Existing Studies

During our literature study on BCSA, we found a prominent research trend in existing studies of vulnerability analysis. While BCSA is promising in discovering vulnerabilities [70, 58, 177, 179, 182, 57, 54, 71, 55, 56], only a few existing studies focused on IoT vulnerability analysis. In practice, only five of the 43 BCSA studies that we studied in Chapter 5 attempted to analyze multiple firmware images. Moreover, even these papers have two critical problems as follows.

[1] https://github.com/SysSec-KAIST/FirmKit

First, the current BCSA studies on IoT vulnerability analysis are limited in scope. As the firmware of IoT devices contains a variety of programs to perform their intended tasks, vulnerabilities can be discovered in any of those programs. For example, vulnerabilities can appear in 1) libraries or utility binaries (*e.g.*, `OpenSSL`, `wget`, `bash`, `vsftpd`), or 2) custom binaries (*e.g.*, CGI binaries). However, the majority of existing studies have focused on analyzing the former case, namely, vulnerabilities in libraries or utility binaries. None have focused on analyzing vulnerabilities in custom binaries. This trend of ignoring custom binaries may stem from the difficulties in establishing their dataset. In the case of libraries or utility binaries, because their source code is available publicly, their dataset can be established by compiling their source code with various compiler options and architectures. As a result, multiple samples of a target vulnerable function can be obtained and used to train a model for conducting BCSA. On the other hand, the source code for custom binaries is typically not available due to vendor restrictions. Therefore, establishing a vulnerability dataset for custom binaries requires a significant amount of effort for manual investigation.

Furthermore, we discovered that none of the existing BCSA studies provide tools that are readily applicable to IoT vulnerability analysis. Only 10 of the 43 studies released their source code, and only two of them [55, 56] support the ARM and MIPS architectures, which are used in the majority of IoT devices. Moreover, upon analyzing the source code for these two tools, we discovered that they lack complete source code, necessitating additional effort to make them run in practice. Therefore, conducting practical IoT vulnerability analysis using existing tools is not trivial.

6.1.2 Motivating Example using VulSeeker

While existing tools have a number of limitations, they may be effective in analyzing IoT vulnerabilities if we make them run in practice. To explore their effectiveness, we examined the performance of the state-of-the-art tool VulSeeker [56]. Although VulSeeker did not provide complete source code, it did provide a partial result of identifying a known vulnerability on a Linksys router; thus, we can infer VulSeeker's effectiveness based on this result. VulSeeker targeted CVE-2015-1791, a race condition vulnerability in `OpenSSL`. To build their dataset, the authors compiled a vulnerable version of `OpenSSL`, v1.0.1f, with a variety of compiler options and architectures. These include 48 combinations of six architectures (x86, ARM, MIPS at 32 and 64 bits), two compilers (GCC v4.9.4 and v5.5.0), and four optimization levels (O0–O3). Then, for each compiled version of the vulnerable function, they employed their tool to perform the search task. Specifically, they measured the similarity score between the vulnerable functions they compiled and the target firmware image's functions. This results in 48 similarity scores for each function in the firmware. They then averaged these scores and sorted the results in descending order. Consequently, VulSeeker detected the vulnerable function at top-21 in the target firmware.

To further investigate this result, we repeated the same experiment using our BCSA tool TIKNIB with presemantic features, which we developed in Chapter 5. Notably, TIKNIB ranked the vulnerable function as the first, *i.e.*, TIKNIB detected it at top-1. From this example, we noticed that the state-of-the-art tool may not be sufficient to analyze IoT device vulnerabilities in practice. In this regard, we conducted an empirical study to determine how BCSA can be applied to practical IoT vulnerability analysis.

6.2 Enabling Practical Large-Scale IoT Vulnerability Analysis

To apply BCSA to IoT vulnerability analysis and evaluate its practical utility, we employed our interpretable model, TIKNIB, and the presemantic features described in Chapter 5. For this analysis, we mainly focused on vulnerabilities in custom binaries, on which none of the previous BCSA studies have investigated. We selected

Table 6.1: Summary of FIRMKIT.

	Linux-based IoT Devices	Smartphone Baseband
# of Firmware Images	1,124	18
# of Functions	52,086,995	1,405,959
# of Command Injection (Custom)	98 (8)	·
# of Information Leak (Custom)	162 (6)	·
# of Buffer Overflow (Custom)	7 (5)	38 (3)
# of Uninitialized Pointer Dereference (Custom)	·	18 (1)
Total # of Vulnerable Functions in Custom Binaries	267 (19)	56 (4)

The unique number of vulnerabilities is indicated in the parentheses.

to analyze the firmware images of wireless routers and IP cameras (used in Chapter 3) and those of smartphone basebands (used in Chapter 4). Using these firmware images, we analyzed vulnerabilities in 1) simple custom binaries, such as CGI binaries in wireless routers and IP cameras, and 2) complex custom binaries, such as monolithic firmware binaries in smartphone basebands. Because there is no ground truth dataset for the vulnerabilities in these firmware images, we had to establish one first. Then, using our heuristic knowledge, *i.e.*, our experience in analyzing IoT device binaries, we developed effective features. Lastly, we investigated the efficacy of BCSA by utilizing TIKNIB and these heuristic features.

6.2.1 Establishing Ground Truth Dataset (FIRMKIT)

To build the ground truth dataset, dubbed FIRMKIT, we manually marked the addresses of vulnerable functions identified in our previous studies described in Chapter 3 and Chapter 4. The ground truth dataset is summarized in Table 6.1. The numbers in parentheses indicate the number of unique vulnerabilities. In the case of Linux-based IoT devices, such as wireless routers and IP cameras, because we discovered the vulnerabilities through emulation-based dynamic analysis, we do not know their function addresses. Therefore, for each vulnerability, we manually analyzed the binaries in the firmware images using IDA Pro [101] and obtained the addresses of the vulnerable functions. We excluded functions that IDA Pro was unable to analyze. As a result, the final number of vulnerabilities differs from the number discovered in Chapter 3. On the other hand, because we manually discovered the vulnerabilities in the firmware binaries of smartphone basebands, we already know their addresses. Using this information, we constructed our ground truth dataset.

6.2.2 Developing Heuristic Features

We randomly selected one sample for each unique vulnerability and queried it in each firmware image by employing TIKNIB with the presemantic features. Then, we determined the rank of the vulnerable function in the firmware in descending order of similarity score. The result was not surprising as only 52.81% of vulnerabilities in Linux-based IoT devices were detected at top-1, *i.e.*, ranked at the first position. In the case of smartphone baseband, TIKNIB detected only 64.29% at top-1.

To improve the performance for practical analysis, we looked into failure cases of vulnerability discovery and discovered two major issues. First, because we used a single sample for each unique vulnerability, ground truth functions compiled with a different architecture than the input function showed low similarity scores. For example, when we searched for CVE-2015-2051, a command injection vulnerability in D-Link routers's URL parsing, we queried a sample function compiled for the ARM architecture in our dataset. The average rank of ground truth functions in ARM-based firmware images was 1.75, whereas those in MIPS-based firmware images was over 1,000.

By analyzing these cases, we discovered that the ARM and MIPS architectures behave differently when calling external functions. More precisely, in binaries compiled for the ARM architecture, library functions such as `memset` and `getenv` are invoked via a wrapper function in the `.plt` section. On the other hand, library functions in MIPS-compiled binaries are invoked directly without the use of a wrapper function. Consequently, features associated with function calls, such as the number of callees, the number of imported callees, or the size of the control flow graph, vary by architecture.

Second, owing to the version difference among the firmware images, newer images have different implementations than older images. For instance, in the case of CVE-2017-5521, a vulnerability in Netgear devices that allows for information leakage, newer devices include an additional routine that verifies the timestamp of the last recovery within the vulnerable function. Therefore, the majority of the features associated with this addition routine were affected severely.

To address these two issues, we developed features that are resistant to changes in architecture and versioning. We drew on our experience in analyzing IoT device binaries. Through empirical analysis of diverse binaries in our previous studies, we discovered that IoT binaries frequently contain function names. Thus, rather than comparing abstracted numeric features, we can directly compare caller and callee names. For example, we can use the names of internal and library functions instead of the numbers of callers and callees. Additionally, we discovered that data strings contained in IoT binaries often contain useful information. CGI binaries include hard-coded strings for parsing URLs, such as `HTTP`, `POST`, `answer`, or `password`. Therefore, we can use these strings to compute the similarity score.

Based on these observations, we developed two heuristic features that compare 1) the strings to which the function refers and 2) the names of the callee functions. To determine string similarity, we used a Jaccard index as follows. Consider the case where we have two functions, A and B. Then, using a whitespace delimiter, we split all strings that function A refers to and union the words into a single set called A_{str}. Similarly, we obtain a set of words for all strings referred to by function B, which we call B_{str}. Then, we compute the string similarity between these two functions in the following manner:

$$Jaccard(A_{str}, B_{str}) = \frac{|A_{str} \cap B_{str}|}{|A_{str} \cup B_{str}|} \tag{6.1}$$

To compare the callees of two functions, A and B, we refer to the set of callee names in function A as A_{str} and the set of callee names in function B as B_{str}. Then, using the same approach described above, we calculate their Jaccard index.

After computing the similarity score for each of these two heuristic features, we averaged them against the score computed by running naive TikNib, *i.e.*, TikNib with presemantic features.

