Towards Laying the Foundation of Firmware Analysis

Muhui JIANG

PhD

The Hong Kong Polytechnic University

2022

The Hong Kong Polytechnic University
Department of Computing

Towards Laying the Foundation of Firmware Analysis

Muhui JIANG

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

July 2021

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____ (Signed)

Muhui JIANG (Name of student)

A dedication to my family.

Abstract

Embedded devices are becoming ubiquitous. Meanwhile, there is a pressing need to perform security assessments for the software (i.e., firmware) of these devices. Static analysis and dynamic analysis are widely used to conduct firmware analysis. This thesis aims to lay the foundation of firmware analysis. Specifically, this thesis explores the limitations and implementation errors of the both static and dynamic firmware analysis tools, makes enhancements to the stability and reliability of these tools, and proposes the new technique and analysis framework to increase the capability and scalability of firmware analysis tools.

Due to different types of peripherals, emulating the firmware of the embedded devices in scale, which supports the dynamic analysis, is challenging. Therefore, static analysis is still widely used. To conduct static analysis, existing works usually leverage off-the-shelf tools to disassemble stripped binaries and (implicitly) assume that reliably disassembling binaries is a solved problem. However, whether this assumption really holds is unknown. We conduct the first comprehensive study on ARM disassembly tools as ARM is becoming the dominant architecture among the embedded devices. Specifically, we build 1,896 ARM binaries (including 248 obfuscated ones) with different compilers, compiling options, and obfuscation methods. We then evaluate them using eight state-of-the-art ARM disassembly tools (including both commercial and noncommercial ones) in different versions on their capabilities to locate instruction boundary, function boundary, and function signature. Instruction and function boundary are two fundamental primitives upon which the other primitives build while function signature is significant for control flow integrity (CFI) techniques. Our work reveals some observations that have not been systematically

summarized and/or confirmed. For instance, we find that the existence of both ARM and Thumb instruction sets, and the reuse of the BL instruction for both function calls and branches bring serious challenges to disassembly tools. Our evaluation sheds light on the limitations of state-of-the-art disassembly tools and points out potential directions for improvement.

Apart from the widely used static analysis, different dynamic analysis frameworks, which are based on the full-system emulator (i.e., QEMU) are proposed for firmware analysis. Emulator is widely used to build dynamic analysis frameworks due to its fine-grained tracing capability, full system monitoring functionality, and scalability of running on different operating systems and architectures. However, whether the emulator is consistent with real devices is unknown. To understand this problem, we aim to automatically locate inconsistent instructions, which behave differently between emulators and real devices. We target ARM architecture, which provides machine readable specification. Based on the specification, we propose a test case generator by designing and implementing the first symbolic execution engine for ARM architecture specification language (ASL). We generate 2,774,649 representative instruction streams and conduct differential testing with these instruction streams between four ARM real devices in different architecture versions (i.e., ARMv5, ARMv6, ARMv7, and ARMv8) and three state-of-the-art emulators (i.e., QEMU, Unicorn, and Angr). We locate a huge number of inconsistent instruction streams (171,857 for QEMU, 223,264 for Unicorn, and 120,169 for Angr). We find undefined implementation in ARM manual and implementation bugs of QEMU are the major causes of inconsistencies. Furthermore, we discover 12 bugs, which influence commonly used instructions (e.g., `BLX`). With the inconsistent instructions, we build three security applications and demonstrate the capability of these instructions on detecting emulators, anti-emulation, and anti-fuzzing.

Though many dynamic firmware analysis frameworks are proposed, booting the Linux kernel (we call this process rehosting the Linux kernel in this thesis.) of embedded device in QEMU is still an unsolved problem. That's because embedded devices usually use different system-on-chips (SoCs) from multiple vendors and only a lim-

ited number of SoCs are currently supported in QEMU. To increase the scalability of the dynamic firmware analysis frameworks, we propose a technique called *peripheral transplantation*. The main idea is to transplant the device drivers of designated peripherals into the Linux kernel. By doing so, it can replace the peripherals in the kernel that are currently unsupported in QEMU with supported ones, thus making the Linux kernel *rehostable*. After that, various applications can be built upon. We implemented this technique inside a prototype system called ECMO and applied it to 815 firmware images, which consist of 20 kernel versions, 37 device models, and 24 vendors. The result shows that ECMO can successfully transplant peripherals for all the 815 Linux kernels. Among them, 710 kernels can be successfully rehosted, i.e., launching a user-space shell (87.1% success rate). The failed cases are mainly because the root file system format (*ramfs*) is not supported by the kernel. We further build three applications, i.e., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, based on the rehosted kernels to demonstrate the usage scenarios of ECMO.

Publications

1. Examiner: Automatically Locating Inconsistent Instructions Between Real Devices and CPU Emulators for ARM, in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)

   Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, Kui Ren

2. ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels, in Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS 2021)

   Muhui Jiang, Lin Ma, Yajin Zhou, Qiang Liu, Cen Zhang, Zhi Wang, Xiapu Luo, Lei Wu, Kui Ren

3. A Comprehensive Study on ARM Disassembly Tools, submitted to IEEE Transactions on Software Engineering (Major Revision)

   Muhui Jiang, Wenlong Zhang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, Kui Ren

4. An Empirical Study on ARM Disassembly Tools, in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)

   Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, Kui Ren

5. A Measurement Study on the (In)security of End-of-Life (EoL) Embedded Devices, submitted to the 31st International World Wide Web Conference (Under Review)

Dingding Wang, **Muhui Jiang**, Rui Chang, Yajin Zhou, Baolei Hou, Xiapu Luo, Lei Wu, Kui Ren

6. FirmGuide: Boosting the Capability of Rehosting Embedded Linux Kernels through Model-Guided Kernel Execution, in Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering 2021 (ASE 2021)

   Qiang Liu, Cen Zhang, Lin Ma, **Muhui Jiang**, Yajin Zhou, Lei Wu, Wenbo Shen, Xiapu Luo, Yang Liu, Kui Ren

7. FirmDep: Towards Rehosting Embedded Web Services, submitted to IEEE Transactions on Dependable and Secure Computing (Under Review)

   Huamao Wu, **Muhui Jiang**, Yajin Zhou, Lei Wu, Jinku Li, Xiapu Luo, Kui Ren

8. Parema: An Unpacking Framework for Demystifying VM-based Android Packers, in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)

   Lei Xue, Yuxiao Yan, Luyi Yan, **Muhui Jiang**, Xiapu Luo, Dinghao Wu, Yajin Zhou

9. PERDICE: Towards Discovering Software Inefficiencies Leading to Cache Misses and Branch Mispredictions, in Proceedings of the 42nd Annual Computer Software and Applications Conference (COMPSAC 2018)

   Ting Chen, Wanyu Huang, **Muhui Jiang**, Xiapu Luo, Lei Xue, Ying Wang, Xiaosong Zhang

10. AutoFlowLeaker: Circumventing Web Censorship through Automation Services, in Proceedings of the 36th IEEE International Symposium on Reliable Distributed Systems (SRDS 2017)

    Shengtuo Hu, Xiaobo Ma, **Muhui Jiang**, Xiapu Luo, and Man Ho Au

11. Characterizing the impacts of application layer DDoS attacks, in Proceedings of the 24th IEEE International Conference on Web Services (ICWS 2017)

    Muhui Jiang, Chenxu Wang, Xiapu Luo, MiuTung Miu, Ting Chen

12. Are HTTP/2 Servers Ready Yet? , in Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)

    Muhui Jiang, Xiapu Luo, TungNgai Miu, Shengtuo Hu, and Weixiong Rao

Acknowledgements

The journey of pursuing Ph.D. degree is not easy. Thanks to the encouragement, support, and invaluable suggestions from the surroundings, without which I cannot complete this thesis.

First, I would like to express my deepest appreciation to my advisor, Dr. Xiapu Luo, for his numerous help during every stage of my Ph.D. study. I really appreciate his considerable expertise, insightful suggestions, and great research taste, which inspire me a lot. It is my great pleasure to be the student of Dr. Luo, and I would like to thank him for supporting me over the years and for giving me the freedom to explore many new areas. I would also like to extend my deepest gratitude to Dr. Yajin Zhou, for his valuable advice, relentless support, and profound belief in my work. It was my great fortune to have the opportunity of working with Dr. Yajin Zhou. I'm deeply indebted to his unwavering guidance, patience, and understanding during my Ph.D. study. Without the help and support from Dr. Luo and Dr. Zhou, this body of work would not have been possible.

Second, I am also grateful to many persons for their support and encouragement. I would like to thank Prof. Yang Liu from the Nanyang Technological University and Dr. Ruoyu Wang from the Arizona State University for the stimulating discussions and insightful comments. I would also like to thank Dr. Zhi Wang from the Florida State University for his constructive suggestions and patient guidance. Thanks also to Dr. Lei Wu for his great suggestions and generous help. I would like to acknowledge the assistance of Nancy Chong. I cannot forget the counselling services you provided when I was depressed. Special thanks to my teammates, Lei Xue, Le Yu, Yutian Tang, Hao Zhou, Xian Zhan, Pengfei Li, Xia Zhou, Dabao Wang, Qiang Liu, Dingding Wang, Wenlong Zhang, Tianyi Xu, Tao Wu, Lin Ma, Yufeng Hu, Siwei Wu, Huamao Wu. I must also thank my friends, Feiteng Mu, Zhaoyan Shen, Mohammed

Aquil Maud Mirza, Xiaoyi Fu, Cen Zhang. Thank you for encouraging me in the days of struggle. Life becomes more colorful because of you guys.

Third, I am grateful to the Hong Kong Polytechnic University, which gives me the financial support and the great research environment. I would like to also thank the Department of Computing for offering me travel grants and the excellent facilities.

Last but certainly not least, I am extremely grateful to my family. I need to thank my parents, who give me the life, the opportunity of education, their selfless love, and all the best they have. In particular, I need to thank my wife, Yitong Chen. Thank you so much for accompanying me and standing by my side in the tough days. I cannot imagine how I can survive the sleepless nights without your encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Firmware Analysis

Embedded devices, which contain computing systems for special proposes, are everywhere and have already been a part of our daily life. For example, the routers for providing network connections, the digital cameras for taking photos, the cell phones we use everyday are all embedded devices.

Embedded device consists of both hardware and software. The hardware varies a lot due to the different functionalities. Meanwhile, the software is developed and customized for controlling the specific device's hardware. We usually call the software of embedded devices as firmware. Specifically, the firmware contains the operating systems, user applications, and the required data.

Once the firmware of embedded devices is compromised, attackers can control the devices to conduct large scale attacks, resulting in serious consequences [44, 31, 17]. For instance, a botnet with hijacked IoT devices could bring down popular websites [53] and disrupt power grids [139]. In this case, the security issue of these devices attract raising attentions [53, 67, 70, 88, 137, 139]. Thus, there is a pressing need to perform security assessments for the software (i.e., firmware) of these devices.

To study the security issues of these devices (i.e., firmware analysis), researchers are actively proposing different mechanisms and frameworks. Existing solutions uti-

Table 1.1: A summary of representative research prototypes and their supported primitives and used disassembly tools.

System	Instruction Boundary	Function Boundary	Control Flow Graph	Call Graph	Disassembly Tools
Firmalice[137]	✓	✓	✓	✓	`angr`
Firmup[71]	✓	✓	✓		`IDA Pro`
Genius[80]	✓	✓	✓		`IDA Pro`
Gemini [149]	✓	✓	✓		`IDA Pro`
RevARM [141]	✓	✓	✓		`IDA Pro`
Bug Search[120]	✓		✓		`IDA Pro`
discovRE[78]	✓	✓	✓	✓	`IDA Pro`
C-FLAT[51]	✓		✓		`Capstone`
Karonte[126]	✓	✓	✓	✓	`angr`

lize two common techniques and they are static analysis and dynamic analysis. However, they have many limitations.

1.2 Motivation

1.2.1 Implicit Assumption of Static Analysis Tools

Static analysis is widely used by the community [67, 71, 80, 137, 149] as it has no false negatives and the scalability of supporting various devices. For instance, it has been used to locate bugs [80] and find authentication bypass vulnerabilities [137] in firmware images.

As shown in Table 1.1, previous systems usually leverage off-the-shelf disassembly tools to extract the required primitives (e.g., instruction boundary) from the target binaries, which are usually stripped without debugging symbols. Researchers assume that reliably disassembling stripped binaries is a solved problem. *However, whether this assumption really holds is unknown.* In fact, due to the specific properties of ARM binaries, state-of-the-art ARM disassemblers have many limitations and may have difficulty recovering the accurate instruction boundaries, function boundaries, or function signatures. These limitations bring obstacles to the functionality of the

static analysis frameworks.

1.2.2 Implicit Assumption of Dynamic Analysis Tools

Dynamic analysis has no false positives and can provide the capability to introspect the runtime state. Based on this capability, different applications, e.g., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, can be built upon.

Researchers proposed different dynamic analysis frameworks to support the firmware analysis, complementing the static analysis technique. Though hardware-based tracing technique exists, they have limitations compared with software emulation. For example, ARM ETM has limited Embedded Trace Buffer (ETB). The size of ETB of the Juno Development Board is 64KB [1] [1]. On the contrary, software emulation is capable of tracing the whole program, provides user-friendly APIs for runtime instrumentation, and can run on multiple operating systems (e.g., Windows and Linux) and host machines in different architectures. Nevertheless, software emulation complements the hardware-based tracing and provides rich functionalities that dynamic analysis systems can build upon. Indeed, many dynamic analysis frameworks [68, 155, 118, 79, 65, 115, 61, 101, 95] are built based on the state-of-the-art CPU emulator, i.e., QEMU, to conduct firmware analysis.

The wide adoption of software emulation usually has an implicit assumption that the execution result of an instruction on the CPU emulator and the real device is identical, thus running a program on the CPU emulator can reflect the result on the real hardware. *However, whether this assumption really holds in reality is unknown.* In fact, the execution result could be different (as shown in our work), either because the CPU emulator has bugs or because it uses a different implementation from the real device. These differences impede the reliability of emulator-based dynamic anal-

[1] The ETB size of different SoCs may be different. However, it's usually limited due to the chip cost and size.

ysis. For instance, the malware can abuse the differences to protect the malicious behaviors from being analyzed in the emulator [125, 91, 89, 107].

1.2.3 Scalability Issue of Dynamic Analysis Tools

Apart from this, the scalability of the dynamic analysis frameworks is an issue. Hundreds of vulnerabilities are discovered every year for the Linux kernel [49]. Once the devices are compromised, attackers can control them to launch further attacks. As such, the security of embedded devices, especially the kernel, deserves a thorough analysis. Running the Linux kernel in QEMU for the desktop system is a solved problem. However, embedded systems usually have different system-on-chips (SoCs) with customized hardware peripherals from multiple vendors. Due to the diverse peripherals in the wild, it is not practical for QEMU to support all kinds of peripherals in any SoC, which is tedious and error-prone. In this case, embedded Linux kernels, which depend on the unsupported peripherals, may stuck, halt, or crash during the rehosting process. Thus, how to rehost the embedded Linux kernels in QEMU is still an open research question.

Previous research [61, 101] provides the capability of rehosting user-space programs by running a customized Linux kernel for a single type of SoC that is already supported in QEMU. This works well because user-space programs mainly depend on standard system calls that are provided by the underlying Linux kernel. Different from user-space programs, the OS kernel (e.g., Linux kernel) interact with hardware peripherals that are usually different in different SoCs.

Some researchers have proposed to use real devices to perform the dynamic analysis [156, 118, 86, 142]. Such solutions do not scale since there exist a large number of embedded devices. Other researchers work towards the bare-metal systems [79, 115, 65], i.e., embedded systems without an OS kernel or having a thin layer of abstraction. However, the methodology cannot be directly used to rehost

the Linux kernel as the Linux kernel is far more complicated than the bare-metal ones.

1.3 Our Work

Our work aims to lay the foundation of firmware analysis to facilitate both the static analysis and dynamic analysis technique. Specifically, we conduct an empirical study on the state-of-the-art ARM disassembly tools to find the limitation of current static analysis tools. In this work, we study eight popular disassemblers including three commercial ones (i.e., `IDA Pro` [22], `Hopper` [21], and `Binary Ninja` [11]) and five noncommercial ones (i.e., `Ghidra` [20], `arm-linux-gnueabi-objdump` [33], `angr` [138], `Radare2` [40], and `BAP` [59]). In particular, we evaluate these tools' capabilities on identifying three primitives, i.e., instruction boundary, function boundary, and function signature. Instruction boundary and function boundary are important as they are the fundamental primitives for other primitives to build upon (e.g., control flow graph and call graph). Meanwhile, function signature can help to build fine-grained control flow integrity (CFI) systems [159, 157, 121, 143], which can make the firmware more secure. We cross-compile 1,040 real-world programs and 19 benchmark programs to generate 1,896 different binaries. We feed the stripped binaries (binaries without debugging symbols) to the eight disassemblers and compare the disassembly results with the ground truth, which is generated with the help of debugging symbols. After comparing the differences between the ground truth and the disassembly results, we calculate the precision, recall, and F1-score. We conclude many interesting findings (Section 3.1) and point out the future directions for researchers.

For dynamic analysis tools, We build a general framework (i.e., Examiner) that can find the implementation bugs and deviations of ARM emulators. We apply Examiner on three different CPU emulators (i.e., QEMU [57], Unicorn [48], and

Angr [3]) and many inconsistent instructions, which behave differently between emulators and real devices, are located. Specifically, we design and implement the first symbolic execution engine for ASL [5]. With the ASL symbolic execution engine, we are able to generate sufficient instruction test cases that can cover different semantics of the ARM instructions. We utilize the different techniques and compare the result between the tested emulators and real devices in different architectures (i.e., ARMv5, ARMv6, ARMv7, and ARMv8). In this case, we build a deterministic differential testing engine that uses the generated test cases as inputs. We model the CPU state and provide the same context when executing an instruction stream on a real CPU and an emulator by inserting the prologue instructions. We then insert the epilogue instructions that can dump the execution result automatically for comparison. Finally, we generate 2,774,649 instruction streams that cover all the 1,998 ARM instruction encodings in four instruction sets (i.e., A64, A32, T32, and T16). Our system locate a huge number of inconsistent instruction streams (171,857 for QEMU, 223,264 for Unicorn, and 120,169 for Angr). Furthermore, we discovered 12 bugs (4 in QEMU, 3 in Unicorn, 5 in Angr) and all of them have been confirmed by developers. Furthermore, we build three different applications (i.e., emulator detection, anti-emulation, and anti-fuzzing) with the inconsistent instructions to demonstrate their usages.

In addition, to boost the capability and scalability of dynamic firmware analysis frameworks, we propose a new technique named peripheral transplantation and implement the prototype system (i.e., ECMO) to rehost the Linux kernels of embedded devices. Specifically, peripheral transplantation technique is device-independent and works towards the Linux kernel without the need of the source code. With peripheral transplantation, we do not need to manually add the emulation support for different kinds of peripherals. Instead, we transplant the device drivers of designated peripherals (if they are not initialized originally) into the target Linux kernel binary

and the emulated models of peripheral into QEMU. We apply ECMO on 815 Linux kernels from firmware images, including 20 different kernel versions and 37 device models. ECMO focuses on transplanting the early-boot peripherals (i.e., interrupt controller, timer, and UART), which are needed to rehost the Linux kernel. Our experiments shows that ECMO can successfully transplant the peripherals for all the 815 Linux kernels and 710 of them are able to launch a shell. The failed cases are due to the unsupported root file system format (ramfs). To demonstrate the capability of ECMO on supporting the other peripherals, we successfully install an Ethernet device driver (i.e., smc91x) on all the rehosted Linux kernels. Furthermore, we build and port three applications (i.e., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing) to demonstrate the usage and functionality of ECMO.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 reviews the related literature. Chapter 3 introduces the empirical study on ARM disassembly tools. This empirical study finds several bugs and explores the limitation of state-of-the-art disassembly tools and points out the potential directions for improvement. Chapter 4 proposes a new framework named EXAMINER. EXAMINER utilizes the differential testing techniques and can automatically locate the inconsistent instructions between the state-of-the-art emulators (i.e., QEMU) and hardware devices. Chapter 5 proposes a new framework named ECMO. ECMO utilizes a new novel technique named peripheral transplantation. With ECMO, we are able to rehost hundreds of Linux kernels from 815 different firmware images. Chapter 6 concludes our work and points out the future work.

Chapter 2

Literature Review

2.1 Disassembly Primitives

Instruction Boundary. Zhang et al. [159] combined the linear sweep and recursive traversal. However, their work is for x86 binaries and there is no experiment describing the accuracy of this algorithm. Ben et al. [58] proposed the idea of speculative disassembly on Thumb binaries with the assumption that binaries are not obfuscated. However, their work is not scalable to real world binaries as ARM instruction set are widely used. Though Kruegel et al. [102] proposed an algorithm on obfuscated binaries, their work does not target the ARM architecture. Bauman et al. [56] proposed the idea of superset disassembly, while Miller et al. [116] proposed the probabilistic disassembly mechanism. However, they only focus on x86/x64 binaries.

Function Boundary. Detecting the function boundary is also a challenging research topic. Rosenblum et al. [130] use the machine learning technique to identify functions. Other works [55, 59, 90, 100, 136] extended this idea with different machine learning algorithms. However, it is rather hard to build a general model. Other tools [36, 102, 138] use heuristics or hard-coded signatures to identify the function boundary. The fundamental problem is that there exist functions that do not have the signatures or do not follow the heuristics. Qiao et al. [123] applied static analysis to detect the function boundary. However, their work only targets the x86 architecture.

Dennis et al. [72] designed a new methodology on detecting function boundaries by analyzing control flow graphs. However, it assumes there is a distinguished function call instruction (e.g., the `call` instruction in x86/x64) in the binary. This mechanism cannot be applied to ARM due to no distinguished function call instruction in ARM binaries.

Function Signature. Function signature recovering is important to control flow integrity [143, 121, 157, 108] and data-dependency analysis [119, 132]. Previous work [143] utilize forward and backward static analysis. However, their work targets *x86/x64* binaries. Apart from the binary analysis technique, deep learning techniques [63] is used to infer the function signature automatically. However, the accuracy highly depends on the training data and is hard to be integrated into the state-of-the-art disassembly tools.

2.2 Emulator Testing and Applications

Testing. Several works are proposed to test the CPU emulators. Lorenzo et al. proposed EmuFuzzer to test the CPU emulators [113, 112]. However, the seed used for testing mainly relies on randomization and a CPU-assisted mechanism, which may not cover all the CPU behaviors. Apart from testing user-level instructions, KEmuFuzzer is proposed to test the whole system emulators [111]. However, KEmuFuzzer relies on the manually written template to generate test cases. For better test case coverage, PokeEMU [110] is proposed. PokeEMU utilizes binary symbolic execution to generate more test cases from a high-fidelity emulator and apply these test cases on low-fidelity emulators. However, whether the high-fidelity emulator strictly follow the rule of specification is unknown. Furthermore, all the above mentioned works target on x86/x64 architectures. With the development of embedded systems and mobiles, the faithfully emulating ability for ARM architecture is a urgent need. Our

work targets on ARM architecture and generates test cases from the specification itself (i.e., ARM ASL). The evaluation results show that we can find the real bugs and many inconsistent implementations between real devices and emulators, which can be abused by attackers. iDEV [124] focuses on the ARM instruction semantic deviation issue. However, EXAMINER is different from iDEV as EXAMINER utilizes the symbolic execution technique to generate test cases and can model the whole CPU state. In this case, we can find more inconsistent instructions compared with iDEV in theory. Furthermore, we evaluate EXAMINER on 4 different ARM versions and three CPU emulators, which shows the scalability and we also demonstrate how these inconsistent instructions can be used in practice with three different applications.

Applications. There are many applications based on QEMU. For example, researchers have developed new fuzzing systems [109, 160, 47] based on QEMU. KVM leverages the device emulation provided by QEMU or the virtio [131] framework for device virtualization. Virtual machine introspection tools [82, 146, 73, 74, 81, 54], which are helpful for debugging or forensic analysis, utilize QEMU to introspect the system states. Furthermore, dynamic analysis frameworks use QEMU to analyze malware behavior [77, 117, 152, 129, 154, 153]. ECMO provides the capability to rehost Linux kernels, which lays the foundation for apply these applications on embedded Linux kernels.

2.3 Differential Testing

Differential testing is introduced by McKeeman et al. [114] to detect the implementation bugs by comparing the inconsistent behaviors between different software. For example, Yang et al. proposed Csmith, a powerful tool that can generate multiple C programs. With Csmith, hundreds of bugs are detected in the C compiler. Regarding the same goal, Le et al. introduced equivalence modulo inputs (EMI) [105] and

many other differential testing tools are built based on EMI to validate the compiler implementations [106, 140].

Apart from testing compilers, researchers also utilize differential testing to validate the Database Management Systems (DBMS). Slutz et al. proposed the tool RAGS to explore bugs by executing different SQL queries on multiple DBMS. Though it is effective, it can only support a small set of SQL statements. Gu et al. evaluate the accuracy of DBMS optimizer by using options and hints to force the generation of different query plans. Jung et al. developed APOLLO [96] to test the performance regression bugs in DBMSs .

Furthermore, differential testing is powerful and applied to different domains such as testing SMT solvers [148, 147], JVM implementations [99] , symbolic execution engines [99], and PDF readers [103]. .

2.4 Firmware Analysis

2.4.1 Static Firmware Analysis

Researchers apply the static analysis technique to analyze the embedded firmware. For instance, Costin et al. [67] conduct a large-scale analysis towards the embedded firmware. By analyzing 32 thousand firmware images, many new vulnerabilities are discovered, influencing 123 products.

Code similarity is widely used to study the security issue of embedded devices. Feng et al. propose Genius [80], a new bug search system to address the scalability issues by translating binary control flow graph to high-level numeric feature vectors. The experiments show that Genius can identify many vulnerabilities in a short time. Considering the inaccuracy of approximate graph-matching algorithm, Xu et al. utilize neural network-based approach to abstract the control flow graph of binary function and build a prototype named Gemini [150]. The result shows Gemini

can identify more vulnerable firmware images compared with Genius. Yaniv et al. introduce a precise and scalable tool named Firmup [71] by considering the relationship between procedures. The result show Firmup has a relatively low false positive and effective on discovering vulnerabilities. In the case that firmware images are not available, Wang et al. [145] applies cross analysis of mobile apps to detect the vulnerable devices. Finally, 324 devices from 73 different vendors are discovered. Our system is used to analyze the firmware images of embedded systems with dynamic analysis. Application building upon ECMO can complement the static analysis ones.

2.4.2 Dynamic Firmware Analysis

Besides static analysis, researchers propose several methods to support the dynamic firmware analysis. Avatar [156] is proposed to support complex dynamic analysis of embedded devices by orchestrating the execution of an emulator and real hardware. Charm [142] applies a similar strategy. It introduces the technique named remote device driver execution by forwarding the MMIO operation to a real mobile. Avatar2 [118] extends Avatar to support replay without real devices. However, they both suffer from the problem of scalability. Inception [66] applies symbolic execution based on KLEE [60] and a custom JTAG to improve testing embedded software. However, it assumes that the source code is available. IoTFuzzer [62] aims to fuzz the firmware from the mobile side. However, the code coverage of firmware and the coverage of attack surface are limited. Pretender [86] is able to conduct automatically rehosting tasks. However, it replies on the debug interface of specific devices. Jetset [95] utilizes the symbolic execution to infer the return values of device registers. However, the functionality of the peripherals cannot be guaranteed. Furthermore, the shell may not be obtained for further development of different applications.

Besides, many researchers utilize the fuzzing technique to detect the security issues of embedded firmware. P2IM [79] is proposed to learn the model of periph-

erals automatically. DICE [115] focused on the DMA controller and can extend the P2IM's analysis coverage. Halucinator [65] proposed a new methodology to rehost the firmware by abstracting the HAL functions. ECMO are different from them in the aspects to transplant peripherals into the target kernel, instead of inferring the peripherals models. Besides, all these systems focus on bare-metal system, which is less complicated than the Linux kernel. Firmadyne [61] and FirmAE [101] target on Linux-based firmware. However, they focus on the user-space program, instead of the Linux kernel.

Chapter 3

An Empirical Study on ARM Disassembly Tools

3.1 Overview

ARM specific properties. ARM binaries have some unique properties, which bring challenges to disassemblers. First, inline data is common in ARM binaries while *"constructs like inline data and overlapping code are very rare"* in x86/x64 binaries [52]. Second, ARM provides two instruction sets: the ARM instruction set and the Thumb instruction set (which includes both 16-bit Thumb-1 and 32-bit Thumb-2 instructions). An ARM binary can contain both ARM and Thumb instructions and switch between them, which brings challenges on identifying the correct instruction set. Third, there is no distinguished function call instruction in ARM binaries, unlike the `call` instruction on x86/x64. Both branch and link instruction (`BL`) and branch instruction (`B`) can be used for function calls or direct branches in the ARM instruction set or Thumb instruction set. This makes identifying functions more challenging as a branch jump may be mis-identified as a function call, resulting in false positives. Due to these unique properties, there is a need to perform an extensive and thorough evaluation of ARM disassembly tools.

We perform an empirical study on ARM disassembly tools. In particular, we eval-

uate these tools' capabilities on identifying three primitives, i.e., instruction boundary, function boundary, and function signature. Instruction boundary and function boundary are fundamental primitives that other primitives (e.g., control flow graph and call graph) are built upon while function signature is important to control flow integrity (CFI) techniques [159, 157, 121, 143]. To make the study comprehensive, it should meet the following requirements:

1. Our evaluation should use diverse programs, including both popular benchmarks and, more importantly, different types of representative programs in the wild.

2. Our evaluation should cover programs compiled in different compilers with various compiling options, e.g., different instruction sets and optimization levels.

3. Our evaluation should consider tools in different options and versions. The option of a tool can affect the disassembly accuracy while the version of a tool can reveal the improvements over time.

4. Our evaluation should include obfuscated binaries [158], since they do exist in the wild and could affect the results of disassembly tools.

To this end, we cross-compile $1,040$ real-world programs and 19 benchmark programs. In total, we get $1,896$ binaries, where 608 are compiled from the SPEC CPU2006 with different compiling options. Among the remaining ones, $1,040$ are compiled from Android daemons, libraries, and user-space programs of embedded systems (i.e., OpenWRT [34]). We also build 248 obfuscated binaries using O-LLVM [98], which is one of the most popular obfuscators, with multiple obfuscation methods. Then we obtain the ground truth with the help of debugging symbols and feed the stripped binaries (binaries without debugging symbols) to eight state-of-the-art ARM disassembly tools including three commercial ones (i.e., `IDA Pro` [22],

`Hopper` [21], and `Binary Ninja` [11]) and five noncommercial ones (i.e., `Ghidra` [20], `arm-linux-gnueabi-objdump` [33], `angr` [138], `Radare2` [40], and `BAP` [59]). For each tool, we select two different versions to study the improvement of these tools over time. The time gaps between two different versions range from 272 days to 735 days. Finally, we measure the precision, recall, and F1-score by comparing the differences between the ground truth and the disassembling result.

Based on the result, we conclude some findings that were not systematically summarized and/or confirmed.

1. The unique properties of ARM binaries do bring challenges to the disassembly tools, especially the two different instruction sets (i.e., the ARM instruction set and the Thumb instruction set) and the mixed use of the `BL label` and `B label` instructions for both function call and branch jump. Disassembly tools do not have a good support to binaries in Thumb instruction set. The precision and recall for disassembling Thumb instructions are usually lower than that of ARM instructions (more than 90% in maximum).

2. Disassembly tools do not support well on recovering function signatures. Among all the tools, `Radare2` performs best with only 0.417 recovery ratio, which indicates that function signature recovering is still challenging for both commercial and noncommercial tools.

3. Many factors, including compilers, compiling options, target CPU architectures, can affect the result. However, the root cause is still due to the unique properties of ARM binaries. Furthermore, the robustness and scalability of disassembly tools should be improved. We observed several exceptions, segment faults and timeout during the analysis.

4. Noncommercial tools gain larger improvement in terms of accuracy. This is

because they perform worse in the old version and the communities can provide feedback and suggestions to help improve the tools. As for performance, we noticed that most of the tools' performance improved a lot and the performance can decrease due to the added features and algorithms.

We have reported our findings along with failed test cases to developers of the evaluated tools [24, 23, 25, 26]. Developers of `Binary Ninja`, `Hopper`, and `angr` verified our findings and provided updates based on the failed cases. `Radare2` assigned bug tag to them. `Ghidra` verified our findings and provided the potential solutions, while `BAP` declared that they would solve the problem in the future.

3.2 Background

3.2.1 Different CPU Architectures and Instruction Sets

ARM has multiple CPU architectures, each with different instruction extensions and features. When building programs, developers can specify the target CPU architecture, e.g., ARMv5 or ARMv7, through compiling options (`-march`). For instance, ARMv5 is the default CPU architecture of the GCC compiler.

Moreover, there are two instruction sets, i.e., the ARM instruction set and the Thumb instruction set. The former is 32-bit long, while the latter is 16-bit long and designed for size-sensitive applications, which is available for ARMv4T CPU architecture and later versions. Since ARMv6T2, Thumb-2 is introduced. It offers *"best of both worlds"* compromise between the ARM instruction set and the Thumb instruction set. It has access to both 16-bit and 32-bit instructions. In this thesis, we use the Thumb instruction set to denote both Thumb and Thumb-2 instruction encoding.

A single binary can contain multiple instruction sets and switch between them, e.g., switching between ARM instructions and Thumb (Thumb-2) instructions. The

C

```
uint8 foo(uint8 x, uint8 a,
          uint16 b, uint16 c)
{
    if (a==2) x += (b >> 8);
    else   x += (c >> 8);
    return x;
}
```

Thumb

```
0x00: 2902  CMP     r1,#2
0x02: d101  BNE     {pc}+0x6 ; 0x8
0x04: 0a11  LSRS    r1,r2,#8
0x06: e000  B       {pc}+0x4 ; 0xa
0x08: 0a19  LSRS    r1,r3,#8
0x0a: 1808  ADDS    r0,r1,r0
0x0c: 0600  LSLS    r0,r0,#24
0x0e: 0e00  LSRS    r0,r0,#24
0x10: 4770  BX      lr
```

Thumb-2

```
0x00: 2902      CMP     r1,#2
0x02: bf14      ITE     NE
0x04: eb002013  ADDNE   r0,r0,r3,LSR #8
0x08: eb002012  ADDEQ   r0,r0,r2,LSR #8
0x0c: b2c0      UXTB    r0,r0
0x0e: 4770      BX      lr
```

ARM

```
0x00: e3510002  CMP     r1,#2
0x04: 10800423  ADDNE   r0,r0,r3,LSR #8
0x08: 00800422  ADDEQ   r0,r0,r2,LSR #8
0x0c: e20000ff  AND     r0,r0,#0xff
0x10: e12fff1e  BX      lr
```

Figure 3.1: The source code and its corresponding ARM, Thumb and Thumb-2 instructions.

switching can occur explicitly by executing branch instructions or implicitly specified by branch targets. For instance, the `BLX label` instruction always changes the instruction set from ARM to Thumb or vice versa. However, the `BX Rm` derives the target instruction set from $bit[0]$ of the register Rm. If it is 0, then the target instruction set is ARM. Otherwise, it is Thumb. The target instruction set of other branch instructions, e.g., `POP {PC, Rm ...}`, also depends on the last bit of the target address. This brings serious challenges for disassembly tools to *statically* determine the target instruction set, especially for the ones that leverage linear sweep strategy (Section 3.2.2).

Figure 3.1 illustrates the source code of a function and the binary compiled using ARM, Thumb and Thumb-2 instructions. Note that, for the two popular compilers (e.g., GCC and Clang), the default instruction set is ARM, and the option `-mthumb` is used to change it to Thumb. The instruction set greatly affects the accuracy of disassembly tool, which we will discuss in Section 3.4.

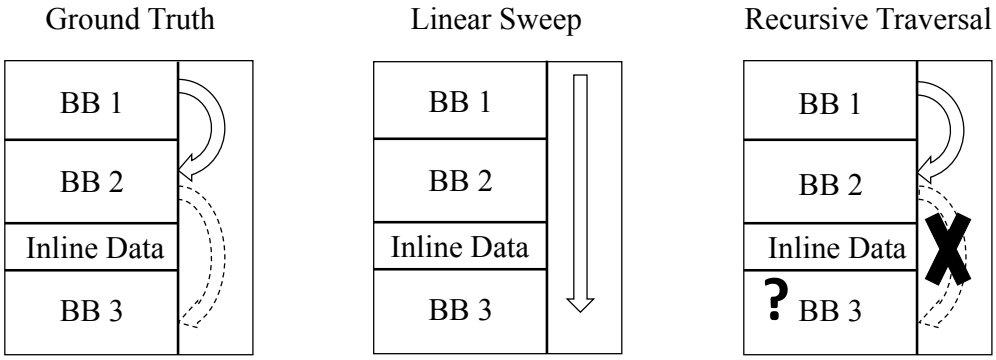


Figure 3.2: Two disassembly strategies. The function has three basic blocks (BBs), and inline data between BB2 and BB3. There is a direct jump from BB1 to BB2 and an indirect jump from BB2 to BB3

3.2.2 Disassembly Strategies

To understand the capability of disassembly tools, we use three primitives, i.e., instruction boundary, function boundary, and function signature in our study. To precisely detect the instruction boundary, a disassembly tool should be able to locate the inline data inside the binary and the correct instruction set (ARM or Thumb).

There are two different disassembly strategies [122, 135]. One is linear sweep, which linearly decodes the code sections. It is used by disassemblers such as the GNU utility `Objdump`. However, the inline data (data inside the code section), which is normal in ARM binaries, and instruction set switching cannot be detected by this strategy since it does not consider the control flow transfers. Figure 3.2 shows a function with three basic blocks and inline data between the basic block 2 and the basic block 3. The `Objdump` tool fails to determine the boundary between code and data, and disassembles the inline data as code.

Another strategy is recursive traversal. Its basic idea is disassembling code from the entry point of a binary, and then recording the branch targets as new entry points (usually appends these branch targets into a list). It repeats this process until no new targets could be found, and all the targets in the list have been traversed. The ad-

vantage of this strategy is that it is unlikely to disassemble inline data as code, since there should be a control flow transfer instruction before the inline data (otherwise, the data will be executed as code at runtime). Moreover, it can handle instruction set switch if the branch target can be determined statically (direct branches). However, the disadvantage is that some code regions may be missed, if they cannot be reached through direct branches. As in Figure 3.2, the code in the basic block 3 may be missed since this block can only be reached through an indirect branch, whose target is determined at runtime. Note that, even though methods [64] have been proposed to detect the targets of a jump table (one type of indirect branches), how to reliably detect other types of indirect branches (e.g., function pointers) is still an open research question. We do find that resolving indirect jump targets can improve the result of disassembly tools (Section 3.4.4).

3.2.3 Function Boundary

Function boundary is an important primitive, which can be used to construct other primitives, e.g., function call graph. Previous work usually leverages function prologue and epilogue patterns to detect functions. The method proposed in [59, 55] scans a binary for known function prologues and epilogues. However, this method is limited by the fact that the prologue and epilogue pattern of a function could be missing or rare, which is common in optimized code. It is rather challenge to maintain a sound and complete pattern database.

Due to these limitations, a compiler-agnostic function identification method was proposed in `Nucleus` [72]. The basic idea is to analyze the inter-procedural control flow graph (ICFG) and conduct connected component analysis based on the generated ICFG. However, `Nucleus` assumes that the binary should have distinguished function call instructions, e.g., the `call` instruction for x86. Unfortunately, this assumption does not hold in ARM binaries. For example, binaries in Thumb in-

struction set use `BL label` instruction for both a direct function call and a direct branch since the range of the branch target is larger than the `B label` instruction. As a result, it causes many false positives to the function detection. We observed a significant decrease of the precision value for the function boundary of binaries in Thumb instruction set (Figure 3.8b).

3.2.4 Function Signature

Function signature, which is also known as type signature, defines the input (number of the parameters) and output (return a value or not) of a function. Function signature is important as it helps to identify the indirect calls in binaries [143, 108], which can protect the control flow hijacking attacks in binary level.

However, it is rather challenging to identify the function signatures from stripped binaries as the language-level information is lost after compilation. Some works propose to use neural networks [63] to identify the function signatures. However, the accuracy highly depends on the training data. What's more, it is not practical to be integrated into the state-of-the-art disassembly tools as the training process is quiet slow and a pre-defined model cannot cover all kinds of cases and is limited by the training data, too.

3.3 Methodology

Figure 3.3 shows the roadmap of this study. First, we build various programs including popular benchmarks and real-world applications (①), using different compilers (GCC and Clang) with diverse compiling options (②) to generate the target binaries. This aims to cover popular compilers and different scenarios that programs are built with different options. After that, we first generate the ground truth by leveraging the debugging symbols (③), and then remove the symbols (④) and feed the stripped binaries to disassembly tools (⑤). We retrieve the result of identified

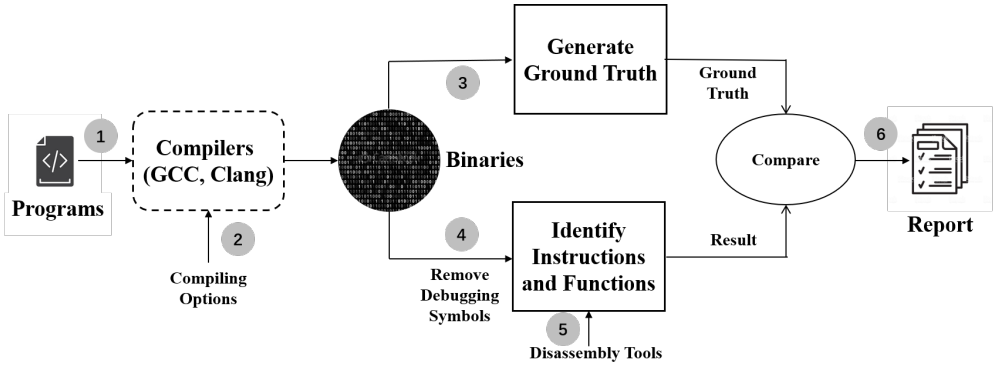


Figure 3.3: Roadmap of our study

instructions, functions, and function signatures for each tool, and compare the result with the ground truth (⑥) to generate the final report, which contains the recall and precision value. We present the main steps of our study in the following sections.

3.3.1 Prepare Binaries.

One may think it is straightforward to compile binaries for evaluation. However, to make our study comprehensive, we need to consider the diversity of compilers, compiling options, types of programs, and obfuscation methods, which will affect the result.

Diverse Compilers. We use two compilers, i.e., GCC and Clang, each with three different versions. Specifically, we use GCC versions 6.5, 7.5 and 8.3 and Clang versions 7.0, 8.0 and 9.0, which cover the major compilers used in the wild.

Representative Compiling Options. As mentioned in Section 3.2, different compiling options (e.g., with or without `-mthumb`) would result in completely different binaries. Considering the diversity of compiling option, we aim to understand which compiling options are mostly used in the real world. Thus, we divide ARM binaries into three types, according to systems they are used.

1. *Type-I: Embedded OSes.* They are used in resource-constrained ARM devices,

mainly the ARM Cortex-M processor families with low computational power. We select FreeRTOS v10.1.1 [46], the most popular real time operating system, and Mbed OS (version 5) [7], the open-source embedded operating system designed for IoT. There are several projects in FreeRTOS, each supports a different development board (or device). We cross-compile all the projects that support the ARM architecture. For the Mbed OS, we compile all the targets that support the ARM architecture.

2. *Type-II: Linux Kernel.* Linux kernel has been used on ARM devices widely. Such devices include mobile devices and ARM servers. For mobile devices, we use the most popular operating system Android, and build the kernel (version 4.4.169) for Android 9.0 (code name: Pie). We also build the kernel for Debian (version 9.6.0), one of the most popular Linux distributions for desktop computers and servers. We download and cross-compile the kernel (version 4.9.144) from Debian's official repository.

3. *Type-III: User-level Programs.* We also use user-level programs, including daemons, libraries used on mobile devices, desktop computers and servers. Specifically, we build user-level programs from Buildroot [12], Android Open Source Project (AOSP, version 9.0.8) [2], and the popular Debian packages. They are representative programs for low-end embedded systems, mobile devices, and ARM desktop/servers. In particular, Buildroot is commonly used in low-end embedded systems, including routers, IP cameras and etc. We compile all binaries targeting ARM development boards. For Debian packages, we use the top five mostly installed packages, i.e., `libpam-modules`, `libattr1`, `libpam0g`, `zlib1g`, and `debianutils`, ranked by the Debian popularity contest [18].

Table 3.1 shows the result of compiling options for Type-I, Type-II and Type-III binaries. We find that the Thumb instruction set is mostly used in Type-I binaries,

Table 3.1: The compiling options for Type-I, Type-II and Type-III binaries. The third column shows the number of total object files (.o files), and the last two columns show the number of object files with the Thumb instruction set and optimization levels.

	Name	# of Objects	Boards/Targets	Thumb	Optimization Levels
Type-I	Mbed	39,183	61	39,183	{'Os': 39,183}
	FreeRTOS	87	9	22	{'Os': 24, 'O2': 32}
Type-II	Linux Kernel (Android)	1,361	1	0	{'Os': 1, 'O2': 1,291, 'O0': 1}
	Linux Kernel (Debian)	1,860	1	0	{'O3': 1, 'O2': 1,788, 'O0': 1, 'Os': 1}
Type-III	AOSP	3,384	1	2,875	{'Os': 2,787, 'O2': 299, 'O0': 27,'O3': 69}
	Buildroot	188,387	103	1,677	{'Os': 188,387}
	Debian Packages	339	5	0	{'O2': 305, 'O3': 34}

and `O2` and `Os` are commonly used optimization levels. Due to this observation, we compile the benchmark programs for the evaluation to both ARM and Thumb instruction sets with `O2` and `Os` optimization levels to reflect real situations of ARM binaries in the wild.

Different Types of Programs. We use three types of programs in our study. They include the widely used benchmark, i.e., SPEC CPU2006, binaries in AOSP and OpenWRT. The latter two represent the binaries for mobile systems and IoT devices.