6.3 Evaluation

As described previously, we integrated heuristic and presemantic features into TikNib Using this TikNib, we conducted a series of experiments to address the following questions:

- $Q1$: How do well-systematized heuristics identify vulnerabilities in simple custom binaries (*i.e.*, CGI binaries in Linux-based IoT devices)?
- $Q2$: How do well-systematized heuristics detect vulnerabilities in complex custom binaries (*i.e.*, smartphone baseband firmware binaries)?
- $Q3$: How do well-systematized heuristics identify vulnerabilities in open-source packages (*i.e.*, OpenSSL libraries in Linux-based IoT devices)?

Table 6.2: Top-k results of identifying vulnerabilities in FIRMKIT using TIKNIB.

Top-k	Linux-Based IoT Devices		Smartphone Cellular Baseband	
	w/o Heuristic Features	w/ Heuristic Features	w/o Heuristic Features	w/ Heuristic Features
1	141 / 267 (52.81%)	263 / 267 (98.50%)	36 / 56 (64.29%)	48 / 56 (85.71%)
5	167 / 267 (62.55%)	263 / 267 (98.50%)	41 / 56 (73.21%)	49 / 56 (87.50%)
10	182 / 267 (68.16%)	266 / 267 (99.63%)	41 / 56 (73.21%)	49 / 56 (87.50%)
50	196 / 267 (73.41%)	266 / 267 (99.63%)	42 / 56 (75.00%)	50 / 56 (89.29%)
100	196 / 267 (73.41%)	267 / 267 (100.0%)	44 / 56 (78.57%)	52 / 56 (92.86%)

- *Q*4: Is BCSA capable of discriminating between vulnerable and benign functions effectively?

6.3.1 Identifying Vulnerabilities in Custom Binaries

To test Q1 and Q2, we randomly selected one sample for each unique vulnerability in FIRMKIT (Table 6.1) and queried it against each firmware image. We measured the top-k similarity scores between the input vulnerable function and each function in the firmware image. As shown in Table 6.2, the heuristic features developed in §6.2.2 significantly improved the performance of original TIKNIB only with presemantic features, thereby supporting both Q1 and Q2. Notably, the majority of vulnerabilities were detected at the top-1 level of each firmware image, supporting Q4.

The detailed results of running TIKNIB with heuristic features against Linux-based IoT devices, such as wireless routers and IP cameras, are shown in Table 6.3. For each unique vulnerability, we selected a sample vulnerable function from the firmware image whose index is shown in parentheses, as shown in the first column of the table.[2] The second column in Table 6.3 indicates the range of similarity scores for the top-1 functions in the firmware images. The third column in Table 6.3 denotes the number of functions discovered by TIKNIB, and the fourth column indicates whether these functions are vulnerable (*i.e.*, *V*), patched (*i.e.*, *P*), not vulnerable (*i.e.*, *N*), or unknown (*i.e.*, *U*). To determine this, we manually analyzed and validated each function and binary. We mark functions as not vulnerable if their binaries do not contain a function related to the input vulnerability, and as unknown if the type of functions could not be determined. In the majority of cases, our system successfully distinguished vulnerabilities from other benign functions, thereby supporting Q4. Notably, our BCSA-based approach (*i.e.*, TIKNIB with heuristic features) discovered more vulnerabilities than an emulation-based approach (*i.e.*, FIRMAE); compare the numbers with and without parentheses in the third column.

In the following, we summarize our findings from analyzing the results along with example cases.

(1) BCSA is capable of identifying vulnerabilities and their associated patches.

We discovered that BCSA is capable of identifying vulnerabilities as well as their patches. Here, we will demonstrate two examples of vulnerabilities. CVE-2016-6277 is a command injection vulnerability in NETGEAR routers caused by an incorrect URI parsing routine. This vulnerability is easily patched by including a routine that checks for invalid characters in URIs. One may implement a blocklist that contains frequently exploited characters for shell command injection. In practice, we discovered that patches for this vulnerability are implemented by checking for special characters such as `';'`, `'`'`, `'$'`, and `'..'`. Notably, all vulnerable functions were ranked higher than patched functions, and the other functions unrelated to the vulnerability were ranked lower than the patched ones. This result demonstrates that TIKNIB with the heuristic features successfully distinguished vulnerable, patched, and unrelated functions, thereby supporting both Q1 and Q4.

[2]For more details, please refer to https://github.com/pr0v3rbs/FirmAE.

Table 6.3: Similarity analysis results of example vulnerabilities in Linux-based IoT devices.

				Vendor							Arch‡			
Vulnerability†	Range⇑	# of Images⇓	Vuln†	Netgear	D-Link	TRENDnet	Belkin	Asus	ZyXEL	Linksys	arm	mips	mipseb	Binary
CVE-2016-6277 (104)	0.95–1.00	29 (3)	V	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.5–0.95	40 (-)	P	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
CVE-2015-2051 (619)	0.81–1.00	5 (4)	V	·	✓	·	·	·	·	·	✓	·	·	/htdocs/cgibin
	0.68–0.73	25 (-)	P	·	✓	·	·	·	·	·	✓	·	·	/htdocs/cgibin
	0.58–0.75	6 (5)	V	·	✓	✓	·	·	·	·	·	✓	·	/htdocs/cgibin
	0.53–0.59	3 (-)	P	·	✓	·	·	·	·	·	·	✓	·	/htdocs/cgibin
	0.68	1 (-)	P	·	✓	·	·	·	·	·	·	·	✓	/htdocs/cgibin
	0.58–0.69	15 (14)	V	·	✓	·	·	·	·	·	·	·	✓	/htdocs/cgibin
	0.53	9 (-)	P	·	✓	·	·	·	·	·	·	·	✓	/htdocs/cgibin
	0.49–0.53	17 (-)	P	·	✓	·	·	·	·	·	·	✓	✓	/usr/sbin/upnpkits
CVE-2017-7240 (118)	0.95–1.00	3 (3)	V	·	·	·	✓	·	·	·	·	✓	·	/usr/sbin/httpd
	0.54–0.83	6 (-)	N	·	·	·	✓	·	·	·	·	✓	·	/usr/sbin/httpd
	0.50–0.53	23 (-)	N	·	·	·	·	✓	✓	✓	·	✓	✓	/usr/sbin/httpd
CVE-2018-10106 (2)	0.99–1.00	45 (42)	V	·	✓	✓	·	·	·	·	·	✓	✓	/htdcos/cgibin
	0.48–0.86	42 (41)	V	·	✓	·	·	·	·	·	✓	✓	✓	/htdocs/cgibin
	0.55–0.84	5 (-)	P	·	✓	·	·	·	·	·	✓	✓	✓	/htdocs/cgibin
CVE-2014-2962 (510)	0.96–1.00	2 (2)	V	·	·	·	✓	·	·	·	·	·	✓	/usr/www/cgi-bin/webproc
	0.66–0.86	13 (0)	V*	✓	·	✓	✓	·	·	·	·	·	✓	/usr/www/cgi-bin/webproc
	0.53	1 (-)	P	✓	·	·	·	·	·	·	·	✓	·	/usr/www/cgi-bin/webproc
CVE-2020-15893 (2)	0.86–1.00	43 (40)	V	·	✓	✓	·	·	·	·	·	✓	✓	/htdocs/cgibin
	0.96	1 (-)	P	·	·	✓	·	·	·	·	·	✓	✓	/htdocs/cgibin
	0.85	17 (12)	V	·	✓	·	·	·	·	·	·	✓	✓	/usr/sbin/upnpkits
	0.82	7 (7)	V	·	✓	·	·	·	·	·	✓	·	·	/htdocs/cgibin
	0.74–0.81	42 (-)	P	·	✓	·	·	·	·	·	✓	✓	✓	/htdocs/cgibin
	0.52	1 (1)	V⋆	·	✓	·	·	·	·	·	·	·	✓	/htdocs/cgibin
CVE-2016-11021 (804)	0.97–1.00	11 (1)	V	·	✓	·	·	·	·	·	·	✓	·	/bin/alphapd
	0.97	2 (2)	V	·	✓	·	·	·	·	·	·	✓	·	/bin/goahead
	0.67–0.75	21 (-)	P	·	✓	✓	·	·	·	·	·	✓	·	/bin/alphapd
	0.60–0.67	9 (0)	V▽	·	✓	·	·	·	·	·	·	✓	·	/bin/alphapd
	0.59	1 (-)	P	·	·	✓	·	·	·	·	·	✓	·	/bin/alphapd
	0.50–0.59	18 (-)	N	·	✓	·	·	·	·	·	·	✓	·	/bin/alphapd
CVE-2017-6077 (186)	0.85–1.00	2 (2)	V	✓	·	·	·	·	·	·	·	·	✓	/usr/sbin/httpd
	0.5–0.85	1 (0)	V◇	✓	·	·	·	·	·	·	·	·	✓	/usr/sbin/httpd
CVE-2012-2765 (37)	0.72–1.00	7 (3)	V	·	·	·	✓	·	·	·	·	✓	·	/usr/sbin/httpd
	0.66	1 (-)	P	·	·	·	✓	·	·	·	·	✓	·	/usr/sbin/httpd
	0.58	2 (0)	V	·	·	·	✓	·	·	·	·	✓	·	/usr/sbin/httpd
	0.53	1 (-)	N	·	·	·	·	·	·	✓	·	✓	·	/usr/sbin/httpd
Linksys Vuln. (53)	0.72–1.00	10 (1)	V♡	·	·	·	·	·	·	✓	·	✓	·	/usr/sbin/httpd
	0.53–0.64	7 (-)	P	·	·	·	·	·	·	✓	·	✓	·	/usr/sbin/httpd
CVE-2017-5521 (99, Stage 1)	0.98–1.00	40 (26)	V	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.74–0.83	73 (-)	P	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.79	2 (0)	V*	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.51–0.52	11 (9)	V	✓	·	·	·	·	·	·	·	✓	·	/usr/sbin/httpd
	0.51–0.59	171 (-)	U♠	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	22 different binaries
CVE-2017-5521 (99, Stage2)	0.98–1.00	79 (26)	V	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.76–0.92	36 (-)	P	✓	·	·	·	·	·	·	✓	·	·	/usr/sbin/httpd
	0.74–0.78	24 (6)	V	✓	·	·	·	·	·	·	·	✓	·	/usr/sbin/httpd
	0.68–0.73	9 (-)	P	✓	·	·	·	·	·	·	·	✓	·	/usr/sbin/httpd
	0.51–0.53	3 (-)	N	✓	·	·	·	·	·	·	·	·	✓	/usr/sbin/httpd
	0.51–0.51	1 (-)	P	✓	·	·	·	·	·	·	·	✓	·	/usr/sbin/upnpd
	0.51–0.51	14 (3)	V	✓	·	·	·	·	·	·	·	✓	·	/usr/sbin/upnpd
Total Vulnerabilities		442 (253)												

† V: vulnerable, P: patched, N: not vulnerable (*i.e.*, no vulnerable functionality), U: unknown.