Specifically, we compile the SPEC CPU2006 using both Clang and GCC compilers with two optimization levels (`Os` and `O2`), two instruction sets (ARM and Thumb) and two CPU target architectures (`-march` with ARMv5 and ARMv7). In total, we get 608 binaries, i.e., 19 benchmark programs $\times$ 2 instruction set $\times$ 2 optimization levels $\times$ 2 compilers $\times$ (3 compiler versions + 1 specific CPU architecture with latest version of compilers) = 608 binaries. Considering the popularity of IoT and mobile systems, we build the latest Android Open Source Project (AOSP version 9) and extract daemon binaries (127 in total) and libraries (667 in total). Also, we build the latest stable version of OpenWRT (version 18.06). There are 12 different target boards that support the ARM architecture. In total, we get 246 binaries.

Obfuscation. To evaluate the impact of obfuscation, we use the O-LLVM [98], an open-source obfuscator, to compile the SPEC CPU2006. O-LLVM supports three different obfuscation methods. Specifically, instructions substitution (`sub`) is used to replace standard operators with more complicated instruction sequences. The bogus control flow (`bcf`) changes a function's control flow graph by adding basic blocks. The control flow flattening (`fla`) uses the control flow flatten algorithm [104] to create a large number of fake control flows. We apply each obfuscation method to each program, and then combine three methods together. Since the bogus control flow graph (*bcf*) consumes too much time (more than 2 hours) when applying it to C++ programs, we do not apply this method to C++ programs.

Summary: In total, we get $1,896$ binaries including 248 obfuscated ones. We believe this dataset is representative to demonstrate the diversity of compilers, compiling options, target architectures and types of devices.

3.3.2 Determine Disassembly Primitives

In this work, we consider the instruction boundary, function boundary, and function signature as fundamental primitives (Table 1.1). Other ones (e.g., direct control flow graph and call graph) could be built upon them.

Instruction Boundary. Instruction boundary refers to the start offset of an instruction, as well as the correct instruction set (ARM or Thumb). The purpose is two-fold. First, it is used to distinguish between code and inline data. Inline data is commonly used in ARM binaries, e.g., for the PC-relative addressing. This is different from x86 binaries, which do not contain inline data [52] except for the jump tables of binaries compiled by Visual Studio. Second, it is used to distinguish between ARM and Thumb instruction sets. This is challenging since the instruction set is partially determined by the target address of an (indirect) branch instruction, which is hard to be obtained by static analysis.

Function Boundary and Signature. Function boundary refers to the start offset of a function. Function boundary recognition is a necessary primitive to construct the call graph, which is critical to the whole program analysis.

Function signature refers to the number of the parameters for a function and whether the function returns a value or not (void function). Function signature is a necessary primitive as it helps a lot in identifying the indirect function calls and protect the software from the control flow hijacking [143].

3.3.3 Generate Ground Truth

After determining the primitives, we need to get the ground truth. However, even with debugging symbols, it is not straightforward to directly get the result. we describe our approaches as follows.

Instruction Boundary. We use mapping symbols [6] in the binaries to get the information of the instruction boundaries. Mapping symbols are generated by compilers to identify inline transitions between code and data, as well as ARM and Thumb instruction sets. There are three types of mapping symbols, including:

- `$a`: Start of a region of code containing ARM instructions.
- `$t`: Start of a region of code containing Thumb instructions.
- `$d`: Start of a region of data.

For instance, the mapping symbol "`0001043c $t`" denotes that the offset `0x0001043c` in the binary is code (not inline data), with the Thumb instruction set.

However, mapping symbols only include the start address of the code and data regions without indicating the offset and the instruction set of each instruction in the region. To deal with this issue, we use Capstone [13] to retrieve the offset of

each instruction. It works well since we have the instruction set (ARM/Thumb) information of each code region to help Capstone disassemble the code region.

Note that, the mapping symbol is an architecture-specific extension of the ARM ELF file. It may not exist in other architectures. By leveraging it, our system can detect the instruction boundary with a sound and complete result. Previous work can only detect 98% of the ground truth and requires a manual verification [52].

Function Boundary and Signature. We leverage DWARF [76], a debugging file format to retrieve the function boundary and signature. DWARF uses the data structure named Debugging Information Entry (DIE) to describe each variable, type, and function, etc. Each DIE has a tag (i.e., `DW_TAG_subprogram`) for function and each function has a key (i.e., `DW_AT_low_pc`) to represent the function start address. For function signature, DIE utilize the key `DW_TAG_formal_parameter` and `DW_AT_type` to represent the number of parameters of a function and whether the function returns a value or not (void function), respectively.

We extract the `DW_TAG_subprogram`, `DW_AT_low_pc`, `DW_TAG_formal_parameter`, and `DW_AT_type` from the DWARF information of each binary to get the ground truth.

3.3.4 Extract the Result

We evaluate eight state-of-the-art ARM disassembly tools, including five noncommercial ones, i.e., `angr` [138], `BAP` [59], `Objdump` [33], `Ghidra` [20], `Radare2` [40], and three commercial ones, i.e., `Binary Ninja` [11], `Hopper` [21] and `IDA Pro` [22]. Furthermore, to study whether these tools' accuracy and efficiency are improved in a rather long period. We collect two different versions for all the eight tools. The release date of the two versions ranges from 272 days to 735 days (Table 3.3). For each tool, the version released earlier is named old version while the latter one is named new version. Each tool has different ways to extract the instruction boundary, function boundary, and function signature. We carefully read the manual of each

tool and write a script to extract the result. Table 3.2 lists the summary of each tool, including the tool type (e.g., noncommercial or commercial) and APIs used to retrieve the result.

Table 3.2: A summary of the evaluated disassembly tools and the APIs or commands that are used to retrieve the results. NA: not applicable.

Tool Type	Tool	Instruction Boundary		Function Boundary	Function Signature	
		Code/Data	ARM/Thumb		Has Return	# of Parameter
Noncommercial	`angr`	graph.node.instruction	address% == 1	cfg.kb.functions()	VariableRecoveryFast() CallingConvention() func.has_return	VariableRecoveryFast() CallingConvention().cc.args
	`BAP`	-dasm	NA	-dasm	NA	NA
	`Objdump`	-d	NA	NA	NA	NA
	`Ghidra`	getFirstInstruction() getInstructionAfter()	getRegister(`TMode')	getFirstFunction() getFunctionAfter()	getSignature() getReturnType() getName()	getParameterCount()
	`Radare2`	pD	NA	afl	afcf	afi
Commercial	`Binary Ninja`	binaryview.instructions()	functions.arch	binaryview.functions()	func.return_type	func.parameter_vars
	`Hopper`	getTypeAtAddress()	getArchitecture()	getEntryPoint()	signatureString()	signatureString()
	`IDA Pro`	isCode()	GetReg(inst,`T')	get_func()	guess_tinfo() tif.get_func_details() tif.get_rettype()	guess_tinfo() tif.get_func_details()

3.4 Evaluation

As discussed in Section 3.3, we build 1,896 binaries (including 248 obfuscated ones) to evaluate eight disassembly tools in different versions. We address the following research questions in Section 3.4.2, Section 3.4.3, Section 3.4.4, ,Section 3.4.5 and Section 3.4.6 respectively.

- **RQ1**: What is the accuracy of disassembly tools towards the whole data set?
- **RQ2**: What are the factors that affect the accuracy of disassembly tools, and what are the reasons?
- **RQ3**: Do different types and options of tools have different results?
- **RQ4**: How efficient are these disassembly tools?
- **RQ5**: How well are these disassembly tools improved (in around one year)?

3.4.1 Evaluation Metrics

We use *precision* and *recall* to measure the accuracy (or effectiveness) of a tool on identifying instruction and function boundary. The definition of these two metrics is in equation 3.1.

$$precision = \frac{tp}{tp + fp} \, recall = \frac{tp}{tp + fn} \tag{3.1}$$

In the equation, we use *tp*, *fp*, *fn* to denote true positives, false positives and false negatives. *Recall* measures the ratio of true positives to the ground truth. A disassembler with high false negatives may have low recall. *Precision* measures the ratio of true positives to the result of a tool. A disassembler with high false positives may have low precision.

Considering the importance of both recall and precision, we also compute the F1 score according to equation 3.2. F1 score can reflect the overall accuracy of a tool.

$$F1\ Score = \frac{2 \times recall \times precision}{recall + precision} \tag{3.2}$$

Measuring the accuracy of function signature recovery is different from the other two primitives as the corresponding function boundary should be identified correctly. Thus, we only consider the correctly identified functions (true positives). We use N_F to denote the number of correctly identified functions. Among the identified functions, we use N_F_S to denote the number of functions whose signatures (i.e., return value and number of parameters) are recovered correctly. We calculate the recovery ratio of function signature with equation 3.3.

$$Recovery\ Ratio = \frac{N\_F}{N\_F\_S} \tag{3.3}$$

3.4.2 Accuracy of the Disassembly Tools (RQ1)

Table 3.3: The result of whole data set. *Prec.* means precision. *Rec.* means Recall. *Inva.* means the number of binaries that tool cannot identify any instructions or functions. *Tim.* means the number of binaries that tool cannot finish the analysis in two hours (CPU time). *Exce.* means the number of binaries that tool raises exceptions during the analysis. *Seg.* means the number of binaries that tool triggers a segment fault during the analysis. *Return* means the recovery ratio for evaluating whether the function return a value or not (void function). *Args* means the recovery ratio for evaluating the parameter number of the function. *All* means that both *Return* and *Args* are correct.

Tool	Release Information		Instruction Boundary				Function Boundary				Function Signature			*Tim.*	*Exce.*	*Seg.*
	Version&Date	Period	*Prec.*	*Rec.*	*F1*	*Inva.*	*Prec.*	*Rec.*	*F1*	*Inva.*	*Return*	*Args*	*All*			
BAP	1.6.0, Apr-19	421	0.887	0.277	0.309	**11**	0.533	0.358	0.387	**11**	-	-	-	**214**	244	0
	2.1.0, May-20		0.727	0.26	0.314	0	0.902	0.298	0.384	0	-	-	-	11	244	0
Objdump	2.30, Jan-18	735	0.702	0.75	0.722	0	-	-	-	-	-	-	-	0	0	0
	2.34, Feb-20		0.701	0.748	0.721	1	-	-	-	-	-	-	-	0	0	0
Ghidra	9.0.4, May-19	272	0.954	0.828	0.873	0	0.855	0.714	0.766	0	0.015	0.345	0.007	13	0	0
	9.1.2, Feb-20		0.954	0.826	0.874	1	0.853	0.682	0.746	0	0.015	0.345	0.007	13	0	0
Radare2	3.6.0, Jun-19	546	0.749	0.837	0.788	0	0.906	0.432	0.521	0	**0.765**	0.567	**0.417**	26	9	0
	5.0.0, Dec-20		0.759	0.847	0.798	1	0.851	0.508	0.591	0	0.746	0.586	0.417	0	0	0
angr	8.19.4.5, Apr-19	571	0.886	0.797	0.83	1	0.404	0.667	0.490	1	0.723	0.366	0.246	16	**364**	**262**
	9.0.4663, Oct-20		0.928	0.984	0.955	1	0.382	0.711	0.486	1	0.705	0.577	0.368	13	292	0
Binary Ninja	1.1.1470, Jan-19	643	0.984	0.857	0.900	0	0.806	0.800	0.781	0	0.301	0.548	0.196	37	0	0
	2.2.2487, Oct-20		0.986	0.869	0.908	1	0.834	0.828	0.808	0	0.412	0.572	0.254	2	0	0
Hopper	4.5.13, Jul-19	347	0.971	**0.986**	**0.978**	1	0.825	**0.816**	0.807	0	0.387	**0.604**	0.271	2	0	0
	4.5.29, Jun-20		0.971	0.986	0.978	0	0.825	0.816	0.807	0	0.387	0.604	0.271	2	0	0
IDA Pro	7.3, Jun-19	410	**0.994**	0.970	**0.978**	5	**0.944**	0.781	**0.838**	5	0.293	0.285	0.04	1	0	0
	7.5, Jul-20		0.994	0.971	0.979	1	0.911	0.788	0.825	0	0.298	0.330	0.055	1	5	0

Table 3.3 shows the overall result. The recall, precision, F1 score, and recovery ratio are computed in the granularity of macro-averaging. A tool may not be able to detect any instruction or function for a given binary. We mark such cases with the flag *Invalid* (*Inva.* in table 3.3). We also set a threshold (two CPU hours in our study) for each tool to analyze a binary. This is because if a tool cannot finish the analysis in two hours, then it is not scaled to analyze a large number of binaries. We count the number of binaries that cannot be analyzed in two CPU hours with the flag *Timeout* (*Tim.* in table 3.3). We also count the number of binaries that trigger an exception or a segment fault for each tool. We mark them with the flag *Exception* (*Exce.* in table 3.3) and *Segfault* (*Seg.* table 3.3), respectively.

Note that, a tool may have different options and versions when performing the analysis. For instance, `angr` provides an option to disable or enable the resolution of indirect jumps. We use the default option for each tool to calculate the overall result and leave the evaluation of the impact of different options in Section 3.4.4. As for the different versions, we select two different versions (Section 3.3.4) for each tool to study the improvement of these tools (in around one year). The answers to research questions 1 to 4 mainly based on the old version. We will discuss the results of the new version in detail in Section 3.4.6.

Instruction Boundary. `IDA Pro` has the highest precision value, while `Hopper` owns the highest recall value. Both of them have the highest F1 scores and are commercial tools. Moreover, these two are robust, since they do not raise any exceptions or generate any segment faults during the analysis. Among all the tools, `BAP` does not perform very well on both the instruction boundary and the function boundary. This is due to the insufficient support of the Thumb instruction set. Besides, `BAP` does not disassemble instructions that are out of the range of recognized functions. That means if a function cannot be detected, then all instructions inside that function will

be ignored. This is the reason why the recall of the instruction boundary is rather low. For other tools, the reason for the lower precision and recall mainly comes from two different reasons. One is the challenges raised by mixed ARM and Thumb instruction sets, and the other is the inline data.

Function Boundary. `IDA Pro` still has the highest precision while `Hopper` has the highest recall. In terms of F1 score, `IDA Pro` has the highest value. It means that the function boundary is correlated with the instruction boundary. `BAP` mainly uses function prologue patterns learnt from a set of binaries to detect the function boundary. Due to the imprecise function prologue patterns, functions with no representative prologue patterns cannot be detected by `BAP`. As for `Radare2`, the recall is relatively low compared with other tools. That is because `Radare2` has a very strict policy on detecting functions. Users can use the command `aaaa` to explore more functions by searching for the function patterns.

Function Signature. Recovering function signature is not easy. Among all the eight disassembly tools, six of them support recovering function signature. The overall recovery ratio (both the return value and parameter counts are right) for `Ghidra` is less than 0.01, indicating that `Ghidra` has little support on stripped ARM binaries. We further analyze the reasons. 98% functions' return types are *undefined* in `Ghidra`. *Undefined* is the default value and means `Ghidra` does not know the return type.

We noticed that `Radare2` has the highest recovery ratio on identifying the return value (0.765) while the recovery ratio on identifying the number of parameters is 0.567. The overall recovery ratio drops to 0.417, which is rather low (though the highest among all the tools) and has a large improvement space. Commercial tools do not perform better on recovering the function signatures. All of the three tools' recovery ratios are less than 0.3. In specific, `IDA Pro` has the lowest accuracy on

recovering function signature while the function boundary identification accuracy is the highest. Discussed with the developer of `IDA Pro`, they also admit that *"For stripped binaries, there is very little info remaining and recovering parameters just from the disassembly is quite difficult."*

Robustness and Scalability. We find some noncommercial tools are not robust. For instance, more than 600 binaries triggered either an exception or a segment fault of `angr`. For the 262 binaries that triggered a segment fault, 160 of them are binaries compiled from the SPEC CPU2006, and 87.5% (140/160) of them are compiled using the Thumb instruction set. Based on this observation, there is a great space for `angr` to improve the support of the Thumb instruction set. This observation also applies to other tools, e.g., `Radare2`.

`BAP` does not scale well because 214 binaries cannot be analyzed in two hours, which is far more than other tools. We also observed timeouts when evaluating other tools except `Objdump`. For example, 37 binaries cannot be analyzed by `Binary Ninja` in two hours.

Failed Cases. Disassembly tools failed to identify the right instruction boundary, function boundary, or function signature due to different kinds of reasons. We manually study the failed cases and find that tools utilizing function prologue patterns to identify the function boundary can make mistakes due to insufficient function prologue patterns. Apart from this, disassembly tools cannot identify the accurate instruction boundary due to the inline data and mixed instruction set. We illustrate three different types of failed cases in the following.

Figure 3.4 shows an example of `BAP`. There is a function starting from the offset `0x1e958`, but `BAP` thinks the function starts from the offset `0x1e960`. That is because the function prologue pattern used by `BAP` is not precise enough. Since ARM binaries vary due to different compilers and compiling options, it is challenging to timely

```
Ground truth function start
0001e958      mov       r2, #0x0
0001e95c      mov       r3, #0x0
BAP function start
0001e960      push      {r4, r5, r6, r7, r8, sb, sl, fp, lr}
0001e964      mov       sl, #0x0
0001e968      sub       sp, sp, #0x34
0001e96c      str       r2, [sp, #0x58 + var_48]
0001e970      str       r3, [sp, #0x58 + var_44]
0001e974      ldr       r2, =loop_length
```

Figure 3.4: `BAP` (version 1.6.0 and version 2.1.0) mis-identify the function boundary

```
00096a7c.     ldr       r4, [pc, #528]
[pc, #528] refers to: 0x96a7c+528+4 = 0x96c90
...
00096c8a      bl        func_11880
func_11880: does not return
00096c8e      mov       r8, r8
00096c90      stc2l     p15, c15, [ip, #0x3fc]
00096c94      vhadd.s8  d0, d0, d12
00096c98      mrc2      p15, #0x7, apsr_nzcv, c11, c15, #0x7
func_96c9c : indirect invoked
00096c9c      push      {r4, lr}
00096c9e      ldr       r1, =0xbcd60
00096ca0      bl        func_11730
00096ca4      pop       {r4, pc}
```

Figure 3.5: `Hopper` (version 4.5.13 and version 4.5.29) disassemble data as code.

update function prologue patterns. This can result in a false positive (`0x1e960`) and false negative (`0x1e958`).

Figure 3.5 shows an example that is caused by inline data. For instance, `Hopper` disassemble the inline data (starting from the offset `0x96c8e`). This is because function `0x11880` is a non-return function and function `0x96c9c` is indirect invoked. Thus, `Hopper` treat the inline data as code and cannot identify the function `0x96c9c`.

Figure 3.6 shows a failed case where `Radare2` uses a wrong instruction set to disassemble the binary. The instruction set from the offset `0x94050` is Thumb. However, `Radare2` disassembles it using the ARM instruction set, although it is an invalid in-

```
0009404c      .dword      0x000ce700
func_94050:indirect invoked
00094050      invalid
00094054      ldrbmi      r4, [r6], r6, asr 12
00094058      stcmi       p5, c11, [0x00094360]
0009405c      stmdavs     fp!, {r3, r4, r7, sb, sl, lr}
00094060      movwls      fp, 0x908a
00094064      stmdavs     ip, {r8, sb, sp} ^
00094068      movwls      sb, 0x2301
```

Figure 3.6: `Radare2` (version 3.6.0 and version 5.0.0) identifies Thumb instruction set as ARM instruction set.

struction. We further locate the potential root cause of this error. Specifically, the basic block (`0x94050`) is indirectly reached from other basic blocks, thus it is hard for the tool to determine the right instruction set. Remember that, the instruction set is determined by the last bit of the target address.

3.4.3 Factors that Affect Accuracy (RQ2)

Our data set consists of binaries that are built using different compilers, compiling options, and target architectures. Some of them are even obfuscated. They represent the diversity of existing binaries in the wild. In the following, we further explore multiple factors that affect the accuracy of disassembly tools. Note that nearly all the tools have difficulty recovering the function signature, which indicates that accurately recovering function signature is challenging. Thus, we focus on the instruction boundary and function boundary while answering research question 2.

Instruction Sets

ARM and Thumb instruction sets are widely used in real-world binaries. To evaluate the impact of instruction sets, we divide binaries into two categories. The first one contains binaries compiled with the flag `-mthumb`, which use the Thumb instruction set. We call them Thumb set binaries. The other one is compiled *without* the flag

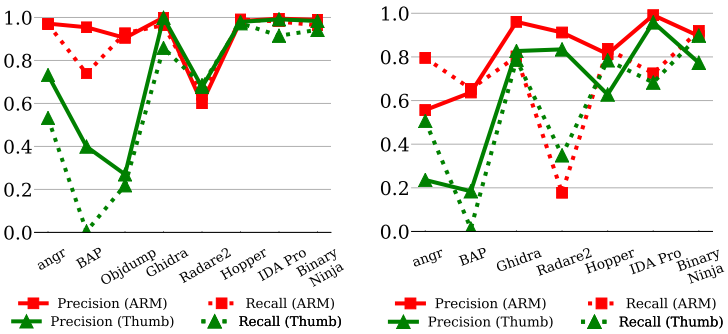
```
ARM:
00070078      bl          sub_9eaa0
(Ghidra thinks function 0x9eaa0 will return)
0007007c      ldr         r3, [sp, #0x78 + var_44]
00070080      tst         r3, #0x1
00070084      beq         loc_70184

Thumb:
000505bc      bl          sub_70668
(Ghidra thinks function 0x70668 is a non-return function)
(Ghidra thinks the offset from 0x505c0 is inline data)
000505c0      movs        r3, #0x1
000505c2      ldr         r2, [sp, #0x80 + var_44]
000505c4      tst         r3, r2
```

Figure 3.7: `Ghidra` (version 9.0.4 and version 9.1.2) performs differently when instruction set is different

`-mthumb`. By default, compilers use ARM instruction set. We call them ARM set binaries.

Figure 3.8 shows the evaluation result. The solid line and dotted line in the figure are used to denote the precision and recall, respectively. The x-axis shows the name of tools and the y-axis represents the average value of recall and precision for all the binaries. Note that, this format also applies to Figures 3.9, 3.10, 3.11, 3.12, 3.13 and 3.14.

First, disassembly tools perform worse for Thumb set binaries, i.e., they have lower precision and recall for the instruction boundary and the function boundary. Specifically, `BAP` has very low recall (0.40) and precision (0.01) for Thumb set binaries. We verified and reported our findings to developers of `BAP`. They acknowledged that `BAP` cannot handle Thumb binaries. Tools like `Objdump` cannot handle Thumb binaries either. This is because `Objdump` uses the ARM instruction set to linearly disassemble a binary without switching the instruction set. There are significant differences between the two instruction sets for tools like `angr` and `Ghidra`. This is because these tools have much better support of the ARM instruction set than the Thumb one.