‡ All architectures are 32-bit based.

† The number in parentheses represents the index of input firmware images. For more details, please refer to https://github.com/pr0v3rbs/FirmAE.

⇑ We considered only the top-1 function in each firmware image.

⇓ The number in parentheses denotes the number of vulnerable functions identified by FIRMAE as the ground truth.

* These functions are potentially vulnerable. Due to emulation failure, we were not able to determine whether these functions are actually called at runtime.

⋆ Because these functions have a dedicated debugging routine, the similarity score is low even though they are vulnerable.

▽ The binaries containing these functions are statically compiled, which degraded the similarity score.

◇ These functions get an additional parameter for VPN.

♡ Five of 10 vulnerable functions are actually vulnerable. The remaining five are not invoked because their function calls were removed by the developers.

♠ The binaries containing these functions are included in older firmware images; thus, they use different implementations.

(2) BCSA can identify and distinguish vulnerabilities against different architectures and binaries.

CVE-2015-2051 is another command injection vulnerability in D-Link routers. As with CVE-2016-6277, this vulnerability originates from an incorrect URI parsing routine. We discovered that the devices affected by this vulnerability use three different architectures: 32-bit ARM, MIPS, and MIPSeb. For each architecture, our system successfully identified vulnerable functions and also distinguished them from patched and other benign functions.

Notably, the vulnerability that we queried was in a binary located at `/htdocs/cgibin`. Our system, however, detected functions associated with this vulnerability in a different binary located at `/usr/sbin/upnpkits` with a similarity score of 0.49–0.53, as shown in the last row of CVE-2015-2051 in Table 6.3. Although these functions have been patched, this result demonstrates that BCSA is capable of successfully identifying vulnerabilities against different binaries.

(3) BCSA can detect differences in compilation environments.

CVE-2017-7240 and CVE-2012-2765 are two examples of vulnerabilities presenting BCSA's ability to detect differences in compilation environments. CVE-2017-7240 is a directory traversal vulnerability in Belkin devices' URL parsing. When a CGI binary parses a URL, it checks a list of allowed file types to ensure that the files are not accessible to an unauthorized user. However, devices that contain this vulnerability allow any file type, *i.e.*, accept '$*$'; thus, any unauthorized user can retrieve any files on the devices. Typically, the allowed list is located in the data section of a CGI binary. Therefore, by conducting this experiment, we can assess BCSA's capability for detecting vulnerabilities that originate in the data section.

To discover this vulnerability, we queried the function that refers to the vulnerable allowed list. Devices that are not vulnerable have a similar function for checking an allowed list. They do, however, use a different allowed list that accepts specific file types but not all. The functions that use the vulnerable allowed list were ranked higher than those that use this non-vulnerable allowed list. When modifying the allowed list, developers may also have modified other code sections, which eventually varied the resulting binary. As a result, even though the vulnerability is located in the data section rather than the code section, BCSA can identify the function that refers to this data section and also differentiate between vulnerable and non-vulnerable functions.

CVE-2012-2765 is another example of a password disclosure vulnerability. This vulnerability originates from client-side password validation on a login page. When a user enters an ID and a password, the server sends a password to the client, which is then verified on the client-side. Patching this vulnerability is straightforward, as the checking routine can simply be moved to the server binary. To identify this vulnerability, we queried the function that sends the password, in the server binary located at `/usr/sbin/httpd`. Notably, BCSA successfully identified both the vulnerable and patched functions, as patching the vulnerability would also change our target binary during compilation, as in the previous case. These two examples demonstrate that BCSA is capable of detecting differences in compilation environments.

(4) Identical vulnerabilities frequently appear in different vendors

We discovered that many devices from distinct vendors share an identical vulnerability. For example, when we queried functions for CVE-2015-2051 and CVE-2018-10106, we identified that both D-Link and TRENDnet devices contained the same (or similar) vulnerable functions. This demonstrates the possibility that these two vendors may share the same (or similar) codebase. On the other hand, we found multiple devices that contain similar functions although the functions are not vulnerable. CVE-2017-7240 affects devices from multiple vendors, such as Belkin, Asus, ZyXEL, and Linksys. Among these devices, only a few Belkin devices were vulnerable. As a result, we deduced that IoT device vendors may share the same or similar codebases.

(5) Recurring vulnerabilities are frequently discovered in newer devices and are assigned multiple CVE numbers.

We found that older vulnerabilities persist in newer devices, and that some of these vulnerabilities have even been assigned new CVE numbers. CVE-2018-10106, for example, is a permission bypass vulnerability in D-Link devices. However, the same vulnerability exists in more recent D-Link devices and has been assigned three distinct CVE numbers: CVE-2019-17506, CVE-2019-20213, and CVE-2020-9376. The same vulnerability was also discovered and assigned the name CVE-2018-7034 on TRENDnet devices.

Another example is CVE-2014-2962, for which the initial vulnerability was discovered in 2006. The same vulnerability has been assigned to D-Link devices as CVE-2006-2337 and CVE-2006-5536, Inca devices as CVE-2006-5607, Belkin devices as CVE-2014-2962, Zte devices as CVE-2015-7250, Fiberhome devices as CVE-2017-15647, and Twsz devices as CVE-2017-8770. These findings indicate that many IoT devices share the same or similar source code, and that IoT developers do not properly manage the versions of device source code.

(6) Certain devices from a single vendor include debugging routines.

During the analysis, we discovered that some devices contain debugging routines that should be removed in the release version. Such cases may originate from incorrect version management in the development stage. CVE-2020-15893, for example, is a command injection vulnerability in the parsing of simple service discovery protocol (SSDP) parameters on D-Link and TRENDnet devices. Among the functions associated with the vulnerability, one vulnerable function has a lower similarity score than the patched ones, as shown in the last row of CVE-2020-15893 in Table 6.3. Unlike other functions related to this vulnerability including those that have been patched, this one includes a special debugging routine that invokes the `_dtrace()` function. As a result, this debugging function had an effect on the call-related features, resulting in the low similarity score.

When analyzing CVE-2012-2765, we discovered a similar case. CVE-2012-2765 is a information leakage vulnerability in Belkin devices that is caused by client-side password checking, as previously described. This case is distinct from CVE-2020-15893 in that the vulnerable function sampled included a debugging routine. Therefore, two vulnerable functions that lack a debugging routine had lower similarity scores (*i.e.*, 0.58) than the patched function (*i.e.*, 0.66). As shown in these two examples, when conducting BCSA-based IoT vulnerability analysis, debugging routines should be considered.

(7) While the majority of binaries are dynamically compiled, some are statically.

While analyzing CVE-2016-11021, we discovered that some of the binaries were statically compiled; however, all of the other binaries we analyzed were dynamically compiled. CVE-2016-11021 is a command injection vulnerability in a debugging feature on D-Link and TRENDnet devices. When a binary is statically compiled, all library functions linked to the binary are inserted into the binary; thus, the inserted functions can affect the binary's original functions, most notably during optimization. As discussed in Chapter 5, compiler optimization can have a significant impact on the resulting BCSA features due to function inlining. Consequently, vulnerable functions in statically compiled binaries were ranked lower than those in dynamically compiled binaries.

(8) Certain devices include binaries that have been customized from open-source projects

We found that IoT devices frequently use open-source projects to implement their firmware. By analyzing CVE-2017-7240 on Belkin devices, the vulnerability previously described, we observed that Belkin, Asus, ZyXEL, and Linksys all use the same URL parsing function. The vulnerability originates from an allowed list that accepts

any file type. By searching the strings used in the allowed list on the web, we discovered that the firmware images for these devices were implemented based on DD-WRT, an open-source IoT firmware project. Notably, the vulnerability exists only in a subset of Belkin devices, and we discovered that those devices had the allowed list incorrectly customized to accept any file type.

Additionally, when analyzing CVE-2016-11021, a command injection vulnerability discovered in D-Link and TRENDnet devices, we discovered a similar case. We observed that the majority of the vulnerabilities were discovered in binaries located at `/bin/alphapd`, whereas two were discovered in a different binary located at `/bin/goahead`, as shown in the CVE-2016-11021 rows in Table 6.3. We discovered that the `alphapd` binaries were implemented based on `GoAhead`, an open-source web server for embedded devices. As the majority of these binaries were implemented based on older versions of `GoAhead`, they all inherit the same vulnerability.

(9) BCSA is effective at analyzing patches.