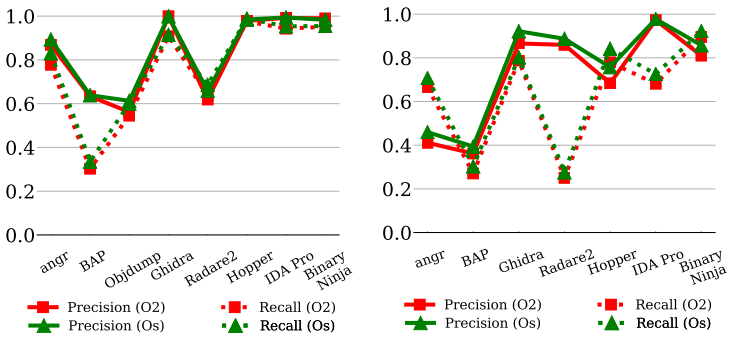
(a) Instruction boundary (b) Function boundary

Figure 3.8: The result of different instruction sets

Second, even for the binaries compiled from the same source code, the Thumb instruction set makes an inconsistency between the result. That is because the instruction set may have side effects on the recognized property of identified functions. Figure 3.7 shows such an example. The instructions at the offset `0x00070078` and `0x000505bc` are same (`BL`), which represent a function call. Both function calls refer to the same callee according to the source code. However, since the instruction set is different, `Ghidra` misinterprets that the callee function in the Thumb instruction set is a non-return function, thus it completely ignores the code after that offset (`0x000505bc`). Several similar cases are observed for the tool. This is the reason why the recall is relatively low for `Thumb` set binaries of `Ghidra`.

Third, the result of the function boundary correlates with the instruction boundary. That's because these two primitives have a strong connection with each other. If the instruction boundary cannot be recognized precisely, it will greatly affect the recognition of the function boundary, and vice versa.

Fourth, the result of the function boundary is worse for the Thumb instruction set. We suspect that is due to the reuse of the `BL label` instruction as both a function call and a direct branch for Thumb set binaries. Specifically, the `BL label` (`BLX label`) instructions are used to directly invoke a function. For the ARM instruc-



(a) Instruction boundary (b) Function boundary

Figure 3.9: The result of different optimization levels

tion set, compilers use instructions, e.g., `B label` for a direct branch. However, for the Thumb instruction set, the range of the `B label` is limited (±2KB for 16-bit Thumb) [10]. Compilers tend to reuse the `BL label` for a direct branch (range is ±4MB for 16-bit Thumb), which is same with a function call. This confuses the disassembly tools, which misinterpret direct branches as function calls. This raises high false positives to identify the function boundary and results in a low precision. Due to this, the proposed method to identify function boundary without relying on function signatures in Nucleus [72] is also ineffective, since it assumes the function call instruction could be identified. The initial result of applying this tool to binaries with the Thumb instruction set show that both the precision and recall are below 0.12.

Summary: The Thumb instruction set does bring serious challenges to disassembly tools.

Optimization Levels

As shown in the Section 3.3.1, optimization levels `O2` and `Os` are mostly used ones. They represent the optimization for performance and size, respectively. To evaluate the impact of optimization levels, we divide binaries into two categories. One contains

Table 3.4: F1 scores for different optimization flags with at least 95% probability value. NA:not applicable.

Tool	angr	BAP	Objdump	Ghidra	Radare2	Hopper	IDA Pro	Binary Ninja
Instruction	0	0	0	0.005	0.065	0.014	0.001	0.018
Function	0.033	0.046	NA	0.020	0.037	0.053	0.033	0.037

binaries compiled with the O2 flag, while the other one contains binaries compiled with the the Os flag.

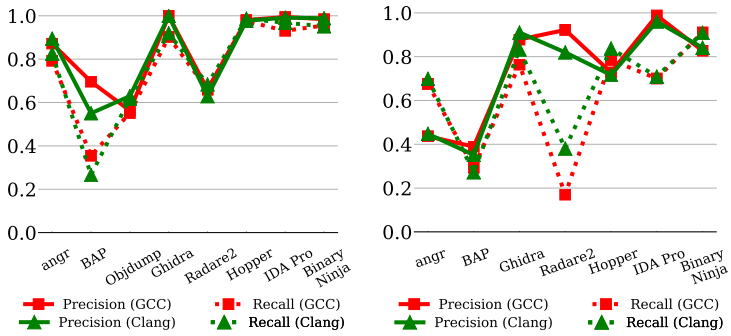
Figure 3.9 shows the result. Surprisingly, there is no significant differences between these two flags in terms of both recall and precision. To verify the conclusion that optimization level does not bring significant difference, we conducted an extra hypothesis test. We compute the F1 scores of the binaries in the two categories and compute the differences between every pair of binaries (i.e. one compiled with the flag O2 while the other one compiled with the flag Os). We then randomly picked 40 samples and conducted t-test on the samples. Table 3.4 shows the result. We noticed that different optimization levels do not bring significant differences on all the eight tools. The maximum differences in terms of F1 score is only 0.065.

We further explore the potential reason. It turns out that the Os flag enables all the optimization methods introduced in the O2 flag. Besides, it includes the ones to reduce binary size [14, 19], e.g., reducing the padding size and alignment. These ones have little impacts for the disassembly tool to identify the instruction and function boundary.

Summary: Optimization levels (O2 and Os) do not bring significant differences.

Compilers

GCC and Clang are two popular compilers. To evaluate the impact of compilers, we build binaries (the SPEC CPU2006) with both GCC and Clang. Figure 3.10 shows the evaluation result.



(a) Instruction boundary (b) Function boundary

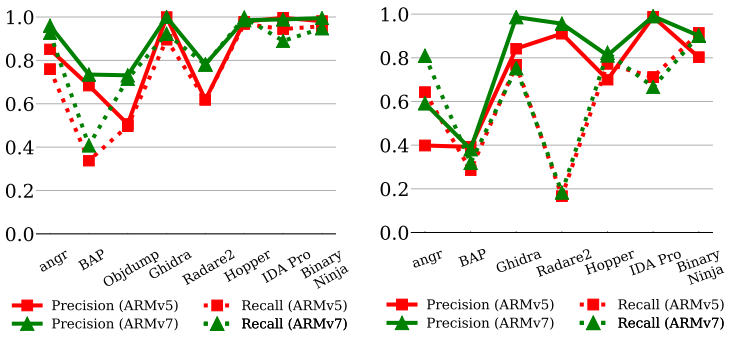
Figure 3.10: The result of different compilers

For the instruction boundary, most tools do not have obvious differences between binaries built with different compilers except `BAP`, which will be explained later. However, for the function boundary, `Radare2` and `BAP` are sensitive to binaries built with different compilers. For `Radare2`, the precision of the function boundary for binaries built with GCC is higher than the binaries built with Clang. `BAP` has a higher precision of the function boundary for binaries compiled with GCC. That is because `BAP` has a better collection of function signatures for binaries compiled with GCC than the ones compiled with Clang. Remember that, `BAP` does not disassemble the instructions that are not in the detected functions. Thus, the precision of the instruction boundary will also be higher for binaries compiled with GCC.

Summary: Compilers do not affect most of the tools, except `Radare2` and `BAP`, mainly due to the function identification method used by them.

CPU Architectures

ARM has multiple architectures, e.g., ARMv7 and ARMv5. Each architecture has different hardware features. For instance, the 16-bit Thumb instruction (Thumb-1) is available from ARMv4, while the 32-bit Thumb-2 instructions are available from ARMv6. Thus, if the binary is built for different architectures, instructions generated



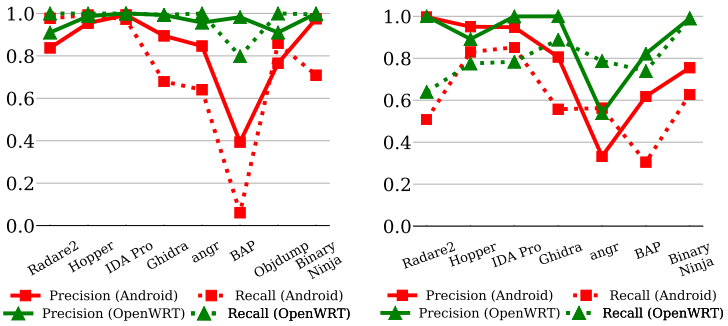
(a) Instruction boundary (b) Function boundary

Figure 3.11: The result of different CPU architectures

by the compilers will be different.

To evaluate the impact of binaries built with different CPU architectures, we use the binaries compiled for ARMv7 (`march=armv7-a`) and AVRMv5 (`march=armv5t`). Figure 3.11 shows the result. We find that disassembly tools perform better for binaries with the ARMv7 architecture, in terms of the precision of function boundary. This is because the Thumb-2 instructions are supported in the ARMv7 architecture, where the `B label` instruction has a much larger jump range (±16MB) than the original one (±2KB in the Thumb-1 instruction set) [10]. Compilers tend to use the `B label` instruction for the direct branch, instead of reusing the `BL label` instruction that is usually for the direct function call (Section 3.4.3). Thus, disassembly tools can distinguish the function call instruction with the direct branch instruction, and identify the function boundary more precisely.

Summary: For the ARMv7 CPU architecture, compilers use `B label` instruction for a direct branch, instead of reusing the `BL label` instruction. This helps the disassembly tools distinguish the direct branch instruction with the function call instruction, leading to a better precision value of identifying the function boundary.



(a) Instruction boundary (b) Function boundary

Figure 3.12: The result of system types

System Types

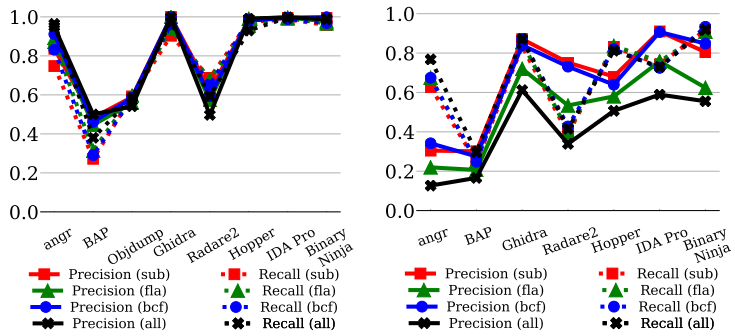
ARM binaries exist in different types of systems. In our work, we also evaluate the impact of different types of binaries. In particular, we use the binaries built from the OpenWRT [34] (Linux based embedded systems used for routers, IP cameras and etc.) and the Android open source project (AOSP version 9), respectively. The result is shown in Figure 3.12.

In general, the result for binaries of OpenWRT is better than the AOSP binaries. We further compared the binaries and found that most of the binaries (80%) in Android are compiled using the Thumb instruction set, while there are no binaries in OpenWRT compiled using the Thumb instruction set. As explained in previous sections, disassembly tools perform worse for Thumb binaries.

Summary: System types affect the result. This is due to the instruction set used in the binaries.

Obfuscation

To evaluate the impact of obfuscation to disassembly tools, we use O-LLVM [98], an open-source obfuscator, to compile the SPEC CPU2006. O-LLVM supports the following obfuscation methods. Specifically, instruction substitution (`sub`) is used to



(a) Instruction boundary (b) Function boundary

Figure 3.13: The result of different obfuscation methods

replace standard operators with more complicated sequences of instructions. The bogus control flow (`bcf`) changes a function call graph by adding basic blocks. The control flow flattening (`fla`) uses the control flow flatten algorithm [104] to create a large number of fake control flows.

We apply each obfuscation method to each program for building the binary, and then combine three methods together. We divide the obfuscated binaries into four groups. In the first three groups, each group contains the binaries that are obfuscated using one individual method. The last group contains the binaries that are obfuscated using all the three methods. The result is shown in Figure 3.13.

We observe that obfuscation does not affect the instruction boundary too much. However, the function boundary is greatly affected by the control flow flattening. This is because the control flow flattening generates a huge number of fake control flows. These fake control flows are using the `BL label` instructions in the Thumb binaries for direct branches. These instructions confuse the disassembly tools and introduce false positives to the function boundary (Section 3.4.3).

Summary: Obfuscation introduces challenges to the disassembly tools to locate the function boundary, especially the control flow flattening. The root cause is due to the reuse of `BL label` instruction for direct branches, which are inserted by the

obfuscation tool.

3.4.4 Types and Options of Tools (RQ3)

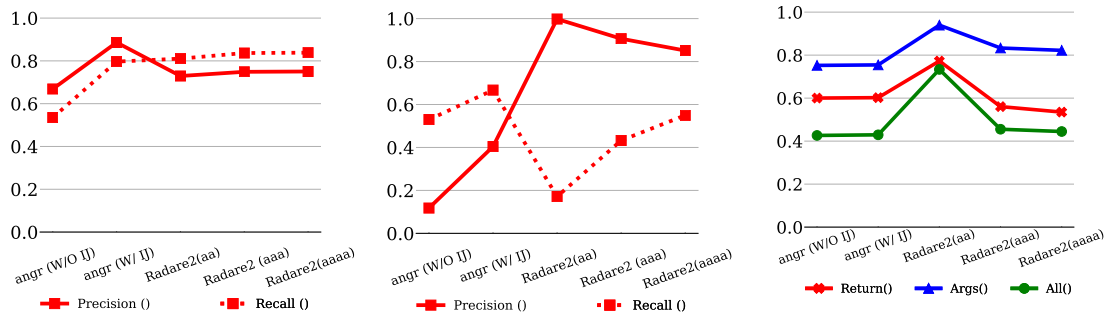
Commercial vs Noncommercial Tools

In our work, we use eight state-of-the-art tools. Among them, there are three commercial tools, i.e., `IDA Pro`, `Binary Ninja` and `Hopper`, and five noncommercial ones. We find that commercial tools have higher precision and recall. As shown in Table 3.3, for the instruction boundary, the three commercial ones are ranked as top three in terms of both precision and recall. For the function boundary, these commercial tools are performing better than other ones, except that `Radare2` has the better precision. For function signature, though `Radare2` has better recovery ratio compared with commercial tools, all the tools' recovery ratios are low. Moreover, the commercial tools are more stable and robust. They do not trigger any segment faults or exceptions during the analysis.

Summary: Compared with noncommercial ones, commercial tools are more accurate, robust, and stable.

Disassembly Tools' Options

Disassembly tools have different options, which can affect the result. We use `angr` and `Radare2` as examples since they provide explicit options that could be changed during the analysis. Figure 3.14 shows the result. Specifically, `angr` provides an option to enable or disable the indirect jump resolving. We observe that enabling the indirect jump resolving will increase the precision and recall for instruction and function boundary, since it can resolve more code sections that could only be reached through indirect branches. However, recovering function signatures mainly rely on the analysis of calling covention. Thus, the different options of `angr` has little impact on the function signature.

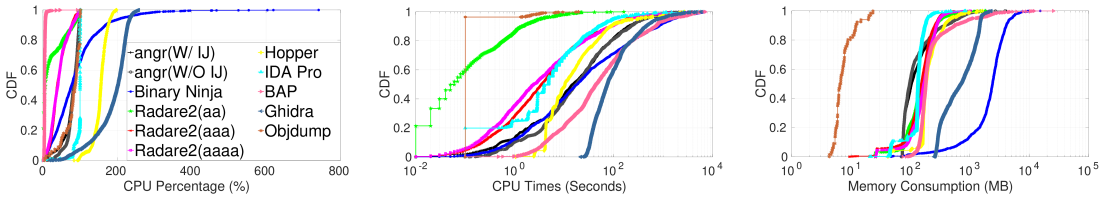


(a) Instruction boundary (b) Function boundary (c) Function Signature

Figure 3.14: The Result of tools' options. W/ and W/O IJ means the indirect jump resolving is enabled and disabled for angr. Return means the recovery ratio on evaluating whether a function return a value or not. Args means the recovery ratio on evaluating the number of the parameters of a function. All means both Return and Args are accurate.

As for `Radare2`, it provides three different options. They are `aa`,`aaa` (the default value) and `aaaa`. Option `aa` only analyzes the function symbols, while option `aaa` adopts more analysis methods, including function calls, type matching analysis, value pointers. Option `aaaa` uses the function prologues to locate more functions and performs constraint type analysis, besides the analysis included in the option `aaa`. We find that, complex analysis does not increase the accuracy of the instruction boundary, but has impacts on the function boundary and function signature. That is because the option `aa` only detects functions based on symbols, thus it misses most functions in the stripped binaries that do not have symbols but has a higher recovering ratio of function signature. Options `aaa` and `aaaa` adopt more analysis methods, e.g., function prologue analysis, that greatly improve the identification of function boundary.

Summary: Disassembly tools' options affect the result. For `angr`, enabling indirect jump resolving can improve the result, while `Radare2` has a better result for function boundary when using the option `aaaa`.



(a) CDF plot of CPU percentage

(b) CDF plot of CPU times

(c) CDF plot of memory consumption

Figure 3.15: The evaluation result of performance. The legend in the latter two figures are same with the first one.

3.4.5 Efficiency of the Disassembly Tools (RQ4)

Efficiency is an important feature of disassembly tools. Efficient tools can handle large binaries within a reasonable time. We report the efficiency of the tools. In particular, we calculate the CPU usage, CPU times and memory consumption during the analysis. Our experiments are done in Ubuntu 18.04 with 128GB memory size and 30 core intel(R) Xeon(R) Silver 4110 CPUs. Specifically, we use the Python library psutil [39] to extract the related information about resource consumption, and then use the function `cpu_times` to obtain the CPU times. We also use the function `cpu_percent (interval = 1)` to extract the CPU percentage. Note that this value can be bigger than 100% in case of a process running multiple threads. We use the function `memory_info` to obtain the memory consumption. The memory size is the Resident Set Size (rss), which is the non-swapped physical memory a process has used. The result is shown in Figure 3.15.

CPU Percentage. `Binary Ninja` consumes lots of CPU resource and can reach to nearly 800% for some binaries. `Ghidra` ranks the second. Half of the binaries would consume 200% of the CPU usage. `BAP` consumes the least CPU percentage. However, according to our observation, `BAP` spends a lot of time to analyze the binaries due to the inefficient usage of CPU.

CPU Times. Among all the tools, `Ghidra` consumes most CPU times compared with the other tools in nearly 80% binaries. There are no significant performance differences for `angr` with and without indirect jump resolving. Since the indirect jump resolving improves the result, we recommend users to enable this option during analysis. For `Radare2`, the option `aaa` needs much more CPU times compared with the option `aa`. However, the precision and recall of the instruction boundary do not have a significance improvement.

Memory Consumption. Among all the tools, `Binary Ninja` consumes most of the memory (nearly 10 GB in maximum), while `Objdump` consumes the least. `IDA Pro` is quite stable for the memory usage. It consumes only around 100MB memory for nearly 70% of the binaries.

3.4.6 Improvement (RQ5)

To understand the improvement of these tools overtime, we collect two different versions for each tool. One is the newest version at the time of writing (new version). The other one (old version) is the version evaluated in [92]. As shown in Table 3.3, the period between the release dates of the two version ranges from 272 days to 735 days, which is long enough for developers to improve the tools.

Improvement on Accuracy

Commercial Tools. By comparing the results, we noticed that the commercial tools do not have significant improvement. For example, `IDA Pro` and `Hopper` remains nearly the same for the F1 score of instruction boundary, function boundary, and function signature identification. For `Binary Ninja`, the F1 score of instruction boundary increases little (i.e., 0.008) while the F1 score of function boundary increase by 0.027. Meanwhile, the function signature increase by 0.058. This may also because the period for the two versions of `Binary Ninja` is quiet long (i.e., 643

days) and the accuracy on particular binaries is increased. For instance, program `447.dealII` compiled by Clang (version 8) with the optimization level `Os` and ARM instruction set has only 0.324 F1 score for instruction boundary in `Binary Ninja` (version 1.1.1470). Due to the imprecise instruction boundary identification, the F1 score for function boundary is rather low (i.e., 0.056). This value increased to 0.901 and 0.778 for instruction and function boundary, respectively in `Binary Ninja` (version 2.2.2487).

Noncommercial Tools. Noncommercial tools gain larger improvement compared with commercial tools. We suspect this is due to two reasons. One is that the noncommercial tools perform worse in the old version, which leaves a larger improvement space. For instance, more than 600 binaries triggered either an exception or a segment fault of `angr` (version 8.19.4.5). We submit the failed test cases to the developer of `angr`, only 305 binaries cannot be analyzed in `angr` (version 9.0.4663) and the F1 score of instruction boundary increased by 0.125. We also noticed that the `angr`'s recovery ratio of function signatures increase most (i.e., 0.368-0.246=0.122) among all the tools. The other reason is that noncommercial tools receives more tests, feedback, and suggestions from the communities. Since they are open-source, communities can submit issue, pull request, and suggestion to the developers directly to speed up the developing process, fix the issues, and increase the accuracy. For instance, `angr` receives more than 1000 github issues and pull requests while `Radare2` receives more than 7000 github issues and 10000 pull requests till January 2021.

Though noncommercial tools gain larger improvement compared with commercial tools. They are still having problems on accurately identifying the instruction boundary, function boundary, and function signature. The failed cases mentioned in Fig. 3.4 and Fig. 3.6 also applies to new versions of the tools. This indicates that accurately disassembling the binaries is still challenging.

Table 3.5: The performance statistics for the improvement.

Tool	Options	CPU Times	CPU Percentage	Memory Consumption
BAP	Default	178.10	-42.92	-30.78
Objdump	Default	-0.14	-0.07	-0.64
Ghidra	Default	68.00	49.82	257.70
Radare2	aa	0.66	4.76	-6.08
	aaa	25.52	2.94	4.87
	aaaa	25.54	2.99	6.00
angr	enable indirect jump	-31.34	0.80	-3.71
	disable indirect jump	-43.26	-0.90	-8.31
Binary Ninja	Default	41.35	6.59	138.62
Hopper	Default	-7.97	-6.25	-7.12
IDA Pro	Default	9.88	14.40	10.29

Summary: Noncommercial tools gain larger improvement compared with commercial tools due to two reasons. One is that noncommercial tools have lower recall and precision value in the old version. The other is that noncommercial tools can receive more test, feedback, and suggestions from communities.

Improvement on Efficiency

Apart from the accuracy, we also evaluate the improvement on efficiency. Table 3.5 shows the overall result. We monitor the efficiency statistics (i.e., CPU percentage, CPU times, and memory consumption) for the tools in old versions and new versions and calculate the average value. We calculate the improvement for each metrics by using the statistics in old versions to minus statistics in new versions. Thus, if the improvement is positive, it means the value of the metrics is reduced and vice versa.

Commercial Tools. For commercial tools, we noticed that `IDA Pro` and `Binary Ninja`'s performance improves a lot. Specifically, the CPU Times of `Binary Ninja` reduces 41.35 seconds and 138.62 MB memory is saved in average. `Hopper`'s performance decrease a little. We suspect this is because of the added new features.

Noncommercial Tools. For noncommercial tools, the CPU times, which indicates how much time are needed for a tool, are reduced for all the tools except `angr` and

`Objdump`. `Objdump`'s CPU times just decrease by 0.14, which can be ignored. For `angr`, we think this is due to the added algorithms (e.g., identification of tail call optimizations) and features to increase the accuracy and robustness of disassembly tasks. As for CPU percentage, `BAP`'s CPU percentage increased a lot. This is because that `BAP` in old version get stuck while analyzing most Thumb binaries, resulting in a low CPU usage. In the new version of `BAP`, the stuck cases are reduced, resulting in a higher CPU usage. We also noticed that several tools' memory consumption are increased(30.78 MB in maximum), which is acceptable.

Summary: Most of the tools improve a lot for performance while `Hopper` and `angr`'s performance decrease due to the added features and algorithms.

3.5 Implications

In this section, we discuss implications based on the evaluation result and point out possible improvements.

ARM-specific disassembly strategies. First, inline data is popular in ARM binaries. Previous research shows that there is few inline data in x86/x64 binaries and the jump tables are located in the `.rodata` section. However, inline data is very common in ARM binaries, which increases the difficulty to locate instruction boundaries. Second, there are two instruction sets, i.e., ARM and Thumb instruction sets. Detecting the right instruction set is challenging for disassembly tools. Furthermore, we noticed that most tools do not have good support on the Thumb instruction set, either with a wrongly detected instruction set or a thrown exception. For instance, `angr` throws exceptions and gets segment faults for several binaries with the Thumb instruction set. `Objdump` can merely identify the Thumb instruction set. Given the fact that the Thumb instruction set is popular, especially in the binaries for mobile systems, there is an urgent need to propose effective solutions. Third, since most

existing works are focusing on x86 and x64 [52, 55, 56, 72, 143, 144], some ARM specific mechanisms should be proposed to deal with the instruction set switching. For instance, the hybrid disassembly technique [159] could be leveraged to locate the inline data and distinguish between different instruction sets, with customizations to adapt to the ARM architecture. Besides this, disassembly tools could perform a further check on its disassembly result. In other words, they could conduct a conflict analysis to improve the result. For example, `Radare2` explicitly knows when there is an invalid instruction. In this case, it can either switch the mode, or further check whether the invalid code is actually inline data through a data reference analysis.

Mechanisms to identify the function boundary. Our result shows that there is still a large space to improve the effectiveness of detecting the function boundary. Tools usually use function signatures to identify functions. These signatures could be generated through a machine learning based method. However, the machine-learning based methods could be limited due to the incompleteness of the training data sets [72]. For instance, the machine-learning based method in `BAP` performs worse than most of the other tools in detecting function boundaries. Furthermore, the mechanisms that work well on x86/x64 [72] cannot be applied to ARM, because ARM does not have a distinguished function call instruction, which is required by the method. According to our evaluation, besides function call, `BL label` is widely used in the Thumb instruction set for direct branch. Disassembly tools cannot distinguish the usage of `BL label` as direct branch with direct function call, resulting in a low precision in terms of the function boundary.

We think a more effective algorithm to detect the function boundary is needed. For example, developers could use the machine-learning based mechanism to detect the function first, and then conduct a static analysis by considering the internal logic between different basic blocks to reduce the false positives and false negatives. More-

over, disassembly tools can further analyze the `BL label` instruction to understand whether it's a function call. We think a further analysis of the usage of the `BL label` instruction can greatly improve the result of the function boundary.

Usability. Tools have different user interfaces and plugin infrastructures. For instance, `IDA Pro`, `Hopper` and `Binary Ninja` have user-friendly GUI interfaces, and provide easy-to-use Python APIs. `angr` itself does not provide GUI, and is invoked purely through a Python script. `BAP` has a good flexible architecture for extension. However, the supported language Ocaml has a steep learning curve, compared to the Python programming language. As for `Radare2`, it is completely different from the other tools. It just loads the binary and provides an interactive shell. Users have to leverage the shell to perform the analysis. There are many different kinds of built-in analysis phases.

We also observe that non-commercial tools suffer from the scalability and stability. For instance, `BAP` cannot finish the analysis on several binaries while `angr` will raise exceptions or get segment faults on several binaries. Furthermore, tools may have different options, which impact the usage of system resources. Users should pick the right options according to the purpose. For example, if users use the `Radare2` to disassembly the instruction (and do not care about the function boundary), they can use the option `aa`, which satisfies the need and is much faster than other options.

Improvements. State-of-the-art disassembly tools are maintained well and updated periodically. We compare the eight disassembly tools in two different versions in terms of both accuracy and performance. The release dates for the two different versions range from 272 days to 735 days, which is long enough to evaluate the improvements.

We find that noncommercial tools gain larger improvements in terms of accuracy compared with the commercial tools. This is because that noncommercial tools

perform worse in old version and noncommercial tools receives more test and feedback from the communities. As for performance, we find most of the tools improve a lot while some tools' performance may decrease due to the added feature and algorithm.

3.6 Discussion

First, with the introduction of the ARMv8 architecture, there exist 64-bit ARM binaries, which are missed in this work. However, 32-bit binaries are still the most popular ones. Due to the compatibility concern, new ARM architectures maintain backward compatibility with old ones. Our findings can still be applied to ARMv8 (ARMv8 supports both AArch64 for 64-bit binaries and AArch32 for 32-bit binaries) and future versions of ARM as long as ARM does not deprecate 32-bit ARM instruction set. Besides, AArch64 simplifies the task of disassembly tools. This is because 32-bit ARM has both 16-bit and 32-bit instructions and much more diverse branch instructions. As shown in our evaluation, the switching between instruction sets brings serious challenges to disassembly tools.

Second, we only evaluate eight state-of-the-art disassembly tools. However, there exist some disassemblers that are either research prototypes or not actively maintained. They are excluded from our work. Moreover, we only evaluate two fundamental disassembly primitives. We think other primitives such as direct control flow graph or direct call graph are easy to be generated if the instruction boundary and function boundary are located correctly[1].

Third, the generation of the ground truth is an essential step. Fortunately, the ARM ELF format introduces mapping symbols that can help to distinguish between different instruction sets, and between code and inline data. By leveraging this information, we can generate a complete and sound result for the instruction boundary.

[1] We understand that the indirect control flow transfer is still a challenging task.

At the same time, we could use the DWARF debugging information to extract the ground truth of the function boundary. For other primitives like control flow graph and call graph, they consist of direct jumps and indirect jumps. Direct jumps can be built based on the precise instruction boundary and function boundary, which is the reason why we do not include them in the evaluation. For the evaluation of indirect jumps, we cannot get a sound and complete ground truth even if we have the source code. This is because some jump targets can only be determined at running time. We leave the evaluation of these primitives as one of the further works.

3.7 Summary

We conduct the first comprehensive study on the capability of eight ARM disassembly tools in different versions to locate instruction boundary, function boundary, and function signatures, using diverse ARM binaries built with different compiling options and compilers. We report our new findings, which shed light on the limitations of the state-of-the-art disassembly tools, and point out potential directions for improvements.

Chapter 4

Examiner: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM

4.1 Overview

In this chapter, we aim to automatically locate inconsistent instructions between real devices and the CPU emulator for the ARM architecture. If an instruction behaves differently between them, then it is an inconsistent instruction. Although previous research [110, 111, 113, 112] provides valuable insights, they are limited to the x86/x64 architecture and cannot be directly applied to the ARM architecture. Our work leverages the differential testing [114] for the purpose. Specifically, we provide the same instruction stream [1] to both the real device and a CPU emulator, and compare the execution result to check whether it is an inconsistent one.

Though the basic idea is straightforward, it faces the following two challenges. *First*, the ARM architecture has multiple versions (e.g., ARM v5, v6, v7 and v8), different register widths (16 bits or 32 bits) and instruction sets. Besides, it has

[1] In this chapter, instruction and instruction stream represent different meanings. For example , we call STR (immediate) an instruction. We call the concrete bytecode (i.e., 0xf84f0ddd) an instruction stream. See Section 4.2.1

mixed instruction modes (ARM, Thumb-1 and Thumb-2). Thus, how to generate effective test cases, i.e. instruction streams that cover previously mentioned architecture variants, while at the same time generating only necessary test cases to save the time cost, is the first challenge. Notice that if we naively enumerate 32-bit instruction streams, the number of test cases would be 2^{32}, which is inefficient, if not possible, to be evaluated. Meanwhile, randomly generated instruction streams are not representative and many instructions are not covered (Section 4.4.1). *Second*, for each test case, we should provide a deterministic environment to execute the single instruction stream and automatically compare the result after the execution. This requires us to set up the same context (with CPU registers and memory regions) before the execution and compare the context afterwards.

Our system solves the challenges with the following two key techniques.

Syntax and semantics aware test case generation. To generate representative instruction streams, we propose a syntax and semantics aware test case generation methodology. Each ARM instruction consists of several *encoding schemas*, which is called *instruction encodings*, that define the instruction's structure (syntax). Each encoding schema maps to one decoding and execution logic that defines the instruction semantics. The encoding schema shows which parts of an instruction are constants and which parts can be mutated (Figure 4.2(a)). The non-constant parts of an instruction are called *encoding symbols*. The decoding and execution logic is expressed in the ARM's Architecture Specific Language (ASL) [127] . We call it the *ASL code* in this thesis (Figure 4.2(b) and (c)). The ASL code executes based on the concrete values of the encoding symbols. For instance, if the concrete value of the encoding symbol `W` (the eighth bit of `STR (immediate)` instruction) is `1`, then the new address will be written back into the destination register `Rn` (line 4 of Figure 4.2(c)).

Specifically, during the test case generation, we first take the syntax-aware strategy. For each encoding symbol, we mutate it based on pre-defined rules. For instance,

for the immediate value symbol, the values in the mutation set cover the maximum value, the minimum value and a fixed number of random values. This strategy generates syntactically correct instructions.

We further take a semantics-aware strategy to generate more instruction streams. That's because the previous strategy may only cover limited instruction semantics as different encoding symbol values can result in different decoding and executing behaviors (Section 4.3.1). To this end, we extract the constraints in ASL code of decoding and executing. We solve the constraints and their negations by designing and implementing the first symbolic execution engine for ASL to find the satisfied values of the encoding symbols. By doing so, the generated test cases can cover different semantics of an instruction.

Deterministic differential testing engine. Our differential testing engine uses the generated test cases as inputs. To get a deterministic testing result, we provide the same context when executing an instruction stream on a real CPU and an emulator. Besides, an instruction stream cannot be directly loaded and executed by the emulator, we carefully design a template binary that converts one instruction stream to a testing binary by inserting the prologue and epilogue instructions. The prologue instructions aim to set the execution environment while the epilogue instructions will dump the execution result for comparison to check whether the testing instruction stream is an inconsistent one.

We have implemented a prototype system called EXAMINER. Our test case generator generated $2,774,649$ instruction streams that cover all the $1,998$ ARM instruction encodings from $1,070$ instructions in four instruction sets (i.e., A64, A32, T32, and T16). On the contrary, the same number of randomly generated instruction streams can only cover 51.4% Instructions. This result shows the sufficiency of our test case generator. We then feed these test cases into our differential testing engine. By comparing the result between the state-of-the-art emulator (i.e., QEMU) and real

devices with four architecture versions (ARMv5, ARMv6, ARMv7-a, and ARMv8-a), our system detected 155,642 inconsistent instruction streams. Furthermore, these inconsistent instruction streams cover 47.8% of the instructions. We then explore the root causes of them. It turns out that implementation bugs of QEMU and the undefined implementation in the ARM manual (i.e., the instruction does not have a well-defined behavior) are the major causes. We discovered four implementation bugs of QEMU and all of them have been confirmed by developers. These bugs influence 13 instruction encodings, including commonly used instructions, e.g., `BLX`, `STR`.

To show the usage of our findings, we further build three applications, i.e., emulator detection, anti-emulation and anti-fuzzing. By (ab)using inconsistent instructions, a program can successfully detect the existence of the CPU emulator and prevent the malicious behavior from being monitored by the dynamic analysis framework based on QEMU. Besides, the coverage of the program being fuzzed inside an emulator can be highly decreased. Note that, we only use these applications to demonstrate the usage scenarios of our findings. There may exist other applications, and we do not claim the contribution of them in this thesis.

Our work makes the following main contributions.

New test case generator. We propose a test case generator by introducing the first symbolic execution engine for ARM ASL code. It can generate representative instruction streams that sufficiently cover different instructions (encodings) and semantics.

New prototype system. We implement a prototype system named EXAMINER that consists of a test case generator and a differential testing engine. Our experiments showed EXAMINER can automatically locate inconsistent instructions.

New findings. We explore and report the root cause of the inconsistent instructions. Implementation bugs of QEMU and undefined implementation in ARM man-

ual are the major causes. Furthermore, four bugs have been discovered and confirmed by QEMU developers. Some of them influence commonly used instructions (e.g., `STR`, `BLX`).

We will release generated test cases and the source code of our system to engage the community.

4.2 Background

4.2.1 Terms

For better illustration and avoid the potential confusion. We give detailed definition towards the following terms used in this chapter.

Instruction. Instruction denotes the category of ARM instructions in terms of functionality, which is usually represented by its name in ARM manual. For example, `STR (immediate)` is an instruction, which aims to store a word from a register to memory.

Instruction Encoding. Instruction encoding refers to the encoding schemas for each instruction. We also call it encoding diagram. One instruction can have several encoding schemas.

Instruction Stream. Instruction stream refers to the bytecode of an instruction. For example, 0xf84f0ddd, which meets one of the encoding schema of instruction `STR (immediate)`. We call 0xf84f0ddd an instruction stream.

4.2.2 ARM Instruction and Instruction Encoding

Processor specification is important as it can verify the implementation of hardware, compilers, emulators, etc. To formalize the specification, ARM introduced the Architecture specification language (ASL) [127], which is machine-readable and executable.

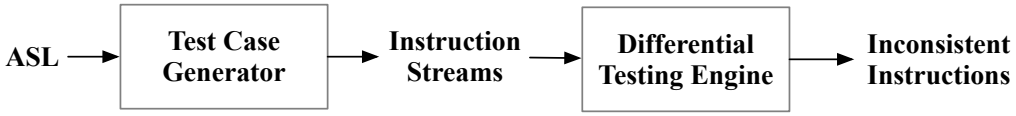
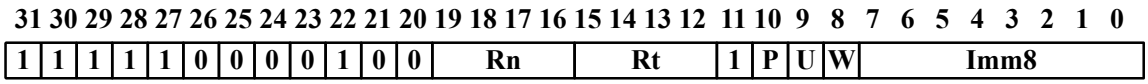


Figure 4.1: The work flow of our system

ARM instructions usually have a fixed length (16 bits or 32 bits). According to ARM manual, one instruction may consist of several different instruction encodings, which describe the instruction structure (syntax). Our system generates the instruction streams that cover all the instruction encodings (which cover all instructions.) Specifically, the instruction encoding describes which parts of the instruction are constant and which parts are not. Each instruction encoding is further described with specific decoding and executing logic. The decoding and executing logic (expressed in ASL) defines the semantics of the instruction.

4.2.3 Instruction Decoding in QEMU

QEMU is the state-of-the-art CPU emulator that supports multiple CPU architectures. When executing an instruction stream, it needs to decode the instruction stream. QEMU adopts a two-stage decoding schema. In the first stage, it matches an instruction stream with pre-defined patterns, each of them represent multiple instructions. Then it distinguishes each instruction encoding based on the concrete value of the instruction. For instance, QEMU groups `VLD4`, `VLD3`, `VLD2`, and `VLD1` instruction into one group (with one common pattern) and then identifies them inside the instruction decoding routine. If no instruction pattern can be found or further decoding routine cannot recognize an instruction stream, the `SIGILL` signal will be raised for the user mode emulation of QEMU.



(a) The encoding schema of the STR (immediate) instruction in Thumb-2 mode.

```
1  if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
2  t = UInt(Rt);
3  n = UInt(Rn);
4  imm32 = ZeroExtend(imm8, 32);
5  index = (P == '1');
6  add = (U == '1');
7  wback = (W == '1');
8  if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

(b) The ASL code for decoding the instruction.

```
1  offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
2  address = if index then offset_addr else R[n];
3  MemU[address,4] = R[t];
4  if wback then R[n] = offset_addr;
```

(c) The ASL code for executing the instruction.

Figure 4.2: A motivating example.

4.3 Design and Implementation

Figure 4.1 shows the workflow of EXAMINER, which consists of a test case generator and a differential testing engine. First, the test case generator retrieves the ASL code to generate the test cases (Section 4.3.2). Then, the differential testing engine receives the generated test cases and conducts differential testing between the emulators and real devices (Section 4.3.3). The instructions leading to different execution results are located as inconsistent instructions. We further analyze the identified inconsistent instructions to understand the root cause of them and how they can be (ab)used.

In the following, we first use an inconsistent instruction detected by our system as a motivation example (Section 4.3.1), and then elaborate the test case generator and the differential testing engine in Section 4.3.2 and Section 4.3.3, respectively.

4.3.1 A Motivating Example

The encoding schema and semantics of the STR (immediate) instruction

Figure 4.2 shows one of the encoding schema of instruction `STR (immediate)` and the corresponding ASL code for decoding and execution logic. According to the encoding schema in Figure 4.2a, the value is constant (i.e., `111110000100`) for offset [31:20]. The encoding symbol `Rn` and `Rt` represent the addressing register and the source register, respectively. The last eight bits ([7:0]) represents a symbol value named `imm8` that will be used as the offset.

Figure 4.2b shows the ASL code of the decoding logic for the encoding schema. Note that the ASL code is simplified for presentation. The complete code can be found in ARM official site [5].

- The ASL code at Line 1 checks the value of `Rn`, `P`, and `W`. If the conditions are satisfied (or constraints are met), the instruction stream will be treated as an `UNDEFINED` one. Consequently, a `SIGILL` signal will be raised in QEMU user mode emulation when an `UNDEFINED` instruction stream is executed.

- In line 2 and 3, the symbol `Rt` and `Rn` will be converted to unsigned integer `t` and `n`, respectively. Similarly, the symbol `imm8` will be extended into a 32-bit integer `imm32`. In line 5, 6, and 7, symbol `index`, `add`, and `wback` will be assigned according to the value of `P`, `U`, and `W`, respectively.

- In line 8, the symbol `t`, `wback`, and `n` will be checked. If the constraint of each condition is met, the instruction stream should be treated as an `UNPREDICTABLE` one. According to ARM's manual, the behavior of an `UNPREDICTABLE` instruction stream is not defined. The CPU processor vendors and the emulator developers can choose an implementation that they think it's proper.

Similarly, Figure 4.2c shows the ASL code for the execution logic of the instruction. The ASL code in Figure 4.2b and Figure 4.2c defines the semantics of the

instruction.

Test case generation

By analyzing the encoding schema, EXAMINER generates the test cases by mutating the non-constant fields, including `Rn`, `Rt`, `P`, `U`, `W` and `Imm8`. This can generate syntactically correct instructions. However, this step is not enough, since it may not generate the values that satisfy the symbolic expression in the ASL code. For instance, one symbolic expression in line 8 of Figure 4.2b is `t == 15`. The random values generated in the first step may not satisfy this expression (all of them are not equal to 15). To this end, we leverage a constraint solver to find the concrete value of the encoding symbol `Rt` that satisfies the constraint, i.e., `15`. Note that, we only use this to illustrate the basic idea. The concrete value `15` of `Rt` likely has been generated in the first step. We take similar actions to solve the constraints for other symbols in line 1 (`add`), 2 (`index`) and 4 (`wback`) of Figure 4.2c. During this process, we generated 576 instruction streams as test cases in total.

Differential testing

We feed each instruction stream into our differential testing engine. The engine generates a corresponding ELF binary for each test case by adding prologue and epilogue instructions. The prologue instructions first set the initial execution context, then the instruction stream will be executed. Finally, the epilogue instructions will dump the result for comparison. We execute the binary on both QEMU and real devices (e.g., RasberryPi 2B). By comparing the execution result, we confirm that `0xf84f0ddd` is an inconsistent instruction stream. Specifically, It will generate a `SIGILL` signal in a real device while a `SIGSEGV` signal in QEMU.

We further analyzed the root cause and successfully disclosed a bug in QEMU. According to Figure 4.2a, the concrete value of the encoding symbol `Rn` of the instruc-

```
1  static bool op_store_ri(DisasContext *s, arg_ldst_ri *a, MemOp mop, int mem_idx)
2  {
3      ISSInfo issinfo = make_issinfo(s, a->rt, a->p, a->w) | ISSIsWrite;
4      TCGv_i32 addr, tmp;
5
6      // Rn=1111 is UNDEFINED for Thumb;
7
8  +   if (s->thumb && a->rn == 15) {
9  +       return false;
10 +   }
11
12     addr = op_addr_ri_pre(s, a);
13
14     tmp = load_reg(s, a->rt);
15     gen_aa32_st_i32(s, args);
16     disas_set_da_iss(s, mop, issinfo);
17     tcg_temp_free_i32(tmp);
18     op_addr_ri_post(s, a, addr, 0);
19     return true;
20 }
```

Figure 4.3: Original code of QEMU and the patch for function op_store_ri, which aims to translate STR instruction

tion stream `0xf84f0ddd` is `1111`. As shown in the ASL code (line 1) in Figure 4.2b. it is an `UNDEFINED` instruction stream. However, QEMU does not properly check this condition. Figure 4.3 shows the (patched) function (i.e., `op_store_ri`) in QEMU for decoding the instruction `STR` (immediate). It continues the decoding process directly from line 12 without any check. We then submit this bug to QEMU developers and the patch is issued (as shown in line 8-10).

4.3.2 Test Case Generator

In theory, for a 32-bit instruction, there exist $2^{32} = 4,294,967,296$ possible instruction streams, which are not practical for evaluation. In our work, we need to generate a small number of representative test cases that cover most behaviors of an instruction.

Specifically, we first parse the encoding schema to retrieve the encoding symbols and then infer the type for symbols, e.g., a register index or an immediate value. After that, we generate an initialized mutation set by pre-defined rules for each type of the symbol (Table 4.1 shows the detailed rules). For instance, we generate the maximum, minimum and random values for an immediate value. Then, we develop a symbolic execution engine to solve the constraints in the ASL code for the decoding

and execution logic. This step can add more values to the mutation set to satisfy the constraints of the symbols in the ASL program. At last, we remove duplicate values and then generate instruction streams as test cases.

Algorithm 1 shows how we generate the test cases. For each instruction, ARM provides a XML file to describe the instruction. We extract the encoding schemas and the corresponding ASL code for decoding and execution by parsing the XML file. We first retrieve the encoding symbols ($Symbols$) and constant values ($Constants$) in the encoding schema, as well as $Constraints$ for the symbolic expression in decoding and execution ASL code (line 2). We then iterate over the $Symbols$ and generate the $MutationSet$ for each symbol (line 3-4), which will be introduced in detail in Section 19. Note this is the initial mutation set for each symbol. For the $Constants$, the $MutationSet$ contains only the fixed value (line 5-6). After that, we solve the constraints to generate new mutation set (i.e., $ValueSet$) for each symbol (line 7-8), which will be introduced in detail in Section 19. Then we check whether the solved value for each symbol is in the symbol's $MutationSet$ (line 9). If not, we append it to the symbols's $MutationSet$ (line 10-11). After that, we combine them to get the $MutationSets$ (line 12).

Finally, considering all the possible combinations of the candidates in the $MutationSet$ for each symbol, we conduct the Cartesian Product on the $MutationSets$ to get the test cases for this specific instruction encoding (line 13).

Initialize Mutation Set

In the phase of initializing mutation set for each symbol, we consider the types of different symbols and aim to cover different values for different types of symbols. In particular, we infer the type based on the symbol name. For instance, a symbol that represents a register index usually has the name `Rd`, `Rm`, `Rn`, etc. As for the immediate value, the symbol name used to be `immn` where `n` represents the length of the value.

Algorithm 1: The algorithm to generate test cases.

Input: The encoding diagram: I_Encode;
The decoding ASL code: I_Decode;
The execution ASL code: $I\_Execute$
Output: The generated test cases: T;

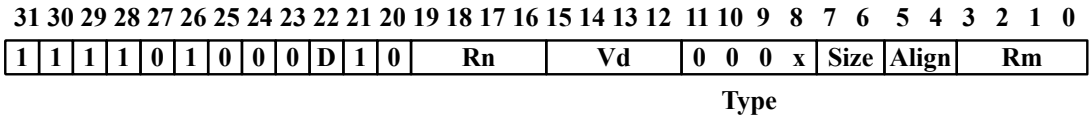
```
Function Generate(I_Encode,I_Decode,I_Execute):
    Symbols, Constants, Constraints = ParseASL(I_Encode, I_Decode, I_Execute)
    for S in Symbols do
        S.MutationSet = InitSet(S)
    end
    for C in Constants do
        C.MutationSet = [ConstantValue]
    end
    for C in Constraints do
        ValueSet = SolveConstraint(C, Symbols, I_Decode, I_Execute)
        for V, S in ValueSet do
            if V not in S.MutationSet then
                S.MutationSet add V
            end
        end
    end
    MutationSets = [S.MutationSet +C.MutationSet]
    TestCase = CartesianProduct(MutationSets)
    return T
```

Table 4.1: The rules of initializing the mutation set.

Type of Symbol Name	Mutation Set
Register Index	0 (R0); 1 (R1); 15 (PC); Random index values
Immediate Value in N bits	Maximum value: 2^N -1; Minimum value: 0; (N-2) Random Value from the enumerated values
Condition	"1110" (Always execute)
Others in 1 bit	"0"; "1"
Others in N bit (N >1)	N random value from the enumerated values

For example, the symbol `imm8` represents a 8-bit immediate value.

Table 4.1 shows the rules to initialize the mutation set. For a register index, we include the `PC` register (index 15), `R0`, `R1` and random values in the set. The register `R0` and `R1` are used to represent the return value for function calls. As for PC, it can explicitly change the execution flow of the program. Thus, the register index in many instruction encodings cannot be 15. We include it in the mutation set to cover such cases. For the immediate value, the maximum and minimum value are the two



(a) Encoding diagram of instruction VLD4 in A32 instruction set

```
case type of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);
elements = 8 DIV ebytes;
d = UInt(D:Vd);
d2 = d + inc;
d3 = d2 + inc;
d4 = d3 + inc;
n = UInt(Rn);
m = UInt(Rm);
wback = (m != 15);
register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

(b) Decoding code of instruction VLD4 in A32 instruction set

Figure 4.4: Test case generator example.

boundary values that need to be covered. Apart from this, we randomly select (N-2) values, where N represents the bit length of the symbol. Note that enumerating all the values for one symbol is not realistic because immediate values have 24 bits, resulting in $2^{24} = 16777216$ candidates.