As demonstrated in Table 6.3, BCSA is effective at rapidly identifying vulnerabilities and their patches. Given that our system successfully identified functions associated with the vulnerability, all that remained was to determine whether or not the function was vulnerable. This significantly reduced our manual analysis effort. The *V* and *P* values in the fourth column of Table 6.3 denote whether the functions are vulnerable or patched. Additionally, we discovered that some patches were applied incorrectly or were insufficient to completely eliminate the vulnerability. The following discusses such cases.

To begin, we discovered that some patches for CVE-2017-5521 had been applied incorrectly. CVE-2017-5521 is a information leakage vulnerability in NETGEAR devices that allows an unauthorized user to obtain the device ID. One straightforward patch would be adding a routine that verifies the authorized status of a user who sends requests prior to printing out the device ID. However, we found two patches that had been applied incorrectly. These patches do verify authorization; however, they simply remove an HTML element on the client-side if a user is not authorized. Therefore, the device ID transferred from the server can still be exposed to the unauthorized user.

Additionally, when analyzing CVE-2012-2765 from Belkin devices, we found that certain patches were applied differently even though firmware images were released on the same date. The second and third rows of Table 6.3 demonstrate that one firmware image was patched while the other two were not, despite the fact that all three firmware images were released on the same date, February 2018.

Moreover, multiple patch versions were discovered for a single vulnerability. There were three patches distributed across 17 devices for the Linksys vulnerability, which is a command injection vulnerability for which we were unable to locate a CVE number. Seven devices were patched by adding a routine to the vulnerable function that validates input strings; the similarity scores of these patches functions are shown in the second Linksys Vuln. row of Table 6.3. Meanwhile, as shown in the first Linksys Vuln. row, the vulnerable functions in the remaining 10 devices were not patched. We investigated these vulnerable cases and discovered that, rather than directly patching the vulnerable function, the developers simply removed the function calls that invoked it. More precisely, two points invoke the vulnerable function: one before and one after authentication. To eliminate the vulnerability, the developers used two distinct patches. The first patch removed calls prior to authentication; thus, the vulnerability can still be exploited via the other function after authentication. Then, the second patch eliminated both points, thereby eradicating the vulnerability completely. We discovered that two of the 10 firmware images were vulnerable in practice due to their lack of patching, three were vulnerable after authentication due to the first patch, and five were patched correctly due to the second patch.

These examples demonstrate BCSA's capability and effectiveness in patch analysis. Furthermore, even if a binary contains a vulnerable function, additional analysis is required to determine whether the vulnerability can be exploited in practice.

Table 6.4: Top-k results of analyzing OpenSSL vulnerabilities in Linux-based firmware images using TIKNIB.

Top-k	CVE-2014-0160		CVE-2015-1791	
	w/o Heuristic Features	w/ Heuristic Features	w/o Heuristic Features	w/ Heuristic Features
1	4 / 34 (11.76%)	34 / 34 (100.0%)	252 / 309 (81.55%)	309 / 309 (100.0%)
5	17 / 34 (50.00%)	34 / 34 (100.0%)	284 / 309 (91.91%)	309 / 309 (100.0%)
10	17 / 34 (50.00%)	34 / 34 (100.0%)	293 / 309 (94.82%)	309 / 309 (100.0%)
50	32 / 34 (94.12%)	34 / 34 (100.0%)	294 / 309 (95.15%)	309 / 309 (100.0%)
100	34 / 34 (100.0%)	34 / 34 (100.0%)	294 / 309 (95.15%)	309 / 309 (100.0%)

Table 6.5: Top-k results of analyzing OpenSSL vulnerabilities, including the patched ones, in Linux-based firmware images using TIKNIB.

Top-k	CVE-2014-0160		CVE-2015-1791	
	w/o Heuristic Features	w/ Heuristic Features	w/o Heuristic Features	w/ Heuristic Features
1	7 / 222 (3.15%)	215 / 222 (96.85%)	252 / 455 (55.38%)	455 / 455 (100.0%)
5	29 / 222 (13.06%)	215 / 222 (96.85%)	284 / 455 (62.42%)	455 / 455 (100.0%)
10	44 / 222 (19.82%)	222 / 222 (100.0%)	293 / 455 (64.40%)	455 / 455 (100.0%)
50	110 / 222 (49.55%)	222 / 222 (100.0%)	337 / 455 (74.07%)	455 / 455 (100.0%)
100	158 / 222 (71.17%)	222 / 222 (100.0%)	382 / 455 (83.96%)	455 / 455 (100.0%)

(10) BCSA's performance degrades when the target function's size is too small.

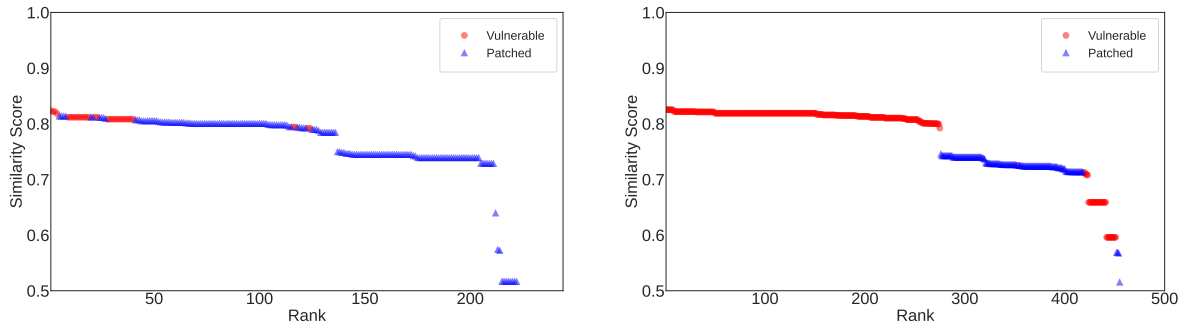
Although we were unable to observe this from Linux-based IoT devices, it is worth noting that the size of an input vulnerable function can have a significant effect on BCSA performance. If a target binary contains multiple small-sized functions, their BCSA features may be indistinguishable. Additionally, in different versions of a target device, such functions may include an additional routine to a target function that may take a larger portion than the original code. As a result, the similarity score of these functions can be severely affected.

We observed this case in the firmware binaries of smartphone basebands, where four of 56 vulnerabilities were not detected within the top-100, as shown in the right part of Table 6.2. Notably, these four vulnerabilities are all related to a single, identical vulnerability that can result in a buffer overflow. There are six ground truth functions for this vulnerability. Our system identified two of the six ground truth functions in older devices; however, it was unable to identify four in newer devices that included an additional routine to support dual-SIM functionality. Because baseband contains a large number of small functions for processing protocol messages, such a change may take a larger portion in a function than the original code. As a result, the similarity scores of four ground truth functions in newer devices were decreased.

6.3.2 Identifying Vulnerabilities in Open-Source Packages

To test Q3, we conducted additional experiments aimed at identifying `OpenSSL` vulnerabilities in the firmware images of wireless routers and IP cameras. In these experiments, we also employed TIKNIB equipped with the heuristic features. We selected CVE-2015-1791 and CVE-2014-0160 as the vulnerabilities to evaluate because they have been frequently used in previous studies [56, 71, 55, 70, 54, 64]. CVE-2015-1791 is a race condition vulnerability in the `ssl3_get_new_session_ticket()` function, while CVE-2014-0160 is an information leakage vulnerability in the `tls1_process_heartbeat()` function. As we do not have the ground truth, we first had to determine the addresses of vulnerable functions in the firmware images. For this, we checked 1) whether a given firmware image contains a `libssl.so` binary, and 2) whether the binary's version is affected by the vulnerability. To check the binary version, we leveraged version strings inserted into the binary during compilation.

We conducted the experiments using the same methodology as VulSeeker [56], as previously described



(a) CVE-2014-0160 (34 of 222 functions are vulnerable).

(b) CVE-2015-1791 (309 of 455 functions are vulnerable).

Figure 6.1: Similarity scores of vulnerable and patched functions in Linux-based firmware images, obtained by running TIKNIB with heuristic features.

in §6.1.2. We compiled the vulnerable version of `OpenSSL` (*i.e.*, v1.0.1f) using various compiler options and architectures, including six architectures (x86, ARM, and MIPS, each at 32 and 64 bits), two compilers (GCC v4.9.4 and v5.5.0), and four optimization levels (O0–O3). As a result, we were able to obtain 48 samples for each of the two vulnerabilities. We compared each sample vulnerable function to the functions in each firmware image, resulting in 48 similarity scores for each firmware function. Then, we calculated the top-k result by averaging these similarity scores for each function. Table 6.4 shows the top-k result of identifying only the vulnerable functions from the firmware images, whereas Table 6.5 presents that of identifying the vulnerable functions as well as their patched ones from the firmware images. In all experiments, TIKNIB with the heuristic features successfully identified the target functions, corroborating Q3.

Additionally, we compared the similarity scores of vulnerable and patched functions to explore Q4. As illustrated in Figure 6.1, vulnerable functions (*i.e.*, the red) have a higher similarity score than patched functions (*i.e.*, the blue), supporting Q4. Meanwhile, in the case of CVE-2015-1791, some of the vulnerable functions have lower similarity scores than those that have been patched. By investigating them, we discovered that these functions were either 1) compiled from older versions of `OpenSSL`, such as v0.9.8k or v0.9.8zc, or 2) statically compiled into another binary, such as `/bin/curl`. However, within similar versions of `OpenSSL`, TIKNIB with heuristic features successfully distinguished vulnerable functions from patched functions.

6.3.3 Comparison Against State-of-the-Art Techniques

In previous experiments, we have demonstrated the effectiveness of TIKNIB with heuristic features in analyzing IoT device vulnerabilities. Next, we evaluate our system against state-of-the-art techniques. Among the 43 existing BCSA tools, only two [55, 56] support both ARM and MIPS architectures. They did not, however, release the complete source code or datasets. As a result, we were unable to directly compare our system to theirs. Instead, we used the same experimental method as in these two studies, using our dataset and ground truth information that we created in §6.3.2. More specifically, we queried each sample vulnerable function against all 52M functions in the 1,124 firmware images (Table 6.1). This resulted in 48 similarity scores for each of the 52M functions. Then, we calculated the top-k result by averaging the similarity scores for each function. Note that this experimental methodology is different from our previous experiments described in §6.3.1 and §6.3.2. We selected CVE-2015-1791 as our target in this experiment, which has also been used in previous research [55, 56].

Table 6.6 summarizes the top-k results for the averaged similarity score for all 52M firmware functions. Though our dataset is distinct from those used in previous studies, our system presented promising results. TIKNIB

Table 6.6: Top-k results of analyzing all 52M functions in Linux-based firmware images using TIKNIB.

Top-k	Existing Approaches		TIKNIB (Our Approach)		
	Gemini†	VulSeeker†	O0-O3 Features‡	O2-O3 Features‡	w/ Heuristic Features
1	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)
5	2 (40%)	3 (60%)	5 (100%)	5 (100%)	5 (100%)
10	4 (40%)	6 (60%)	9 (90%)	10 (100%)	10 (100%)
50	36 (72%)	41 (82%)	19 (38%)	46 (92%)	50 (100%)
100	75 (75%)	83 (83%)	50 (50%)	82 (82%)	100 (100%)

† Only two of 43 BCSA papers that we studied in Chapter 5 support both ARM and MIPS architectures. We were not able to compare the results directly because they released neither their datasets nor complete source code. For the purpose of comparison, we present the results stated in their papers.

‡ *O2-O3 Features* represent the features extracted from the dataset compiled with `O2` and `O3`. As these features outperforms *O0-O3 Features* extracted from the dataset compiled with `O0`–`O3`, it is highly likely that the tested firmware images were compiled with O2 or O3.

equipped with heuristic features identified vulnerable functions perfectly in its top-100 result. Notably, TIKNIB with presemantic features learned from the dataset containing only `O2` and `O3` binaries performs better than TIKNIB with those learned from the dataset containing all `O0`–`O3` binaries. This result indicates that a large number of firmware images were most likely compiled with `O2` or `O3`. Moreover, our system achieved a result comparable to that of state-of-the-art tools. Note that *our goal is not to assert that our approach is superior to these tools, and this result demonstrates that properly engineered features based on heuristic knowledge (i.e., heuristic features) are effective in IoT vulnerability analysis.*

6.4 Discussion

In this study, we leveraged BCSA for identifying previously known vulnerabilities in IoT devices. Therefore, this study inherits the discussion points of BCSA described in the previous chapter (Chapter 5). Although TIKNIB equipped with the heuristic features demonstrated promising results, additional effective features may exist. For example, as discussed in §5.6, type information can be used as a feature. We encourage further research in this area to consider recovering debugging information from binary code, such as function types [246, 247, 243, 244, 248, 249]. Additionally, inter-procedural analysis can be used to address compiler optimization issues; function inlining can severely affect the features that we used. One can also develop effective features for static binaries.

Even if BCSA successfully identifies functions from IoT devices that are associated with a targeted vulnerability, additional analysis may be required to determine whether the identified functions are vulnerable. We manually verified the identified functions in this study; however, one may leverage other promising techniques in various analysis scopes. For instance, one can run symbolic execution at the function level to verify the vulnerability. Additionally, one can emulate the target binary (*i.e.*, at the binary level) and perform a dynamic vulnerability check using PoC scripts. On the other hand, an entire firmware image may need to be analyzed, as vulnerabilities can spread across multiple binaries. We reserve such promising research topics in the IoT vulnerability analysis for future work.

6.5 Conclusion

Discovering previously known vulnerabilities in another device is a prominent issue in the IoT ecosystem. To counter this issue, several studies have focused on developing novel approaches that combine BCSA and sophisticated machine learning techniques. However, few studies have investigated IoT devices in real-world

scenarios, and none have examined vulnerabilities in custom binaries, such as CGI binaries. Moreover, there is no readily available tool for conducting practical analysis. To address this, we established the first large-scale ground-truth dataset for IoT vulnerability analysis, named FIRMKIT, which contains 1,142 firmware images and 323 vulnerabilities. We discovered that BCSA is effective in practice for scalable IoT vulnerability analysis, and that simple heuristic features can significantly improve the performance of BCSA. We believe that the findings from this study will contribute to further research in this field. To encourage further research, we make our datasets and source code publicly available.

Furthermore, from this study, we learned that many IoT vendors frequently manage their devices' and source code's versions incorrectly, and even copy and paste buggy code without validating it. Our empirical analysis revealed that many vendors lack an internal policy or process for developing more secure software. As a result, numerous IoT devices share similar/identical vulnerabilities and are continuously exploited by them. One possible explanation for such phenomena is that IoT vendors are not financially motivated to secure their software. A business's typical objective is to profit from the sale of its products. As a result, after weighing the risk and probability of a security breach, IoT vendors may conclude that enhancing the security of their devices is not economically viable. Meanwhile, contemporary software communities are engaged in lively debates about security economics. For instance, *DevSecOps*, which stands for development, security, and operations, is a methodology for integrating security as a shared responsibility throughout the entire information technology lifecycle. We believe that if IoT vendors take this approach, they will contribute to the IoT ecosystem's security. In this regard, we invite further research into security economics to determine how to incentivize IoT vendors to incorporate security into their software development lifecycle.

Chapter 7. Thesis Conclusion

Security analysis techniques have made significant advancements in order to secure the IoT ecosystem. However, many recent techniques are not yet scalable owing to the obscurity and diversity issues underlying the IoT ecosystem. This immaturity may be a result of a prominent trend in existing academic research; existing studies focuses primarily on novelty and freshness while ignoring hands-on analysis and engineering efforts (*i.e.*, heuristic-based approaches).

In this thesis, we argued that developing and systematizing heuristics based on empirical analysis is both inevitable and necessary to achieve scalable security analysis of IoT devices. We examined the limitations of recent approaches to IoT vulnerability analysis and discovered that heuristics can effectively address these limitations. Heuristics that we developed 1) significantly increased the rate of firmware emulation, 2) successfully analyzed firmware structures, and 3) successfully identified numerous known vulnerabilities. Additionally, we demonstrated that the developed heuristics can be transferred to a variety of device types and versions, as many IoT devices share common codebases. By systematizing the developed heuristics, we successfully tested a total of 1,256 vulnerabilities from 1,143 firmware images in wireless routers, IP cameras, and smartphone baseband devices, thereby corroborating our hypothesis: *"While heuristics may appear trivial and not technically novel, developing and systematizing 'dirty' heuristics is critical, and it is the last-mile effort required to enable large-scale vulnerability analysis on the IoT ecosystem."*. We conclude by encouraging further research into developing an integrated knowledge base for IoT analysis based on empirical investigations, believing that doing so will eventually secure the IoT ecosystem.

Bibliography

[1] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: an analysis of iot devices on home networks," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[2] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[3] Lily Hay Newman, "What We Know About Friday's Massive East Coast Internet Outage," https://www.wired.com/2016/10/internet-outage-ddos-dns-dyn/, 2016, [Online; accessed 28-February-2021].

[4] ——, "GitHub Survived the Biggest DDoS Attack Ever Recorded," https://www.wired.com/story/github-ddos-memcached/, 2018, [Online; accessed 28-February-2021].

[5] S. Khandelwal, "Multiple Backdoors found in D-Link DWR-932 B LTE Router," http://thehackernews.com/2016/09/hacking-d-link-wireless-router.html?m=1, Sep. 2016.

[6] D. Makrushin, "Backdoors in D-Link's backyard," May. 2018, https://securelist.com/backdoors-in-d-links-backyard/85530.

[7] B. Krebs, "Source Code for IoT Botnet 'Mirai' Released," https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released, Oct. 2016.

[8] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[9] S. Khandelwal, "Satori IoT Botnet Exploits Zero-Day to Zombify Huawei Routers," Dec. 2017, https://thehackernews.com/2017/12/satori-mirai-iot-botnet.html.

[10] D. Maciejak, "Yet Another Crypto Mining Botnet?" May. 2018, https://www.fortinet.com/blog/threat-research/yet-another-crypto-mining-botnet.html.

[11] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013, pp. 463–478.

[12] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[13] H. Y. Xunchao Hu, Yaowen Zheng, "An extensible dynamic analysis framework for iot devices," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2018.

[14] N. Group *et al.*, "A linux system call fuzzer using TriforceAFL," https://github.com/nccgroup/TriforceAFL, 2017.

[15] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019, pp. 1099–1114.

[16] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–Jun. 2016.

[17] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[18] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *Proceedings of the 22th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Beijing, China, Sep. 2019.

[19] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[20] A. Vetterl and R. Clayton, "Honware: A virtual honeypot framework for capturing CPE and IoT zero days," in *2019 APWG Symposium on Electronic Crime Research (eCrime)*. IEEE, 2019, pp. 1–13.

[21] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual Computer Security Applications Conference (ACSAC)*, Virtual, Dec. 2020.

[22] D. Kim, E. Kim, M. Kim, Y. Jang, and Y. Kim, "Enabling the large-scale emulation of internet of things firmware with heuristic workarounds," *IEEE Security & Privacy*, 2021.