Solve Constraints

Symbolic expressions in ASL code represent the different execution paths of the instruction. For instance, `d4` in Figure 4.4 is a symbolic expression ($d4 = UInt(D : Vd) + inc + inc + inc$) that determines whether the instruction is an `UNPREDICTABLE` one. To make our test case representative, the generated test cases should cover as many execution paths as possible. To this end, we design and implement a symbolic execution engine for the ASL code. Specifically, we assign symbolic values for encoding symbols. Then we generate the symbolic expression for each variable in the ASL code. After that, we retrieve the constraint of the symbolic expression and find the

concrete values of the encoding symbols that satisfy the constraint and its negation, e.g., solve the constraints $(d4 > 31) == true$ and $(d4 > 31) == false$.

Figure 4.4 shows a concrete example. In line 18, there is a symbolic expression `d4` and a constraint $d4 > 31$. All the related statements (line 3, 5, 10, 11, 12, and 13) are retrieved via backward slicing and highlighted in the green color. To solve this constraint, we conduct backward symbolic execution. Specifically, the symbol `d4` is calculated by the expression $d4 = d3 + inc$ in line 13. Thus, the constraint is converted to $d3 + inc > 31$. Given the relationship between `d3` and `d2` in line 11, and between `d2` and `d1` in line 11, we further convert it to $UInt(D : Vd) + 3 \times inc > 31$. The expression `UInt(D:Vd)` is converted to $Vd + 2^4 \times D$ as the symbol `Vd` has 4 bits. Thus, we have the constraint $Vd + 16 \times D + 3 \times inc > 31$. Symbol inc is assigned at line 3 and line 5. Thus, the constraint is $inc == 1$ *or* $inc == 2$. Apart from this, we need to consider the length of each symbol. Since D is one bit and Vd has four bits. Their constraints are $D \geq 0$ *and* $D < 2$, $Vd \geq 0$ `and` $Vd < 16$.

We feed all these constraints to the SMT solver. It returns a solution which is a combination of symbol values that satisfy the constraints. One possible solution is that Vd is 13, D is 1, and inc is 2. We then negate the constraint $d4 > 31$ and repeat the above mentioned process. In this case, the solution is Vd is 0, D is 0 and inc is 1. Thus, the generated $ValueSet$ contains three symbols and each symbol has two candidate values. Note inc's value depends on $Type$'s value. As we will also solve the constraint $Type ==$ '0000' and $Type ==$ '0001', the final mutation set of $Type$ must contain the value that can make inc to be either 1 or 2. Due to the Cartesian Product between each symbol's mutation set, we can always generate the instruction streams that can satisfy the constraint $d4 > 31$ and its negation.

Note that the path explosion in symbolic execution is not an issue for our purpose since the decoding and execution ASL code has limited constraints, resulting in limited paths. Meanwhile, we model the utility function calls (e.g., UInt) so that the

symbol will not be propagated into these functions. Our experiment in Section 4.4.1 shows that we can generate the test cases within 4 minutes.

A Demonstration Example

Table 4.2 describes how we generate all the test cases for instruction `VLD4` in Figure 4.4. In total, we split the encoding diagram into nine parts including seven symbols and two constant values (None in the column "Symbol Name"). For constant values, the initialized mutation set has one fixed value. For other symbols, we initialize the mutation set, which is described in column "Init Mutation Set", according to algorithm 1. Then we extract the constraints, and find the satisfied values. Column "Related Constraints" lists the constraints for each symbol. After solving the constraints and their negations, new mutation sets for each symbol will be generated. Finally, we have the mutation set for each symbol, which is denoted by column "Final Mutation Set". We conduct the Cartesian Product between the mutation set of each symbol. In total, we generate $1 \times 2 \times 1 \times 4 \times 6 \times 2 \times 4 \times 3 \times 5 = 5,760$ test cases for this instruction encoding.

Table 4.2: The generated mutation set for each symbol of instruction VLD4 in Figure 4.4

Symbol Name	Bit Length	Start Offset	End Offset	Type	Init Mutation Set	Related Constraint	Set Added by Solving Constraints and Their Negations	Final Mutation Set	Set Size
None	9	23	31	Fixed Value	"111101000"	NA	NA	"111101000"	1
D	1	22	22	Others in 1 bit	"0", "1"	d4 > 31	"0", "1"	"0", "1"	2
None	2	20	21	Fixed Value	"10"	NA	NA	"10"	1
Rn	4	16	19	Register Index	"0000","0001", "0110","1111"	n == 15	"0000","1111"	"0000","0001", "0110","1111"	4
Vd	4	12	15	Others in 4 bit	"0101","0110", "1001","1100"	d4 > 31	"0000","1101"	"0000","0101","0110", "1001","1100","1101"	6
Type	4	8	11	Others in 4 bit	"0000","0001"	Type == '0000' Type == '0001'	"0000","0001"	"0000","0001"	2
Size	2	6	7	Others in 2 bit	"01","10"	Size == '11'	"00","11"	"00","01","10","11"	4
Align	2	4	5	Others in 2 bit	"00","11"	Align == '00'	"00","01"	"00","01","11"	3
Rm	4	0	3	Register Index	"0000","0001", "0111","1111"	m != 15 m != 13	"0000","1101","1111"	"0000","0001", "0111","1101","1111"	5

4.3.3 Differential Testing Engine

Model the CPU

The differential testing engine receives the generated instruction streams, and detects inconsistent ones. Formally, given one instruction stream I, we denote the state before the execution of I as the initial state CPU_I and the state after the execution of I as the final state CPU_F. We denote the CPU T's initial state $CPU_I(T)$ with the tuple $< PC_T, Reg_T, Mem_T, Sta_T >$. PC denotes the program counter, which points to the next instruction that will be executed. Reg denotes the registers used by processors while Mem denotes the memory space that the tested instruction I may write into. Note we do not consider the whole memory space as comparing the whole memory space is time- and resource-consuming. Sta denotes the status register, which is $APSR$ in ARM architecture. We denote the CPU T's final state $CPU_F(T)$ with the tuple $[PC_T, Reg_T, Mem_T, Sta_T, Sig_T]$. Inside $CPU_F(T)$, all the other attributes have the same meanings as they are inside $CPU_I(T)$ except Sig. Sig denotes the signal or exception that the instruction stream I may trigger. If no signal or exception is triggered, the value of Sig is 0.

Given the CPU emulator E, the real device R, our differential testing engine guarantees that E's initial state $CPU_I(E)$ is equal to R's initial state $CPU_I(R)$. $CPU_I(E) = CPU_I(R)$ iff:

$$\forall \phi \in < PC, Reg, Mem, Sta >: \phi_E = \phi_R$$

After the execution of I, I is treated as an inconsistent instruction stream if the final state $CPU_F(E)$ is not equal to the R's final state $CPU_F(R)$. More formally, $CPU_F(E) \neq CPU_F(R)$ iff:

$$\exists \phi \in [PC, Reg, Mem, Sta, Sig] : \phi_E \neq \phi_R$$

Our Strategy

To conduct the differential testing, we insert prologue and epilogue instructions. We first register the signal handlers to capture different signals. To make the initial state consistent, we set the value of general purpose registers to zero except PC. After setting up the initial state, an instruction stream will be executed. Then we dump the CPU state either after the execution or in the signal handler so that we can compare the execution result. For registers including status register (i.e., APSR), we push them on the stack and then write them into a file. For the memory, we check the instruction stream with Capstone [13] to see whether it will write a value into a memory location. If so, we load the memory address, and push it on the stack for later inspection. Note that the memory write instructions are limited. We manually check the functionality of Capstone on analyzing these instructions and it works well. Finally, we compare the result collected from the emulator and a real device. If the instruction stream results in a different CPU final state, $(CPU_F(E) \neq CPU_F(R))$, it will be treated as an inconsistent instruction stream.

4.3.4 Implementation Details

We implement EXAMINER in Python, C and ARM assembly. In particular, we implement the test case generator in Python. We parse the ASL code, extract the lexical and syntactic information with regular expressions. We use Z3 [50] as the SMT solver to solve the constraints. The differential testing engine is implemented in C and assembly code with some glue scripts in Python. Specifically, the initial state setup and the execution result dumping is implemented with inline assembly code. In total, EXAMINER contains $5,074$ lines of Python code, 220 lines of C code, and 200 lines of assembly code.

4.4 Evaluation

In this section, we evaluate EXAMINER by answering the following three research questions.

- **RQ1:** Is EXAMINER able to generate sufficient test cases?
- **RQ2:** Is EXAMINER able to detect inconsistent instructions? What are the root causes of these inconsistent instructions?
- **RQ3:** Is EXAMINER general to be applied to the other emulators?
- **RQ4:** What are the possible usage scenarios of inconsistent instructions?

4.4.1 Sufficiency of Test Case Generator (RQ1)

We generate the test cases according to ARMv8-A manual, which introduces ASL. Specifically, the manual includes four different instruction sets. In AArch 64 mode, A64 instruction set is supported. For the AArch 32 mode, it consists of three different instruction sets. They are ARM32 with 32-bit instruction length (A32), Thumb-2 with instruction length of mixed 16-bits and 32-bits (T32), and Thumb-1 with 16-bit instruction length (T16). They are also supported by previous ARM architectures (e.g., ARMv5, ARMv6, ARMv7). To locate the inconsistent instructions in different ARM architectures, we generate the test cases for all the instruction sets.

The generated test case is sufficient. Table 4.3 shows the statistics of the generated instruction streams. The column "EXAMINER " denotes the number of different attributes for our test case generator. In total, $2,774,649$ instruction streams are generated within 4 minutes, which cover $1,998$ instruction encodings in $1,070$ instructions. Note that the total number of instruction encodings and instructions in ARM manual is $1,998$ and $1,070$, respectively, which means all the instruction encodings and instructions are covered.

Table 4.3: The statistics of the generated instruction streams. "EXAMINER " denotes the number of generated test cases by our test case generator. "Random" denotes the number of randomly generated test cases. "Ratio" denotes the percentage of dividing "Random" by "EXAMINER ". Note that one instruction may have different instruction encodings for different instruction sets. The total number of instructions for A32, T32, and T16 is 489.

Instruction Set	Time (s) of	Instruction Stream			Instruction Encoding			Instruction			Covered Constraints		
	EXAMINER	EXAMINER	Random	Ratio	EXAMINER	Random	ratio	EXAMINER	Random	ratio	EXAMINER	Random	Ratio
A64	70.51	1,094,700	421,645	38.5%	839	265	31.6%	581	178	30.6%	3,436	934	27.2%
A32	75.05	870,221	578,845	66.5%	550	415	75.5%	481	361	75.1%	4,718	3,725	79%
T32	74.58	808,770	34,598	4.2%	531	351	66.1%	451	283	62.7%	4,425	3,203	72.3%
T16	2.32	958	796	83.0%	78	57	73.1%	68	49	72.1%	122	84	68.9%
Overall	222.46	2,774,649	1,035,884	37.3%	1,998	1,088	54.5%	1,070	550	51.4%	12,701	7,946	62.6%

Note that the generated instruction streams are rather small for T16 due to the small number of instruction encoding schemes and limited instruction length. Overall, all the generated instruction streams are syntactically corrected, which means they all map to one of the encoding schemas. Furthermore, more than 12 thousand constraints and their negations, which are related to encoding symbols, are solved, indicating the multiple behaviors of the instructions are explored.

To further demonstrate the effectiveness of the test case generator, we randomly generate the same number of test cases for each instruction set. We repeat the randomly generated process 10 times. Then we check whether the generated instructions are syntactically correct ones or not. If they are, we calculate how many instruction encodings, how many instructions, and how many constraints are covered by these instruction streams. According to the Column "Random" and "Ratio" in Table 4.3, only 37.3% generated instruction streams are syntactically correct, which means all the others are illegal instructions and they are not effective to test the potential different behaviors between real devices and CPU emulators. Among the syntactically correct instruction streams, it can only cover 54.5% instruction encodings and 51.4% instructions. Nearly a half of instructions can not be covered with the randomly generated instruction streams. Specifically, many of the T32 instructions cannot be covered with randomly generated instructions, which means many of these instructions have fixed values. As for the coverage of constraints, 37.4% constraints can not be explored, resulting in a relatively limited behaviors being explored.

Answer to RQ1: EXAMINER can generate sufficient test cases, which are all syntactical correct instruction streams and can cover all instruction encodings and instructions. On the contrary, Only 37.3% of the same number of randomly generated instruction streams are syntactical correct. Furthermore, 45.5% instruction encodings, 48.6% instructions, and 37.4% constraints cannot be explored by these randomly generated instructions.

4.4.2 Differential Testing Results and Root Causes (RQ2)

We feed the generated test cases into our differential testing engine to locate the inconsistent instructions. Table 4.4 shows the result.

Experiment Setup. We conduct the differential testing between QEMU (version 5.1.0) and four real devices (OLinuXino iMX233 in ARMv5, RaspberryPi Zero in ARMv6, RaspberryPi 2B in ARMv7, and Hikey 970 in ARMv8). For ARMv5, only ARM32 is supported. Meanwhile, QEMU does not support Thumb-2 for ARM1176 of ARMv6. Thus, we only test the A32 instruction set on ARMv5 and ARMv6.

In total, it takes around 2700 seconds of CPU time for QEMU, which is run on the Intel i7-9700 CPU. For the real devices, the CPU time cost ranges from 5276 seconds to 46238 seconds (13 hours), depending on the specific devices. Thanks to the representative test cases, the differential testing for all the test cases can be finished within acceptable time.

Testing Result. According to table 4.4, 171,857 inconsistent instruction streams are found, owning to 6.2% of the whole test cases. Note one instruction stream may be tested in different architectures (e.g.,A32 instruction set in ARMv5, ARMv6, and ARMv7), the number in column "Overall" is the union of the other columns. Furthermore, these inconsistent instruction streams cover 531 different instruction encodings and 316 instructions, owning 26.6% and 29.5% of the tested instruction encodings and instructions, respectively.

Table 4.4: The results of differential testing for QEMU. "CPU Time" denotes the sum of the CPU time for all test cases, which is in seconds. We do not count the sum of CPU time for real devices as they have different CPUs. "Inst" denotes Instruction. "Inst_S" denotes Instruction Stream. "Inst_E" denotes Instruction Encoding. UNPRE. denotes UNPREDICTABLE. X | Y : X denotes the number of the attribute indicated by the row name while Y denotes the percentage of dividing X by Z. For data in "Testing Result", Z stands for the row "Tested Inst_S", "Tested Inst_E", or "Tested Inst". For data in and "Root Cause", Z stands for "Inconsistent Inst_S", "Inconsistent Inst_E", or "Inconsistent Inst".

Architecture	ARMv5	ARMv6	ARMv7		ARMv8	Overall
Experiment Setup						
Instruction Set	A32	A32	A32	T32&T16	A64	-
QEMU Binary	qemu-arm	qemu-arm	qemu-arm		qemu-aarch64	-
QEMU Model	ARM926	ARM1176	Cortex-A7		Cortex-A72	-
Device Name	OLinuXino IMX233	RaspberryPi Zero	RaspberryPi 2B		Hikey 970	-
CPU Time (Device)	46238.0s	6901.7s	6194.2s	5276.0s	9145.0s	-
CPU Time (QEMU)	530.5s	540.6s	538.0s	462.1s	625.9s	2702.1s
Tested Inst_S	870,221	870,221	870,221	809,728	1,094,700	2,774,649
Tested Inst_E	550	550	550	609	839	1,998
Tested Inst	481	481	481	462	581	1,070
Testing Result	The percentage is based on the number of tested instructions (streams/encodings)					
Inconsistent Inst_S	40,896 | 4.7%	18,043 | 2.1%	66,860 | 7.7%	51,821 | 6.4%	21,374 | 2.0%	171,857 | 6.2%
Inconsistent Inst_E	184 | 33.5%	175 | 31.8%	273 | 49.6%	271 | 44.5%	17 | 2.0%	531 | 26.6%
Inconsistent Inst	173 | 40.0%	167 | 34.7%	232 | 48.2%	228 | 49.4%	15 | 2.6%	316 | 29.5%
Root Cause	The percentage is based on the number of inconsistent instructions (streams/encodings)					
Bugs (Inst_S)	1 | 0.0%	1 | 0.0%	1 | 0.0%	580 | 1.1%	2 | 0.0%	583 | 0.3%
Bugs (Inst_E)	1 | 0.5%	1 | 0.6%	1 | 0.4%	7 | 2.6%	2 | 11.8%	10 | 1.9%
Bugs (Inst)	1 | 0.6%	1 | 0.6%	1 | 0.4%	5 | 2.2%	2 | 13.3%	8 | 2.5%
UNPRE. (Inst_S)	40,891 | 100.0%	18,042 | 100.0%	66,859 | 100.0%	51,241 | 98.9%	21,372 | 100.0%	171,274 | 99.7%
UNPRE. (Inst_E)	183 | 99.5%	174 | 99.4%	272 | 99.6%	270 | 97.4%	15 | 88.4%	527 | 98.1%
UNPRE. (Inst)	172 | 99.4%	166 | 99.4%	231 | 99.6%	227 | 97.8%	13 | 86.7%	312 | 97.5%

```
1  boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)
2  // It is IMPLEMENTATION DEFINED whether the
3  // detection of memory aborts happens before or
4  // after the check on the local Exclusive Monitor.
5  // As a result, a failure of the local monitor can
6  // occur on some implementations  even if the
7  // memory access would give an memory abort.
8      ...
9      return
```

Figure 4.5: Two different implementations are defined in the annotation of function ExclusiveMonitorsPass, which is called by many instructions' executing code

Root Cause. Based on the inconsistent streams, we explore the root cause. First, there are implementation bugs. We discovered 4 bugs of QEMU in total, which influence 10 instruction encodings. Some of the bugs are related to very common instructions. For example, `BLX` instruction may be undefined instructions in specific cases, which should raise SIGILL signal. However, QEMU does not follow the specification. We also noticed one instruction (i.e., `WFI`) that can make QEMU crash. `WFI` denotes waiting for interrupt and is usually used in system-mode emulation. However, ARM manual specifies that it can also be used in user-space. QEMU does not handle this instruction well and an abort will be generated. All of the 4 bugs are confirmed and patched by QEMU developers. This also demonstrates the capability of EXAMINER in discovering the bugs of the emulator implementation.

Apart from the bugs, most of the inconsistent instructions are due to the undefined implementation in the ARM manual. There are three different kinds of undefined implementations. The first one is UNPREDICTABLE (Section 4.3.1). UNPREDICTABLE leaves open implementation decision for emulators and processors. The second is Constraint UNPREDICTABLE. Constraint UNPREDICTABLE provides candidate implementation strategies and the developer or vendor can choose from one of them. The last one is defined in the annotation part of the ASL code. Figure 4.5 shows an example. In the function `ExclusiveMonitorsPass`, which is called by the executing code of instruction `STREXH`, there is an annotation for the implementation. Note the check on the *local Exclusive Monitor* would update the value

Table 4.5: The statistics on detecting emulators

Mobile Type	CPU	A64	A32	T32 & T16
Samsung S8	SnapDragon 835	✓	✓	✓
Huawei Mate20	Kirin 980	✓	✓	✓
IQOO Neo5	SnapDragon 870	✓	✓	✓
Huawei P40	Kirin 990	✓	✓	✓
Huawei Mate40 Pro	Kirin 9000	✓	✓	✓
Honor 9	Kirin 960	✓	✓	✓
Honor 20	Kirin 710	✓	✓	✓
Blackberry Key2	SnapDragon 660	✓	✓	✓
Google Pixel	SnapDragon 821	✓	✓	✓
Samsung Zflip	SnapDragon 855	✓	✓	✓
Google Pixel3	SnapDragon 845	✓	✓	✓

of a register. Thus, if the detection of memory aborts happens before the check, the value of the register would not be updated while the detection happens after the check can update the value, resulting in different register value.

Answer to RQ2: EXAMINER can detect inconsistent instructions. In total, 171,274 inconsistent instruction streams are found, which covers 26.6% (i.e.,531/1998) instruction encodings and 29.5% instructions (i.e., 316/1070). The implementation bugs of QEMU and the undefined implementation in ARM manual are the major root causes. 4 bugs are discovered and confirmed by QEMU developers, which influence 10 instruction encodings including commonly used instructions (e.g., `BLX`).

4.4.3 Generalization of EXAMINER (RQ3)

Table 4.6: The results of differential testing for Unicorn and Angr. The attributes denotes the same meaning explained in the caption of Table 4.4.