[23] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "BaseSpec: Comparative analysis of baseband software and cellular specifications for l3 protocols," Feb. 2021.

[24] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," 2020.

[25] R. Mitchell, *Web Scraping with Python: Collecting More Data from the Modern Web*. "O'Reilly Media, Inc.", 2018.

[26] C. Heffner, "Firmware Analysis Tool," 2010, https://github.com/ReFirmLabs/binwalk.

[27] C. Heffner, J. Collake *et al.*, "Firmware Mod Kit," 2011, https://github.com/rampageX/firmware-mod-kit.

[28] Threat9, "RouterSploit," 2016, https://github.com/threat9/routersploit.

[29] X. Mendez, "wfuzz," https://github.com/xmendez/wfuzz, 2014.

[30] W. Chunlei, L. Li, and L. Qiang, "Automatic fuzz testing of web service vulnerability," in *Proceedings of the International Conference on Information and Communications Technologies (ICT)*, Nanjing, China, May 2014.

[31] R. Dawes, "OWASP WebScarab Project," 2011.

[32] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, vol. 34, 2011.

[33] H. Moore *et al.*, "The Metasploit project," 2009, https://www.metasploit.com.

[34] D. Stuttard, "Burp Suite," 2008, https://portswigger.net/burp.

[35] A. Stasinopoulos, C. Ntantogian, and C. Xenakis, "Commix: Detecting and exploiting command injection flaws," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2015.

[36] T. Laskos, "Arachni," 2010, http://www.arachni-scanner.com.

[37] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, 2005.

[38] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[39] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Kyoto, Japan, Jun. 2014.

[40] M. Kammerstetter, D. Burian, and W. Kastner, "Embedded security testing with peripheral device caching and runtime program state approximation," in *Proceedings of the 10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[41] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[42] M. Muench, A. Francillon, and D. Balzarotti, "Avatar2: A multi-target orchestration platform," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2018.

[43] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[44] M. Zalewski, "American fuzzy lop (AFL)," http://lcamtuf.coredump.cx/afl, 2017.

[45] A. Cui, "Embedded device firmware vulnerability hunting using frak," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2012.

[46] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[47] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015, pp. 611–626.

[48] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[49] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the 40th IEEE European Symposium on Security and Privacy (EUROSP)*, Paris, France, Apr. 2017.

[50] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, Sep. 2017, pp. 388–398.

[51] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, Jun. 2017, pp. 201–212.

[52] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[53] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the Annual Computer Security Applications Conference*, 2014, pp. 406–415.

[54] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

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

[56] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sep. 2018, pp. 896–899.

[57] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proceedings of the 23th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019, pp. 392–404.

[58] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-os binary search," in *Proceedings of the 24nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle WA, Nov. 2016, pp. 678–689.

[59] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017, pp. 346–359.

[60] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Saclay, France, Jun. 2018, pp. 114–138.

[61] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Transactions on Software Engineering*, 2018.

[62] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with DNN," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sep. 2018, pp. 667–678.

[63] S. H. Ding, B. C. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[64] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Gothenburg, Sweden, Jun. 2019, pp. 309–329.

[65] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010, pp. 61–76.

[66] 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 (CCS)*, Chicago, Illinois, Oct. 2011, pp. 309–320.

[67] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, Nov. 2014, pp. 389–400.

[68] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Naples, Italy, 2014, pp. 66–77.

[69] X. Meng, B. P. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, Oslo, Norway, Sep. 2017, pp. 286–304.

[70] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 709–724.

[71] 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 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016, pp. 480–491.

[72] C. Mulliner and C. Miller, "Fuzzing the Phone in your Phone," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2009.

[73] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale." in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[74] F. Van Den Broek, B. Hond, and A. C. Torres, "Security Testing of GSM Implementations," in *Proceedings of the 6th International Symposium on Engineering Secure Software and Systems (ESSoS)*, Munich, Germany, 2014, pp. 179–195.

[75] R.-P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, Bellevue, WA, Aug. 2012, pp. 12–21.

[76] N. Golde and D. Komaromy, "Breaking Band: reverse engineering and exploiting the shannon baseband," *REcon*, 2016. [Online]. Available: https://comsecuris.com/slides/recon2016-breaking_band.pdf

[77] A. Cama, "A walk with Shannon," in *OPCDE*, 2018.

[78] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," in *Proceedings of the 13th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Virtual, Jul. 2020.

[79] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[80] K. Fang and G. Yan, "Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning," in *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, Barcelona, Spain, Sep. 2018.

[81] D. Rupprecht, K. Jansen, and C. Pöpper, "Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[82] M. T. Raza, F. M. Anwar, and S. Lu, "Exposing LTE Security Weaknesses at Protocol Inter-Layer, and Inter-Radio Interactions," in *Proceedings of the 13th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Niagara Falls, Canada, Oct. 2017, pp. 312–338.

[83] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert, "New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Miami, FL, May 2019.

[84] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[85] "CVE-2014-3936," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3936, 2014.

[86] R. Bodenheim, J. Butts, S. Dunlap, and B. Mullins, "Evaluation of the ability of the shodan search engine to identify internet-facing industrial control devices," *International Journal of Critical Infrastructure Protection*, vol. 7, no. 2, pp. 114–123, 2014.

[87] Shodan, "D-Link Internet Report," https://dlink-report.shodan.io/, 2016.

[88] M. Wilson, "Premium Wireless Routers Market Size, Share, Statistics, Trends, Types, Applications, Analysis and Forecast— Global Industry Research and Forecast 2019-2024," https://marketersmedia.com/premium-wireless-routers-market-size-share-statistics-trends-types-applications-analysis-and-forecast-global-industry-researc 520294, 2019.

[89] F. Fainelli, "The OpenWrt embedded development framework," in *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

[90] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *Proceedings of the 2013 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[91] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[92] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[93] "Selenium," 2004, https://www.seleniumhq.org.

[94] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015, pp. 725–741.

[95] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[96] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[97] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018, pp. 745–761.

[98] L. Rist, J. Vestergaard, D. Haslinger, A. Pasquale, and J. Smith, "Conpot ics/scada honeypot," http://conpot.org, 2013.

[99] S. Litchfield, D. Formby, J. Rogers, S. Meliopoulos, and R. Beyah, "Rethinking the honeypot for cyber-physical systems," *IEEE Internet Computing*, vol. 20, no. 5, pp. 9–17, 2016.

[100] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: analysing the rise of IoT compromises," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, Washington, DC, Aug. 2015.

[101] S. Hex-Rays, "IDA Pro," https://www.hex-rays.com/products/ida.

[102] 3GPP, "TS 24.007; Mobile radio interface signalling layer 3; General aspects," 2018.

[103] S. Chin, "Top-tier smartphone makers going to in-house processors: report," https://www.fierceelectronics.com/electronics/top-tier-smartphone-makers-going-to-house-processors-report.

[104] R.-P. Weinmann, "Baseband exploitation in 2013: Hexagon challenges," in *PACSEC 2013*, Tokyo, Japan, 2013.

[105] G. Delugré, "Reverse engineering a Qualcomm baseband," in *28th Chaos Communication Congress*, Berlin, Germany, dec 2011.

[106] W. Hengeveld, "IDA processor module for the hexagon (QDSP6v55) processor," https://github.com/gsmk/hexagon, 2013.

[107] D. Berard and V. Fargues, "How to design a baseband debugger," in *Information and Communication Technology Security Symposium (SSTIC)*, Rennes, France, jun 2020.

[108] "URL Anonymized due to vendor's request."

[109] S. Hex-Rays, "IDA FLIRT Technology: In-Depth," https://www.hex-rays.com/products/ida/tech/flirt/in_depth/.

[110] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Virtual, Jul. 2020.

[111] A. Costin, A. Zarras, and A. Francillon, "Towards Automated Classification of Firmware Images and Identification of Embedded Devices," in *Proceedings of the 32nd IFIP International Information Security Conference (IFIP SEC)*, Rome, Italy, 2017, pp. 233–247.

[112] 3GPP, "TS 24.008; Mobile radio interface Layer 3 specification; Core network protocols; Stage 3," 2019.

[113] ——, "TS 24.011; Point-to-Point (PP) Short Message Service (SMS) support on mobile radio interface," 2019.

[114] ——, "TS 24.301; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3," 2019.

[115] ——, "TS 44.018; Mobile radio interface layer 3 specification; GSM/EDGE Radio Resource Control (RRC) protocol," 2019.

[116] ——, "3GPP Releases," https://www.3gpp.org/specifications/releases.

[117] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[118] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[119] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A Formal Analysis of 5G Authentication," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[120] X. Hu, C. Liu, S. Liu, W. You, Y. Li, and Y. Zhao, "A Systematic Analysis Method for 5G Non-Access Stratum Signalling Security," *IEEE Access*, vol. 7, pp. 125 424–125 441, 2019.

[121] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New Privacy Issues in Mobile Telephony: Fix and Verification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[122] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*, Cleveland, OH, Mar. 2014, pp. 323–332.

[123] 3GPP, "TS 36.523-3: Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Packet Core (EPC); User Equipment (UE) conformance specification; Part 3: Test suites," 2019.

[124] M. T. Raza and S. Lu, "A systematic way to lte testing," in *Proceedings of the 25th Annual international conference on Mobile computing and networking (MobiCom)*, Los Cabos, Mexico, Oct. 2019.

[125] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[126] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "srsLTE: An Open-Source Platform for LTE Evolution and Experimentation," in *Proceedings of the 10th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (WiNTECH)*, New York, NY, Oct. 2016.

[127] D. A. Burgess and H. S. Samra, "The OpenBTS Project," *Open Source Cellular Infrastructure*, 2008. [Online]. Available: http://openBTS.org

[128] N. Nikaein, R. Knopp, F. Kaltenberger, L. Gauthier, C. Bonnet, D. Nussbaum, and R. Ghaddab, "OpenAirInterface: An Open LTE Network in a PC," in *Proceedings of the 20th Annual international conference on Mobile computing and networking (MobiCom)*, Maui, Hawaii, Sep. 2014.

[129] B. Wojtowicz, "OpenLTE," *An open source 3GPP LTE implementation*, 2016. [Online]. Available: http://openlte.sourceforge.net

[130] U. Ettus, "B210." [Online]. Available: https://www.ettus.com/all-products/ub210-kit/

[131] L. Nuand, "bladeRF." [Online]. Available: https://www.nuand.com/bladerf-2-0-micro/

[132] H. Lin, "LTE REDIRECTION: Forcing Targeted LTE Cellphone into Unsafe Network," in *Hack In The Box Security Conference (HITBSec-Conf)*, 2016.

[133] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert, "Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[134] G.-H. Tu, Y. Li, C. Peng, C.-Y. Li, H. Wang, and S. Lu, "Control-Plane Protocol Interactions in Cellular Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 223–234, 2014.

[135] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, "Breaking and Fixing VoLTE: Exploiting Hidden Data Channels and Mis-implementations," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[136] S. Holtmanns, S. P. Rao, and I. Oliver, "User Location Tracking Attacks for LTE Networks Using the Interworking Functionality," in *Proceedings of the 15th Annual IFIP Networking Conference*, Vienna, Austria, May 2016.

[137] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper, "LTE Security Disabled: Misconfiguration in Commercial Networks," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Miami, FL, May 2019.

[138] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking LTE on Layer Two," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[139] M. Lichtman, R. P. Jover, M. Labib, R. Rao, V. Marojevic, and J. H. Reed, "LTE/LTE-A Jamming, Spoofing, and Sniffing: Threat Assessment and Mitigation," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 54–61, 2016.

[140] H. Yang, S. Bae, M. Son, H. Kim, S. M. Kim, and Y. Kim, "Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[141] Marc Heuse, Heiko Eißfeld, Andrea Fioraldi, and Dominik Maier, "AFLplusplus (AFL++)," https://github.com/vanhauser-thc/AFLplusplus, 2020.

[142] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009, pp. 243–253.

[143] "gpl-violations.org project prevails in court case on gpl violation by d-link." [Online]. Available: https://web.archive.org/web/20141007073104/http://gpl-violations.org/news/20060922-dlink-judgement_frankfurt.html

[144] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[145] N. Marastoni, R. Giacobazzi, and M. Dalla Preda, "A deep learning approach to program similarity," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, 2018, pp. 26–35.

[146] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2019.

[147] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[148] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, "Hybrid firmware analysis for known mobile and iot security vulnerabilities," in *Proceedings of the 50th International Conference on Dependable Systems and Networks (DSN)*, Valencia, Spain, Jul. 2020, pp. 373–384.

[149] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2019.

[150] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014, pp. 303–317.

[151] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017, pp. 319–330.

[152] S. H. Ding, B. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, Aug. 2016, pp. 461–470.

[153] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, Mar. 2016, pp. 57–67.

[154] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017, pp. 155–166.

[155] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 88–98.

[156] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017, pp. 353–364.

[157] 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 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013, pp. 353–368.

[158] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[159] R. Real and J. M. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic biology*, vol. 45, no. 3, pp. 380–385, 1996.

[160] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, no. 8, pp. 689–694, 1997.

[161] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in neural information processing systems*, 1994, pp. 737–744.

[162] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[163] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*, Birmingham, UK, 2008, pp. 238–255.

[164] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," in *Information and Communication Technology Security Symposium (SSTIC)*, Rennes, France, Jun. 2005.

[165] H. Flake, "Structural comparison of executable objects," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Dortmund, Germany, Jul. 2004, pp. 161–174.

[166] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, Rome, Italy, Jan. 2013, p. 4.

[167] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with inter-procedural control flow," in *Proceedings of the 2012 International Conference on Information Security and Cryptology (ICISC)*, 2012, pp. 92–109.

[168] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA)*, Boca Raton, FL, Jul. 2012, pp. 386–391.

[169] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic'juice'," in *Proceedings of the 2nd Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, Rome, Italy, Jan. 2013, p. 5.

[170] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.

[171] S. Alrabaee, L. Wang, and M. Debbabi, "BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs)," *Digital Investigation*, vol. 18, pp. S11–S22, 2016.

[172] T. Kim, Y. R. Lee, B. Kang, and E. G. Im, "Binary executable file similarity calculation using function matching," *The Journal of Supercomputing*, vol. 75, pp. 607–622, 2019.

[173] H. Guo, S. Huang, C. Huang, M. Zhang, Z. Pan, F. Shi, H. Huang, D. Hu, and X. Wang, "A lightweight cross-version binary code similarity detection based on similarity and correlation coefficient features," *IEEE Access*, vol. 8, pp. 120 501–120 512, 2020.

[174] "Bindiff." [Online]. Available: https://www.zynamics.com/bindiff.html

[175] "Diaphora, a Free and Open Source program diffing tool." [Online]. Available: http://diaphora.re/

[176] J. W. Oh, "Darungrim: a patch analysis and binary diffing too," 2015.

[177] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014, pp. 349–360.

[178] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proceedings of the 8th International Conference on Software Security and Reliability (SERE)*, 2014, pp. 78–87.

[179] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, Jun. 2016, pp. 266–280.

[180] N. Lageman, E. D. Kilmer, R. J. Walls, and P. D. McDaniel, "BinDNN: Resilient function matching using deep learning," in *Proceedings of the 12th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Guangzhou, China, Oct. 2016, pp. 517–537.

[181] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, and A. Hanna, "Binsign: fingerprinting binary functions to support automated analysis of code executables," in *Proceedings of the 32nd IFIP International Information Security Conference (IFIP SEC)*, Rome, Italy, 2017, pp. 341–355.

[182] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017, pp. 79–94.

[183] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017, pp. 253–270.

[184] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017, pp. 342–352.

[185] C. Karamitas and A. Kehagias, "Efficient features for function matching between binary executables," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Campobasso, Italy, Mar. 2018, pp. 335–345.

[186] B. Yuan, J. Wang, Z. Fang, and L. Qi, "A new software birthmark based on weight sequences of dynamic control flow graph for plagiarism detection," *The Computer Journal*, 2018.

[187] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu, "Binmatch: A semantics-based hybrid approach on binary code clone analysis," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid, Spain, Sep. 2018.

[188] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 42–47.

[189] M. Luo, C. Yang, X. Gong, and L. Yu, "Funcnet: A euclidean embedding approach for lightweight cross-platform binary recognition," in *Proceedings of the 15th International Conference on Security and Privacy in Communication Networks (SecureComm)*, Orlando, FL, Oct. 2019, pp. 517–537.

[190] J. Jiang, G. Li, M. Yu, G. Li, C. Liu, Z. Lv, B. Lv, and W. Huang, "Similarity of binaries across optimization levels and obfuscation," in *Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS)*, Guildford, United Kingdom, Sep. 2020, pp. 295–315.

[191] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and robust binary library function identification using function shape," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, Jul. 2017, pp. 301–324.

[192] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May–Jun. 2014, pp. 175–186.

[193] 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 (CCS)*, Chicago, IL, Nov. 2009, pp. 611–620.

[194] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[195] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire os distributions," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012, pp. 48–62.

[196] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "SplitScreen: Enabling efficient, distributed malware detection," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2010, pp. 377–390.

[197] 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 (MSR)*, 2013, pp. 329–338.

[198] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013, pp. 305–316.

[199] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.

[200] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, pp. 1157–1177, 2017.

[201] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 1996, pp. 120–128.

[202] Z. Tian, Q. Wang, C. Gao, L. Chen, and D. Wu, "Plagiarism detection of multi-threaded programs via siamese neural networks," *IEEE Access*, vol. 8, pp. 160 802–160 814, 2020.

[203] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[204] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proceedings of the 14th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Menlo Park, CA, Sep. 2011, pp. 41–60.

[205] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, Jun. 1988, pp. 35–46.

[206] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, 1987.

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

[208] J. Yang, Y.-G. Jiang, A. G. Hauptmann, and C.-W. Ngo, "Evaluating bag-of-visual-words representations in scene classification," in *Proceedings of the 9th ACM SIGMM International Workshop on Multimedia Information Retrieval (MIR)*, Augsburg, Germany, 2007, pp. 197–206.

[209] R. Arandjelovic and A. Zisserman, "All about vlad," in *Proceedings of the 26 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Portland, OR, Jun. 2013, pp. 1578–1585.

[210] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, New York, NY, Jun. 2016, pp. 2702–2711.

[211] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[212] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[213] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[214] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning (ICML)*, Beijing, China, Jun. 2014, pp. 1188–1196.

[215] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2019.

[216] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, Princeton, NJ, Jul. 2008, pp. 423–427.

[217] SecurityTeam, "Pie," 2016. [Online]. Available: https://wiki.ubuntu.com/SecurityTeam/PIE

[218] "GNU packages." [Online]. Available: https://ftp.gnu.org/gnu/

[219] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, 2015, pp. 3–9.

[220] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de) obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006, pp. 140–144.