Tool	Unicorn				Angr			
Architecture	ARMv7		ARMv8	Overall	ARMv7		ARMv8	Overall
Instruction Set	A32	T32 & T16	A64	-	A32	T32 & T16	A64	-
CPU Time	31.8s	32.9s	32.4s	97.1s	7654.2s	7873.1s	10004.1s	25531.4s
Tested Inst_S	328,780	336,987	371,770	1,037,537	328,780	336,987	371,770	1,037,537
Tested Inst_E	352	398	205	955	352	398	205	955
Tested Inst	313	285	77	418	313	285	77	418
Testing Result	The percentage is based on the number of tested instructions (streams/encodings)							
Inconsistent Inst_S	103,520 | 31.5%	119,394 | 35.4%	350 | 0.1%	223,264 | 21.5%	70,493 | 21.4%	37,364 | 11.1%	12,312 | 3.3%	120,169 | 11.6%
Inconsistent Inst_E	267 | 75.9%	300 | 75.4%	3 | 1.5%	570 | 59.7%	154 | 43.8%	161 | 40.4%	23 | 11.2%	338 | 35.4%
Inconsistent Inst	231 | 73.8%	254 | 89.1%	2 | 2.6%	298 | 71.3%	126 | 40.3%	130 | 45.6%	10 | 13.0%	197 | 47.1%
Root Cause	The percentage is based on the number of inconsistent instructions (streams/encodings)							
Bugs (Inst_S)	0 | 0.0%	529 | 0.4%	0 | 0.0%	529 | 0.2%	0 | 0.0%	0 | 0.0%	0 | 0.0%	0 | 0.0%
Bugs (Inst_E)	0 | 0.0%	7 | 2.3%	0 | 0.0%	7 | 1.2%	0 | 0.0%	0 | 0.0%	0 | 0.0%	0 | 0.0%
Bugs (Inst)	0 | 0.0%	5 | 2.0%	0 | 0.0%	5 | 1.7%	0 | 0.0%	0 | 0.0%	0 | 0.0%	0 | 0.0%
UNPRE. (Inst_S)	103,520 | 100%	118,865 | 99.6%	350 | 100%	222,735	70,493 | 100%	37,364 | 100%	12,312 | 100%	120,169 | 100%
UNPRE. (Inst_E)	267 | 100%	296 | 99.7%	3 | 100%	566 | 98.8%	154 | 100%	161 | 100%	23 | 100%	338 | 100%
UNPRE. (Inst)	231 | 100%	253 | 98.0%	2 | 100%	297 | 98.3%	126 | 100%	130 | 100%	10 | 100%	197 | 100%

To demonstrate the generalization of EXAMINER, we further apply EXAMINER on evaluating the other two lightweight but also popular CPU emulators (i.e., Unicorn in version 1.0.2rc4 and Angr in version 9.0.7833). Different from QEMU, Unicorn and Angr do not provide options to specify the ARMv5 or ARMv6 architecture. In this case, we evaluate ARMv7 and ARMv8. Meanwhile, Unicorn and Angr do not have good support on advanced instructions [8]. For instance, many SIMD instructions will make Angr crash, resulting in 5 new bugs. Instructions (e.g., `WFE` [9]) that rely on kernel or multiprocessor are also not supported. Thus, we filter out these instructions in the experiment.

Table 4.6 shows the result. 223, 264 and 120, 169 inconsistent instructions streams are identified for Unicorn and Angr, respectively. They also cover hundreds of instruction encodings. We also explored the root cause of these inconsistent instructions. Similar to QEMU, undefined implementation and bugs are the major causes. 3 bugs are located in Unicorn.

Answer to RQ3: EXAMINER is general to be applied to the other CPU emulators (i.e., Unicorn and Angr). With EXAMINER, we disclosed 8 more bugs (5 in Angr and 3 in Unicorn) and located a huge number of inconsistent instruction streams in the two CPU emulators).

4.4.4 Applications of Inconsistent Instructions (RQ4)

The inconsistent instructions can be used to detect the existence of emulators. Furthermore, detecting emulators can prevent the binary from being analyzed or fuzzed, which is known as anti-emulation and anti-fuzzing technique.

Emulator Detection

The inconsistent instructions can be used to detect emulators. We use the inconsistent instructions for emulator detection. Considering the popularity of Android systems, we target Android applications. Specifically, we build a native library by

```
1  void sig_handler(int signum) {
2      record_execution_result(i++);
3      siglongjmp(sig_env, i);
4  }
5
6  Bool JNI_Function_Is_In_Emulator() {
7      register_signals(sig_handler);
8      i = sigsetjmp(sig_env,0);
9      switch (i){
10         case 1:
11             execute(inconsistent_instruction_n);
12             record_execution_result(i++);
13             longjmp(sig_env,i++);
14         case 2:
15         ...
16         case n:
17     }
18     return compare_result();
19 }
```

Figure 4.6: Pseudo code of the native code for detecting the emulator.

using the inconsistent instructions.

Figure 4.6 shows the pseudo code of the library. Function *JNI_Function_Is_In_Emulator* (line 6) returns True if the emulator is detected. Inside the function, we register signal handlers for different signals (line 7). After the execution of each instruction stream, we will record the execution result either in the signal handler (line 2) or after the execution (line 12), which depends on the execution result. Then we use the function *longjmp* (line 13) or *siglongjmp* (line 3) to jump back to the place where calling *sigsetjmp* (line 8). Note the return value of *sigsetjmp* depends on the parameter of the *longjmp* or *siglongjmp*. Thus, the *switch* would check the value of i, which is the return value of *sigsetjmp*, repeatedly. As i would increase one after the execution of one instruction stream, we can execute hundreds of instruction streams in one function by adding corresponding *case* conditions. Each instruction stream can make an equal contribution to the final decision on whether the current execution environment is in real devices or emulators. Finally, if more instruction streams decide the application are running inside an emulator, the *compare_result()* will return True and vice versa.

We automatically generate the test library with template code and build three Android apps for different instruction set (one for A64, one for A32, and one for T32 &

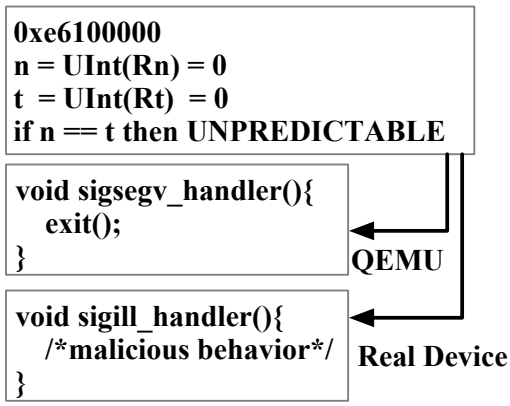


Figure 4.7: Inconsistent instruction can prevent the malicious behavior being detected by emulators

T16). We run the applications on 12 different mobiles from 6 different vendors. These mobiles also use different CPUs, which decides how an instruction stream should be executed. Meanwhile, we run the applications in the Android emulator provided by Android studio (Version 4.1.2). We compare the result running in each real mobiles with the one in emulators. If the function *JNI_Function_Is_In_Emulator* returns True in emulator and returns False in real mobiles. We consider it will successfully detect the emulator. Table 4.5 shows the evaluation result, by testing the three Android apps (one for A64, one for A32, and one for T32 & T16) in 12 mobiles, all the mobile apps can detect the existence of emulator and real mobiles successfully.

Anti-Emulation

Anti-emulation technique is important. On the attacker's side, it can be proposed to increase the bar for analyzing the malware so that the defense mechanism can be developed slower. On the defender's side, commercial software needs to protect the core functionality and algorithms from being analyzed. Thus, it is widely used in the wild [151].

The inconsistent instructions can be used to conduct anti-emulation and can prevent the malware's malicious behavior being analyzed. We demonstrate how the

```
1    0x10000: e51b3008 LDR r3,[fp,#-8]
2    0x10004: e1a03000 MOV r3,r0
3    0x10008: e7cf0e9f BFC r0, #0xf, #1
4    // BFC instruction is to clear specific bits
5    // e7cf0e9f is an UNPREDICTABLE encoding
6    // e7cf0e9f is executed normally in real device
7    // e7cf0e9f triggers SIGILL signal on QEMU
8    0x1000c: e1a00003 MOV r0,r3
9    0x10010: e50b3008 STR r3,[fp,#-8]
```

Figure 4.8: Instrumented instruction streams for anti-fuzzing.

inconsistent instruction can be used to hide the malicious behavior.

We use one of the state-of-the-art dynamic analysis platforms (i.e., PANDA [35]) to demonstrate the usage. PANDA is built upon QEMU and supports taint analysis, record and replay, operating system introspection, and so on. We port one of the open source rootkits (i.e., Suterusu [45]) to Debian 7.3. We register two different signal handlers for SIGILL and SIGSEGV, respectively. Then we instrument one instruction stream (i.e., 0xe6100000). This is a LDR instruction encoding in ARM instruction set. According to the encoding schema, n equals to t and both these two symbols' values are zero. The ASL code of decoding would check whether n equals to t. If so, it should be the UNPREDICTABLE behavior. Real devices think this is an illegal instruction stream and will raise the SIGILL signal while QEMU tries to execute the instruction stream. Then SIGSEGV will be raised as the address pointed by R0 cannot be accessed. In this case, the malicious behavior will only be triggered in real devices. Meanwhile, when we use the PANDA to analyze the malware, no malicious behavior will be monitored and the program will exit inside the *sigsegv_handler*.

Anti-Fuzz

Fuzzing is widely used to explore the zero-day vulnerabilities. To help the released binaries from being fuzzed by attackers, researchers utilize anti-fuzzing techniques [97, 85]. Considering that many new binary fuzzing frameworks are based on QEMU, the inconsistent instructions can be used by developers as a mitigation

approach towards fuzzing technique.

We demonstrate how the inconsistent instructions can be used to conduct anti-fuzzing tasks with a relatively low overhead and high decreased coverage ratio. Figure 4.8 shows a snippet of assembly code instrumented into the release binary. In address 0x10008, the instruction `BFC` is used to clear bits for register R0. Note we move the value of R0 to R3 before the instruction `BFC` and return it back after the execution of `BFC`. This can guarantee the instrumented instructions will not affect the execution of the binary on the real device. The instruction stream 0xe7cf0e9f results in an UNPREDICTABLE condition. It can be executed normally in real devices while triggering a signal on QEMU.

We developed a GCC plugin to instrument the above mentioned inconsistent instruction streams at each function entry and apply this plugin on three popular used libraries (i.e., libtiff, libpng, and libjpeg) during the compilation process to generate released binaries.

Table 4.7 shows the space and runtime overhead of the instrumented binary compared with the normal (non-instrumented) one. The space overhead is measured by comparing the binary size. For runtime overhead, we measure it by running test suites on both binaries and comparing the cost of time. We noticed that the instrumented binary imposes negligible space and runtime overhead to the binary. The average space overhead for the protected binary is around 4%, and the runtime overhead is less than 1%.

We then measure the functionality of anti-fuzzing. We fuzz the instrumented binaries and the normal ones with AFL-QEMU (version 2.56b) for 24 hours. The seed corpus is the test suite used for each library in Table 4.7. We collect the coverage information for the instrumented and the normal ones. Figure 4.9 shows the results. It is easy to see that the coverage for instrumented binaries cannot increase (because QEMU fails to execute binaries correctly), while the normal ones will increase with

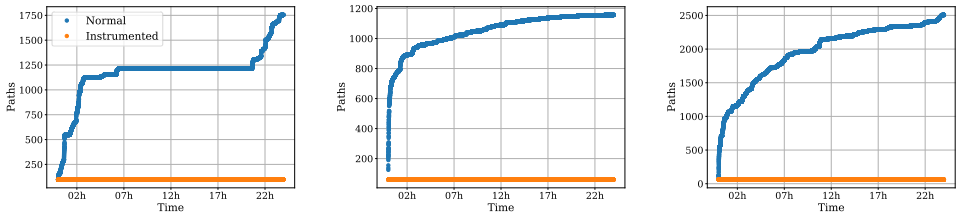
Table 4.7: Overhead information of anti-fuzzing.

Library [1]	Test Suite[2]	Space Overhead	Runtime Overhead
libpng (readpng)	built-in (254)	4.0% (+7KB)	0.52%
libjpeg (djpeg)	GIT [3] (97)	4.3% (+8KB)	0.61%
libtiff (tiffinfo)	built-in (61)	2.2% (+8KB)	0.59%
Overall		3.5%	0.57%

[1] All libraries are compiled using default compile parameters.
[2] The test inputs for libjpeg is taken from Google Image Test Suite.
[3] The number of test inputs in test suite is shown in the bracket.

the fuzzing time.

Note this is to demonstrate the ability of inconsistent instructions on anti-fuzzing tasks. How to stealthily use these instructions is out of our scope. It is not easy for attackers to precisely recognize all the inconsistent instructions, which will be discussed in detail (Section 4.5).



(a) libjpeg (b) libpng (c) libtiff

Figure 4.9: The result of Anti-Fuzzing experiment on three libraries. The blue lines show the coverage over 24 hours of fuzzing. The orange line shows the coverage for instrumented binaries, which decreases due to failed executions of QEMU.

Answer to RQ3: The inconsistent instructions are useful. We demonstrate that the inconsistent instructions can be used to detect the existence of the CPU emulator and prevent the malicious behavior from being monitored by dynamic analysis frameworks. Furthermore, the path coverage of programs fuzzed in emulators can be highly decreased with the help of inconsistent instructions.

4.5 Discussion

Detecting (Ab)Used Inconsistent Instructions Section 4.4.4 shows that attackers or vendors can (ab)use these inconsistent instructions. It is not easy to recognize these inconsistent instructions due to the huge number of inconsistent instructions. Some of them are even commonly used (i.e., `BLX`). Apart from this, attackers can encrypt these instruction streams as data. Then these encrypted instruction streams can be decrypted and executed during runtime, which can increase the bar for detection. Furthermore, how to hide these inconsistent instruction streams from being detected is a *Cat and Mouse* problem. Stealthily using these instructions is out of our scope.

Testing Instructions in Privileged Environments Currently, the generated instruction streams are tested under unprivileged mode in both CPU emulators and real devices. Some instruction streams may have different execution results under privileged mode. For instance, instruction `WFI`, which results in a bug of QEMU user-mode, may not be an inconsistent instruction while executing in privileged mode. We plan to port EXAMINER to kernel-space in the future.

Testing Instruction Stream Sequences EXAMINER now tests only one instruction stream each time during the differential testing. We can also test multiple instruction streams (instruction stream sequences) in the differential testing. The instruction stream sequences may trigger multiple system states and we can test the decoding/executing logic towards different state flags. How to design representative instruction stream sequences, and how to locate the inconsistent one will be the challenge, which is left as future work. Nevertheless, We have already discovered a huge number of inconsistent instruction streams with EXAMINER, covering 29.5% of instructions. Every instruction stream sequence that contain the inconsistent instruction stream can result in inconsistent behaviors.

Other Architectures The whole framework of EXAMINER is architecture-independent. However, we rely on ARM ASL to generate the test cases, which can explore multiple behaviors. If other architectures propose such kinds of specification language, we are able to generate the test cases. Otherwise, new test case generation algorithm should be developed.

4.6 Summary

We design and implement EXAMINER, a framework that can automatically locate the inconsistent ARM instructions. With EXAMINER, we generate 2,774,649 representative instruction streams and detect 171,857 inconsistent ones for QEMU. To demonstrate EXAMINER's generalization, we further apply EXAMINER on two other emulators (i,e,, Unicorn and Angr) and a huge number of inconsistent instructions are located. We noticed that bugs and undefined implementation in ARM manual are the root causes. Furthermore, we disclosed 12 bugs (4 in QEMU, 3 in Unicorn, 5 in Angr). Some of them influence commonly used instructions (e.g., `BLX`) and can even crash the emulators (e.g., QEMU and Angr). We also demonstrate the capability of inconsistent instructions on detecting emulators, anti-emulation, and anti-fuzzing.

Chapter 5

ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels

5.1 Overview

Dynamic analysis has been widely used for various purposes [83, 69, 87, 109, 160, 93]. It can monitor the runtime behavior of the target system, complementing the static analysis [126, 137, 67, 88]. Rehosting, also known as emulation, is used to run a target system inside an emulated environment, e.g., QEMU, and provides the capability to introspect the runtime state. Based on this capability, different applications, e.g., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, can be built. Running the Linux kernel in QEMU for the desktop system is a solved problem. However, rehosting embedded system is challenging. First, rehosting Linux kernel is dependent on the emulation of peripherals. Without the right emulation of these peripherals, Linux kernel may halt or crash during the rehosting process. Second, peripherals vary widely. Due to the diverse peripherals in the wild, it is not practical for QEMU to support all kinds of peripherals in any SoC. Third, vendors may not strictly follow the GPL license [84, 75], resulting in the lack of public information (e.g., specifications, datasheets, and source code). These obstruct the diagnosis of failures when adding emulation support of new SoCs in QEMU. Thus, how to rehost

the embedded Linux kernels in QEMU is still an open research question.

Previous research [61, 101] provides the capability of rehosting user-space programs by running a customized Linux kernel for one SoC that is supported in QEMU. This works well because user-space programs mainly depend on standard system calls that are provided by the underlying Linux kernel. Different from user-space programs, the OS kernel interact with peripherals that are usually different in different SoCs. Some researchers have proposed to use real devices to perform the dynamic analysis [156, 118, 86, 142]. Such solutions do not scale since there exist a large number of embedded devices. Other mechanism that are for the bare-metal systems [79, 115, 65], i.e., embedded systems without an OS kernel or having a thin layer of abstraction, cannot be directly used to rehost the Linux kernel as the Linux kernel is far more complicated than the bare-metal ones.

Key Insights. To address the above mentioned three challenges, we have three key insights. First, only early-boot peripherals (i.e., interrupt controller, timer, and UART) need to be supported during the rehosting process. After successfully rehosting the Linux kernel, we are able to install the different peripheral drivers in ramfs to support the other peripherals with kernel modules. Second, Linux kernel provides interfaces to implement drivers of these peripherals, which brings the chance to replace these diverse peripherals with designated ones. Third, embedded Linux kernels are usually modified based on the mainstream Linux kernel, which is open-sourced. The modification mainly aims to add support for specific peripherals while most of the other code is unchanged.

With the insights, we propose a solution called *peripheral transplantation. It is device-independent, and works towards the Linux kernel binary without the need of the source code of the target system.* The main idea is, instead of manually adding emulation support of various peripherals in QEMU, we can transplant the device

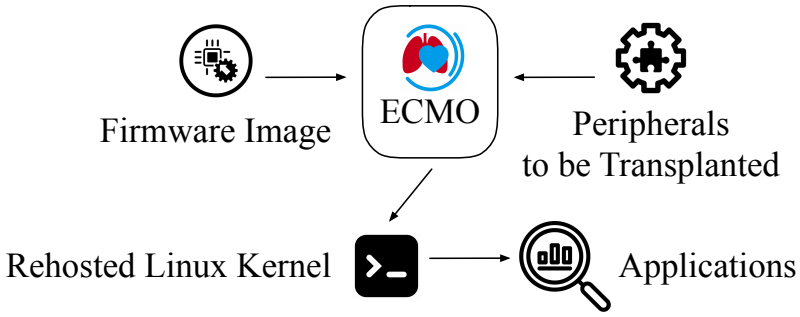


Figure 5.1: The overview of our system (ECMO)

drivers of designated peripherals into the target Linux kernel binary. It replaces the peripherals in the target Linux kernel that are currently unsupported in QEMU with supported ones, thus making the Linux kernel *rehostable*. In particular, given a Linux kernel retrieved from the firmware image of an embedded device, our system turns it into a rehostable one that can be successfully booted in QEMU. After that, various applications can be built to analyze the rehosted kernel.

Specifically, our system transplants two components, i.e., the emulated models of peripheral into QEMU and their device drivers into the Linux kernel (if they are not initialized originally). Transplanting the peripheral model requires us to build the hardware emulation code for a specified (or simplified) peripheral and integrate it into `QEMU`. We utilize APIs provided by `QEMU` for transplanting emulated peripheral models.

However, transplanting a driver into the Linux kernel is non-trivial. *First*, we need to substitute the original (unsupported) device driver with the transplanted one. Since the peripheral driver is initialized with indirect calls, we need to locate function pointers and rewrite them in a stripped binary on the fly, which is challenging. *Second*, the transplanted driver should not affect the memory view of the original kernel. Otherwise, the memory holding the transplanted driver can be overwritten since the Linux kernel is not aware of the existence of that memory region. *Third*, the transplanted driver needs to invoke APIs in the Linux kernel. Otherwise, the

transplanted driver cannot function as desired.

To overcome the difficulties of transplanting drivers, we design and implement a new algorithm to identify the required function pointers (Section 5.4.2) and introduce *opaque memory* (Section 5.4.3) to guarantee that the transplanted driver does not affect the memory view of the original kernel. Finally, we implement and integrate the *peripheral transplantation* technique into QEMU to create a prototype called ECMO. Figure 5.1 shows the overview of ECMO. It receives the firmware image and the peripherals to be transplanted. Then it transplants the peripherals to the Linux kernel binary to make it rehostable in QEMU and launch a shell. Note that ECMO focuses on transplanting the early-boot peripherals (i.e., interrupt controller, timer, and UART), which are needed to rehost the Linux kernel. Once the Linux kernel is rehosted, users can install different peripheral drivers to support more peripherals with kernel modules and build various applications to analyze the rehosted kernel.

We apply ECMO on 815 Linux kernels extracted from firmware images, including 20 different kernel versions and 37 device models. ECMO now only supports ARM architecture, which is widely used in embedded systems [41]. However, it does not rely on any architecture specific feature and can be easily extended to the other architectures (Section 5.6). Our experiment shows that ECMO can successfully transplant peripherals for all 815 Linux kernels. Among them, 710 are able to launch a shell. The failed cases are due to the unsupported root file system format (*ramfs*) in the rehosted kernel. Furthermore, we successfully install one Ethernet device driver (i.e., smc91x) on all the rehosted Linux kernel, which demonstrates the capability to support more peripherals based on rehosted Linux kernel. To demonstrate the functionality and usefulness of our system, we build and port three applications, including kernel crash analysis, rootkit forensic analysis, and kernel fuzzing. Note that, the applications themselves are not the contribution of our work. They are used to demonstrate the usage scenarios of our system. Other applications that can

be built on QEMU can also be ported.

In summary, this work makes the following main contributions.

- **Novel technique.** We propose a *device-independent* technique called *peripheral transplantation* that can rehost Linux kernels of embedded devices without the availability of the source code.
- **New system.** We implement and integrate the *peripheral transplantation* technique into QEMU, to create a prototype system called ECMO.
- **Comprehensive evaluation.** We apply ECMO to 815 Linux kernels from different images. It can transplant peripherals for all the Linux kernels and successfully launch the shell for 710 ones.

To engage with the community, we release the source code of our system in https://github.com/valour01/ecmo.

5.2 Background

5.2.1 Linux Kernel

Linux kernel source code can be categorized into three types according to their functionalities. The first type is the *architecture independent code*, which contains the core functionality used by all CPU architectures. The second type is *architecture dependent code*. For instance, the sub-directories under the *arch/* directory contain the code for multiple CPU architectures. The third type is *board-specific code*, which is used by specific board (machine). For instance, the directory *arch/arm/versatile/* contains the code used by the machine named *versatile*. The kernel compiled for one machine usually cannot be directly booted on other machines (or QEMU instances that emulate different machines.)

```
1  MACHINE_START(VERSATILE_AB, "ARM-Versatile AB")
2      .atag_offset = 0x100,
3      .map_io = versatile_map_io,
4      .init_early = versatile_init_early,
5      .init_irq = versatile_init_irq,
6      .init_time = versatile_timer_init,
7      .init_machine = versatile_init,
8      .restart = versatile_restart,
9  MACHINE_END
```

Figure 5.2: The machine description for *ARM-Versatile AB*.

```
1  //UART read call back
2  static uint64_t serial_mm_read(void *opaque,
3      hwaddr addr, unsigned size) {
4    SerialMM *s = SERIAL_MM(opaque);
5    return serial_ioport_read(&s->serial,
6                              addr >> s->regshift, 1);
7  }
8  //register read/write call back functions
9  static const MemoryRegionOps serial_mm_ops = {
10   .read = serial_mm_read,
11   .write = serial_mm_write,
12   ...
13 };
```

Figure 5.3: The callback functions for UART emulation in QEMU

5.2.2 ARM Machines

Embedded systems usually use SoCs from multiple vendors with different designs. For instance, they contain different peripherals. Each SoC is expressed as a machine in the Linux kernel. Manufacturers develop the *board support package* (BSP) (e.g., drivers of peripherals) so that Linux kernel can use these peripherals.