[221] "VMProtect." [Online]. Available: http://vmpsoft.com

[222] "Stunnix C/C++ Obfuscator." [Online]. Available: http://stunnix.com/prod/cxxo/

[223] "Semantic Designs: Source Code Obfuscators." [Online]. Available: http://www.semdesigns.com/Products/Obfuscators/

[224] C. Collberg, "The tigress c diversifier/obfuscator," *Retrieved August*, vol. 14, p. 2015, 2015.

[225] "Crosstool-NG." [Online]. Available: https://github.com/crosstool-ng/crosstool-ng

[226] D. MacKenzie, B. Elliston, and A. Demaille, "Autoconf — creating automatic configuration scripts," 1996.

[227] O. Tange, "GNU parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[228] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the Python in Science Conference*, 2008, pp. 11–15.

[229] Intel Corporation, "Intel® 64 and ia-32 architectures software developer's manual," https://software.intel.com/en-us/articles/intel-sdm.

[230] D. Seal, *ARM Architecture Reference Manual*. Pearson Education, 2001.

[231] MIPS Technologies, Inc., "Mips32 architecture for programmers volume ii: The mips32 instruction set," 2001.

[232] Capstone, "The ultimate disassembler." [Online]. Available: https://www.capstone-engine.org/

[233] Wikipedia, "Relative change and difference — wikipedia, the free encyclopedia," 2018, [Online; accessed ¡today¿]. [Online]. Available: "https://en.wikipedia.org/w/index.php?title=Relative_change_and_difference&oldid=872867886"

[234] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[235] R. Caruana and D. Freitag, "Greedy attribute selection," in *Proceedings of the Eleventh International Conference on Machine Learning (ICML)*, New Brunswick, NJ, Jul. 1994, pp. 28–36.

[236] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[237] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[238] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[239] "Using the GNU compiler collection (GCC): Optimize options." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[240] "Clang - the clang c, c++, and objective-c compiler." [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html

[241] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, pp. 3–19, 2009.

[242] "Themida: Advanced windows software protection system." [Online]. Available: https://www.oreans.com/themida.php

[243] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017, pp. 99–116.

[244] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016, pp. 934–953.

[245] D. Hiebert, "Exuberant Ctags," 1999.

[246] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[247] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, Jun. 2013, pp. 51–60.

[248] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018, pp. 1667–1680.

[249] F. Artuso, G. A. Di Luna, L. Massarelli, and L. Querzoni, "In nomine function: Naming functions in stripped binaries with neural networks," *arXiv*, pp. arXiv–1912, 2019.

[250] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, Jul. 2011, pp. 100–110.

[251] M. C. Tol, K. Yurtseven, B. Gulmezoglu, and B. Sunar, "Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings," *arXiv preprint arXiv:2006.14147*, 2020.

[252] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Minneapolis, MN, Jul. 2012, pp. 122–132.

[253] D. Babić, D. Reynaud, and D. Song, "Malware analysis with tree automata inference," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, Jul. 2011, pp. 116–131.

[254] Y. Xiao, S. Cao, Z. Cao, F. Wang, F. Lin, J. Wu, and H. Bi, "Matching similar functions in different versions of a malware," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 252–259.

[255] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *Proceedings of the 30th IFIP International Information Security Conference (IFIP SEC)*, Hamburg, Germany, 2015, pp. 416–430.

[256] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Fossil: a resilient and efficient system for identifying foss functions in malware binaries," *ACM Transactions on Privacy and Security*, vol. 21, pp. 1–34, 2018.

[257] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012, pp. 169–182.

[258] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 921–937.

[259] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, Canada, Mar. 2015, pp. 261–270.

[260] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, "A neural network-based approach for cryptographic function detection in malware," *IEEE Access*, vol. 8, pp. 23 506–23 521, 2020.

[261] S. Alrabaee, M. Debbabi, and L. Wang, "Cpa: Accurate cross-platform binary authorship characterization using lda," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3051–3066, 2020.

[262] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, May 2017, pp. 462–472.

[263] Y. Hu, Y. Zhang, and D. Gu, "Automatically patching vulnerabilities of binary programs via code transfer from correct versions," *IEEE Access*, vol. 7, pp. 28 170–28 184, 2019.

[264] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual, Nov. 2020.

[265] I. U. Haq and J. Caballero, "A survey of binary code similarity," *arXiv preprint arXiv:1909.11424*, 2019.

[266] 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*, vol. 28, pp. 654–670, 2002.

[267] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD/PODS Conference*, San Diego, CA, Jun. 2003, pp. 76–85.

[268] 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 *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004, pp. 289–302.

[269] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007, pp. 96–105.

[270] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.

[271] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SymDiff: A language-agnostic semantic diff tool for imperative programs," in *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, Berkeley, CA, Jul. 2012, pp. 712–717.

[272] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017, pp. 595–614.

[273] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May 2016, pp. 297–308.

[274] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2016, pp. 201–213.

[275] D. Miyani, Z. Huang, and D. Lie, "BinPro: A tool for binary source code provenance," *arXiv preprint arXiv:1711.00830*, 2017.

[276] A. Rahimian, P. Charland, S. Preda, and M. Debbabi, "RESource: A framework for online matching of assembly with open source code," in *Proceedings of the 5th International Symposium on Foundations and Practice of Security (FPS)*, Montreal, Canada, 2012, pp. 211–226.

[277] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 63–72.

[278] *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[279] *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[280] *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[281] *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[282] *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[283] *Proceedings of the 2nd Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, Rome, Italy, Jan. 2013.

[284] *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[285] *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Miami, FL, May 2019.

[286] *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[287] *Proceedings of the 32nd IFIP International Information Security Conference (IFIP SEC)*, Rome, Italy, 2017.

[288] *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[289] *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.

[290] *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sep. 2018.

[291] *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[292] *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2019.

[293] *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017.

[294] *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[295] *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[296] *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

Acknowledgment

First and foremost, I want to express my sincere gratitude to my advisor, Prof. Yongdae Kim, for his essential counsel and assistance. I initially joined the system security laboratory (*i.e.*, SysSec) during my final year as a computer science student at KAIST (Jan. 2013). During my undergraduate studies, I was just a hacker with no knowledge of research. Over the long journey of nine years, he taught me how to be a researcher while maintaining my hacker identity. Despite the fact that academia prefers scientists, he recommended me to continue working as a hacker while doing research. As a result, I was able to finish my Ph.D. with this insightful thesis. *"Think like an adversary,"* he said, became one of my favorite phrases. I will never forget those remarks, which I will apply to any profession, not only security.

Prof. Sang Kil Cha and Prof. Sooel Son have taught me how to be a great researcher, while I still have a long way to go. I would also want to thank Prof. Yeongjin Jang and Prof. Insu Yun for being my research and life mentors. They taught me not just how to write papers, but also how to live as a doctoral student. I would also want to thank Prof. Shin Yoo for giving me with a lot of inspiration and constructive comments on my thesis. In addition, I appreciate Heejung Kim and Sojin Lee for their administrative assistance and invaluable life advise. I would also want to thank Prof. Sukyoung Ryu for providing me with several opportunities to enjoy my Ph.D. life.

As a fantastic collaborator and friend, Eunsoo (a.k.a., hahah) Kim tolerated me. I consider myself very lucky to have had the chance to work closely with him during my whole academic career, beginning in 2010. I would also want to thank Mingeun Kim for his tireless efforts even after he graduated. We proceeded to discuss about FirmAE and eventually turned it to be a paper, which served as a springboard for my thesis.

Since 2013, I have also had the good fortune to work with a number of smart and kind researchers. Yunmok Son, Hyunwoo Choi, Byeongdo Hong, Hyunwook Hong, Hocheol Shin, Juhwan Noh, Hongil Kim, Yujin Kwon, and Sangwook Bae (selected) are outstanding researchers who endured a difficult path to get a Ph.D. Over their shoulders, I learnt how to be a Ph.D.

Additionally, I could not have endured the painful binary analysis without the help of Youjin Lee, JeongOh Kyea, Suryeon Kim, Sangsup Lee, Hyunki Kim, Myojoon Kil, Yohan Choi, and Hyunsik Jeong, who have served the software analysis team (a.k.a., reverse-engineering labor union) since 2014. In addition, I extend my best wishes to the new team leader, Changhun Song, as well as teammates JoonHa Jang and YongHwa Lee.

I am also thankful to my amazing colleagues both within and outside of SysSec: Dohyun Kim, Jinseob Jeong, CheolJun park, Jiho Lee, Mincheol Son, ManGi Cho, Jaehoon Kim, Taekkyung Oh, Hoang Dinh Tuan, Yeongbin Hwang, Sanggu Han, Hojoon Yang, Eunkyu Lee, Seokbin Yun, Soobin Lee, Geunwoo Lim, Kyuchan Shim, Minjung Kim, Hyunjin Choo, Suwan Park, Jaeyeong Choi, Kibum Choi, Shinjo Park, Youngseok Park, Sanrok Lee, Jangjoon Lee, Minhee Kwon, Sungjae Hwang, Daejun Kim, and Minkyo Seo.

I could not have survived without happiness as a Ph.D. student or as a human being. KAIST GoN, KaisHack, HUG KAIST, Yogi Buddies, Military Colleagues, B.Fam, B.Chimpanzees, and others I might have overlooked all assisted me in enjoying my Ph.D. life and maintaining my mental health. I would also want to express my special thanks to Chul Min Kim, Hyungseok Han (a.k.a., DaramG), Donghwan Kwon, Dane Baek, Simon Park, Jin Hyoung Kim, and Minsung Kim for being terrific buddies, collaborators, and life hacks.

Finally, I would want to express my profound appreciation to Jungin Lee and my parents, Eunok Jeon and Jaelak Kim, for their constant support, patience, persistent encouragement, and love.