Linux kernel introduces the structure *machine_desc* for ARM to describe different machines. The structure *machine_desc* provides interfaces to implement BSPs. For example, Figure 5.2 shows an example of one machine *ARM-Versatile AB* in the Linux kernel (Version 3.18.20). It initializes function pointers and data pointers with its implementation. Specifically, in line 5, the function pointer *init_irq* is assigned the value as *versatile_init_irq*. During the booting process, the Linux kernel will invoke the function *machine_desc→init_irq* to initialize the IC (interrupt controller). The same logic applies to the function pointer *init_time*. Linux kernel invokes the function *machine_desc→init_time* to initialize the timer.

5.2.3 QEMU

QEMU [57] is one of the most popular full-system emulators. It emulates different machines by providing different machine models. A machine model consists of CPU, memory, and different kinds of peripheral models. To emulate a peripheral, QEMU registers the read/write callback functions for the MMIO (memory-mapped I/O) address space of the peripheral. Once the Linux kernel running inside QEMU reads from or writes into the address inside the MMIO range, the registered callback functions inside QEMU will be invoked to emulate the peripheral. Basically, it maintains an internal state machine to implement the peripheral's functionality. Figure 5.3 shows an example of the registered callback functions for UART emulation. Specifically, when the Linux kernel reads from the MMIO space of the emulated UART device (e.g., `0x01C42000`), the `serial_mm_read` function will be invoked by QEMU to emulate the read access.

5.3 Challenges and Our Solution

The main goal of our work is to rehost Linux kernel binaries that are originally running on embedded systems in QEMU. This lays the foundation of applications that rely on the capability to introspect runtime states of the Linux kernel, e.g., kernel crash and vulnerability analysis [83, 69], rootkit forensic analysis [146, 128], and kernel fuzzing [109, 133].

5.3.1 Challenges

Rehosting the Linux kernel on QEMU faces the following challenges:

Peripheral dependency. Rehosting the Linux kernel requires QEMU to emulate the peripherals, e.g., the interrupt controller, that the Linux kernel depends on. During the booting process, Linux kernel will read from or write into the peripheral

registers and execute the code according to the state specified by the value of peripheral registers. Without the emulation of these peripherals, the rehosted kernel will halt or crash during the booting process.

Peripheral diversity. SoCs vary widely [43] and different vendors, e.g., Broadcom, Marvell may design and develop different SoCs. These new SoCs introduce many new peripherals that are not currently supported in QEMU and the open-sourced mainstream of the Linux kernel. Due to the diversity of peripherals, there are still a large number of devices that are not supported. Meanwhile, manually developing peripheral emulation routine is tedious and error-prone, especially due to the diversity of peripherals. Thus, the diversity of peripherals brings significant challenge to build a general emulator, which can re-host various Linux kernels of embedded devices.

Lack of public information. The information (e.g., specifications, datasheets, and source code) of SoCs and firmware images are usually not public. This is because vendors may not release the detailed hardware specification. Furthermore, vendors may not release the source code immediately after releasing the image and not all vendors strictly follow the GPL license [84, 75]. Meanwhile, the binary of the Linux kernel is stripped and has no particular headers (i.e., ELF section headers) or debugging information. These obstruct the diagnosis of failures when adding emulation support of new SoCs in QEMU.

5.3.2 Our Solution: Peripheral Transplantation

In this work, we propose a technique called *peripheral transplantation*. The main idea is, instead of manually adding emulation support of various peripherals in QEMU, we can *replace the peripherals that are used in target Linux kernels with existing peripherals in QEMU*. By doing so, we can rehost the Linux kernel and the kernel functionality is intact (Section 5.5.4).

Figure 5.4 shows the overview of peripheral transplantation. This involves the

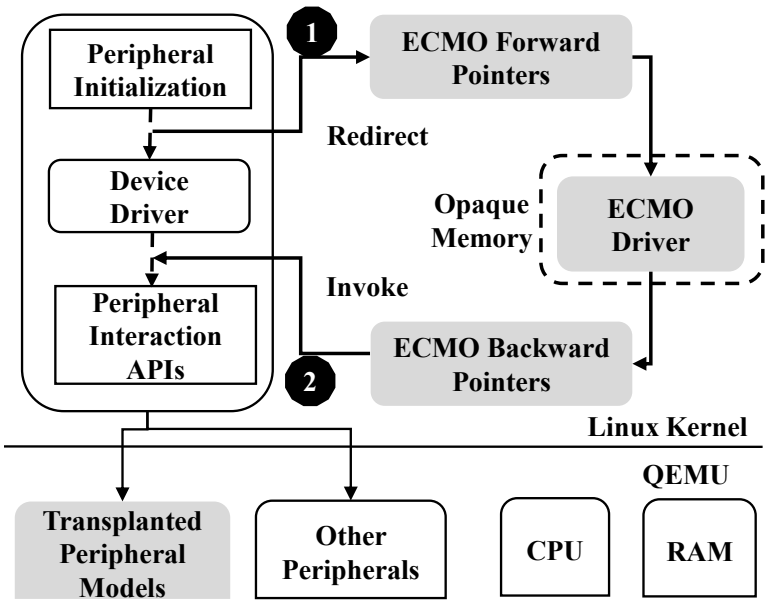


Figure 5.4: The overview of peripheral transplantation.

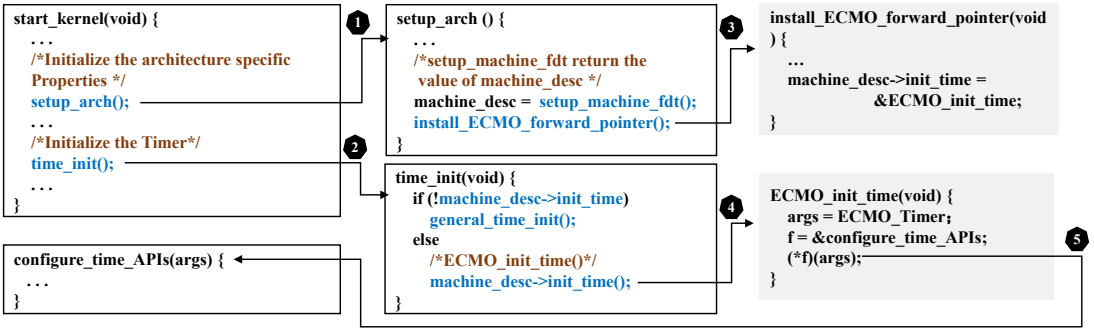


Figure 5.5: A concrete example of peripheral transplantation.

injection of peripheral models into QEMU and the *ECMO Driver* into the Linux kernel. To distinguish them from original ones of the (emulated) machine, we call the transplanted peripheral models *ECMO Peripheral*. To let the kernel use the transplanted *ECMO Driver*, our system identities the functions that are used to initialize device drivers (*ECMO Forward Pointers*) and redirects them to the functions inside the *ECMO Driver* (Fig. 5.4 ❶). Moreover, our system identifies the APIs that are responsible for interacting with peripheral models. These APIs are used by

Figure 5.6: The work flow of our system.

the *ECMO Driver* to communicate with the transplanted peripheral models (Fig. 5.4 ❷). The addresses of these functions are called *ECMO Backward Pointers* in this paper. We will elaborate how to identify the ECMO Pointers in Section 5.4.2.

Note that, to ensure the *ECMO Driver* does not affect the memory view of the rehosted Linux kernel, we propose the concept of the *opaque memory*. This memory region is available on the emulated machine but cannot be seen by the Linux kernel. As such, we can prevent the kernel from allocating memory pages that are reserved for the *ECMO Driver*. We will elaborate this in Section 5.4.3.

5.3.3 An Illustration Example of Peripheral Transplantation

Fig. 5.5 shows a concrete example of transplanting one peripheral (i.e., timer) into the Linux kernel. In particular, the function `start_kernel` is responsible for initializing the Linux kernel. It will invoke several different functions, including `setup_arch` and and `time_init`.

The function `setup_arch` will setup architecture-related configurations and initialize the `machine_desc` structure (Fig. 5.5 ❶). This structure contains multiple function pointers (*ECMO Forward Pointers*) that will be used to initialize corresponding drivers. Our system first locates the function `setup_arch` and then injects a function (`install_ECMO_forward_pointers`) to change the pointers to our own ones (Fig. 5.5 ❸).

When the function `init_time` is invoked to initialize the timer (Fig. 5.5 ❷), the `ECMO_init_time`, which is pointed by `machine_desc-> init_time`, will be invoked to

```
1  Assembly code:
2    mov r0, #0
3    str r0, [r2], #4
4    str r0, [r2], #4
5    str r0, [r2], #4
6    str r0, [r2], #4
7    cmp r2, r3
8    blo 1b
9    tst r4, #1
10   bic r4, r4, #1
11   blne  cache_on
12   mov r0, r4 //r0 stores the value of output_start
13   mov r1, sp
14   add r2, sp, #0x10000
15   mov r3, r7
16   bl  decompress_kernel
17   // we can dump the decompressed Linux kernel after
18   // function decompress_kernel returns
19
20 Simplified C code:
21   void decompress_kernel(uint32 output_start, args)
```

Figure 5.7: The assembly code that invokes function *decompress_kernel*, which is in *arch/arm/boot/compressed/head.S*.

initialize the injected timer driver (*ECMO Driver*) in QEMU (Fig. 5.5 ❹) (through *ECMO Forward Pointers*), instead of the original one. Accordingly, this function will invoke APIs (through *ECMO Backward Pointers*) in the Linux kernel to interact with the *ECMO Peripheral* (Fig. 5.5 ❺).

Note that, the code snippets in Fig. 5.5 are for the illustration purpose. *Our system does not rely on the availability of the source code. It directly works towards the Linux kernel binary that is retrieved from a firmware image.*

5.4 System Design and Implementation

In order to rehost Linux kernels, our system first extracts and decompresses the Linux kernel from the given firmware image (Section 5.4.1). We then apply multiple strategies to identify both *ECMO Forward and Backward Pointers* (Section 5.4.2). These pointers are essential for *ECMO Drivers*. At last, we semi-automatically generate *ECMO Drivers* and load them at runtime to boot the kernels (Section 5.4.3). Fig. 5.6 shows the overall workflow.

5.4.1 Decompress Linux Kernel

Firmware image usually consists of the OS, which is the Linux kernel, and user applications. However, the Linux kernel inside the firmware images is usually compressed. To identify ECMO Pointers, we need to first extract the Linux kernel and decompress it. With the decompressed Linux kernel, we can utilize different strategies to locate the ECMO Pointers.

Specifically, we feed the firmware image to firmware extraction tool (i.e., Binwalk) to extract the kernel image. Then we directly feed the extracted kernel image (with added u-boot information) to QEMU. Since the code for decompressing the Linux kernel does not operate on the peripherals (except the UART to show the message of decompressing Linux kernel), it can be successfully executed in vanilla QEMU.

As shown in Fig. 5.7, function `decompress_kernel` in line 16 is invoked to decompress the kernel. Its first parameter (i.e., `output_start`) indicates the start address of the decompressed kernel. Thus, if we can identify when `decompress_kernel` is invoked, we can get the first parameter by checking the machine register (`R0` in ARM) and dump the decompressed Linux kernel.

We notice that the function *decompress_kernel* is invoked by the assembly code in *arch/arm/boot/compressed/head.S*. We observe that this snippet of assembly code remains unchanged in different kernel versions. With this observation, we identify the address of instruction `BL decompress_kernel` by strictly comparing the execution trace of QEMU and the hard coded assembly code. After finding the instruction, we can obtain the address of the function *decompress_kernel* and the value of *output_start* according to the execution trace. With this information, we can dump the decompressed Linux kernel after the function *decompress_kernel* returns.

By doing so, we can automatically retrieve decompressed Linux kernels from firmware images.

5.4.2 Identity ECMO Pointers

Our system needs to obtain the addresses of two essential types of functions in the Linux kernel. Specifically, the *ECMO Forward Pointers* contain the functions that are used by the Linux kernel to initialize device drivers. We dynamically hook and redirect them to *ECMO Drivers* at runtime in QEMU. The *ECMO Backward Pointers* contain the APIs that are used by the *ECMO Driver* to invoke functions provided by the Linux kernel to interact with emulated peripherals in QEMU.

Precisely identifying ECMO Pointers is not easy. The main challenge is the decompressed Linux kernel is stripped and only contains the binary data. It has neither meaningful headers nor debugging symbols and contains thousands of functions. Furthermore, the Linux kernel is compiled with different compilers and compiling options, which can result in different binaries. Thus, we cannot have any assumption on the compiling options or compilers. We also cannot rely on run-time symbol tables like `/proc/kallsym` because they are only available *after* booting. However, we have the insight that embedded Linux kernels are usually modified based on the mainstream Linux kernel and the modification mainly aims to add support for specific peripherals with *board-specific code*. Meanwhile, ECMO Pointers are functions in *architecture independent code* or *architecture dependent code* (Section 5.2.1), which is unchanged and open-source.

In this case, we can automatically identify ECMO Pointers by leveraging the source code of the mainline Linux kernel. For instance, if we find that a function uses a specific string by reading the source code, then we can easily identify this function inside the binary by locating the function that has references to the same string. Of course, this simple strategy may not always work, since some functions do not have such obvious patterns or multiple functions can refer to the same string. Thus, we take three different strategies to identify ECMO Pointers (Section 5.4.2).

We illustrate each step in the following.

Disassemble the Linux Kernel

The first step is to disassemble the Linux kernel for further analysis, including constructing the control flow graph and identifying function boundaries. Accurately disassembling the ARM binaries is still challenging, especially when the binary is stripped [92]. This is because inline data is very common in ARM binaries and there are two different instruction sets (i.e., ARM and Thumb). Furthermore, ARM does not have a distinguished function call instruction, which can influence the accuracy of identifying function boundaries. In this case, we choose to ensure that this step does not introduce false negatives, i.e., all the code sections should be dissembled. Otherwise, we cannot identify the functions if they are not correctly disassembled. However, we can tolerate the false positives, i.e., the inline data may be wrongly disassembled as code. The strategies described in Section 5.4.2 can help us to filter out these false positives.

After disassembling the Linux kernel and constructing the control flow graphs, we further locate function boundaries by combining the algorithm introduced in Nucleus [72] and angr [3]. Nucleus can identify the functions indirectly called while angr locates the function according to the prologue. These two tools can help to reduce the false negatives and guarantee that the required function addresses (ECMO Pointers) will be located during the disassembly process. Finally, we build a mapping for each function and various types of information, e.g., number of basic blocks, string references, number of called functions and etc. This mapping describes the signature (or portrait) of each function. Note that, our system does not require that the constructed control flow graphs are sound or complete, as long as they can provide enough information for further analysis (Section 5.4.2).

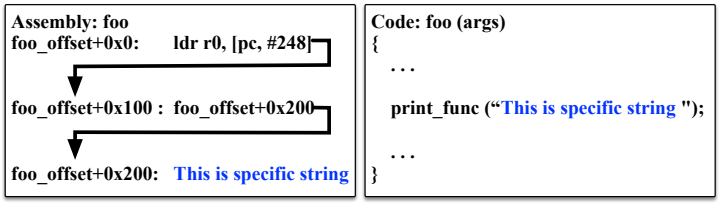
Identify Pointer Addresses

Algorithm 2: The algorithm to identify the addresses of ECMO pointers from the Linux kernel binary.

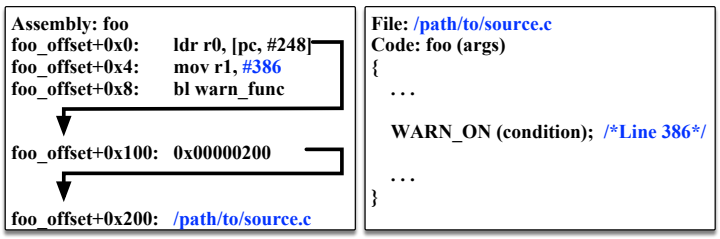
```
   Input: The decompressed Linux kernel LKB;
   The source code of ECMO Pointers SC (architecture independent code or architecture
   dependent code);
   Output: The addresses of ECMO Pointers FA;
1  Function Identify(LKB,SC):
2      CFG = Disassembly(LKB)
3      Generated_Functions = GenerateFunctions(CFG)
4      for S_F in SC do
5          for G_F in Generated_Functions do
6              for Filtering_Strategy in Filtering_strategies do
7                  if Filtering_Strategy(S_F,G_F) then
8                      Append G_F to S_F.Candidates
9                  end
10             end
11         end
12     end
13     for S_F in SC do
14         if Length(S_F.Candidates) == 1 then
15             FA[S_F] = S_F.Candidates
16         end
17     end
18     return FA
19
```

Algorithm 2 describes the process to locate pointer addresses of ECMO Pointers in the decompressed Linux kernel binary, i.e., *LKB*. Note that, we first need to get the source code of the functions, i.e., *SC*, inside the mainline Linux kernel. The outputs of this algorithm are the addresses of ECMO Pointers, i.e., *FA* (line 12).

First, we disassemble the decompressed Linux kernel, construct the control flow graph (line 2) and generate function boundaries (line 3). Then for the source code function of each ECMO Pointer (line 4), we loop through the generated functions (line 5) and apply different filtering strategies (line 6). If one filtering strategy can identify one address as a candidate address of the ECMO Pointer (line 7), this address will be appended to the candidate list (line 8). Finally, we check the candidates of each ECMO Pointer (line 9). If there is only one candidate (line 10), it means the



(a) Specific constant string: the constant string is referenced by a data pointer (i.e., foo_offset+0x200).



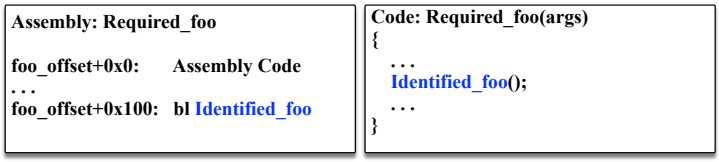
(b) Warning information: line number (i.e., 386) is the operand of assembly code; file name (i.e., /path/to/source.c) is a constant string.

Figure 5.8: Strategy-I: Lexical information

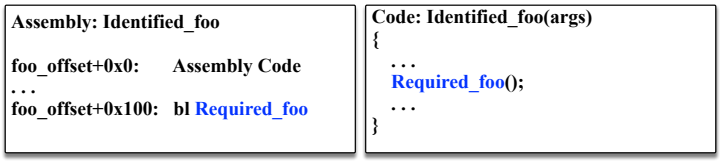
address of this ECMO Pointer is successfully identified in the kernel binary (line 11). Note that even if there is more than one candidate for each ECMO Pointer, ECMO can automatically try all the candidates and the one that can rehost the Linux kernel should be the right one. We do not find such cases in our experiments.

Strategy-I: Lexical information. The first strategy uses the lexical information inside a function as its signature, e.g., a specific constant string and the warning information. If the function we want to identify has such strings, we can then lookup the disassembly code to find the functions that have data references to the same string. The line number and file name in the warning information can further help to locate the function.

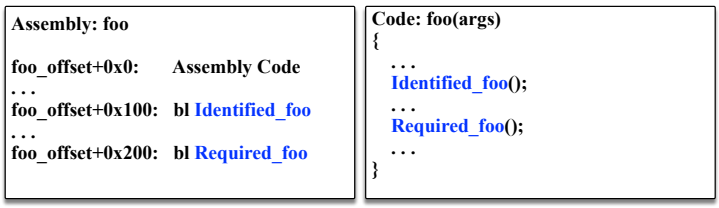
Fig. 5.8a shows a pair of the disassembled code and the source code in the mainline Linux kernel. In the source code, the function `foo` contains a specific constant string "*This is a specific string*". In the assembly code, the instruction at `foo_offset+0x0`



(a) Caller relationship: Required_foo is the caller of Identified_foo.



(b) Callee relationship: Required_foo is the callee of Identified_foo.



(c) Sibling relationship: Required_foo and Identified_foo are both called by foo.

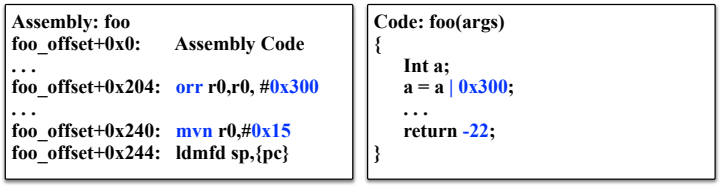
Figure 5.9: Strategy-II: Function relationship

will load the data pointers (i.e., `foo_offset+0x100`) using the `LDR` instruction. The data pointer refers to another pointer (i.e., `foo_offset+0x200`), which contains the same constant string. Based on this, we can locate function `foo` in the disassembled kernel. Fig. 5.8b shows a similar example with the warning information. The WARN_ON will call function *warn_func*. The first parameter is the filename, which is a specific constant string. The second parameter is the line number of WARN_ON. Usually, the line number is hard coded as an operand of instruction after compilation. Thus, functions containing specific constant strings or warning information can be easily identified.

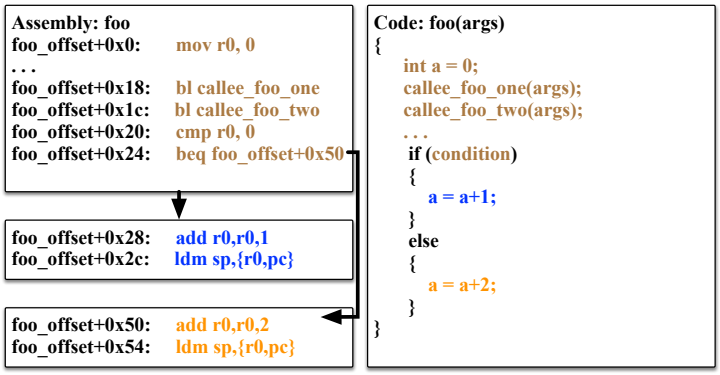
Strategy-II: Function relationship. The second strategy uses the relationship

between functions. That's because functions that do not contain specific strings cannot be identified by the strategy-I. However, we can use the relationship between the functions we want to identify and the ones that have been identified using the previous strategy. For instance, if we have identified the function (`Identified_foo`) and this function is *only* invoked by the function `Required_foo`, then we can easily locate the `Required_foo` by finding the caller of the `Identified_foo` function (Figure 5.9a). Similar strategies can be applied to the callee and sibling relationship, as shown in Figure 5.9b and Figure 5.9c, respectively. Note that we do not need to have a precise call graph, which is hard to generate due to the indirect call and inline function. This is because strategy I can identify several functions due to the many specific constant strings in the Linux kernel. Only if one of the functions identified by Strategy I (`Identified_foo`) has certain function relationships with the target function (`Required_foo`), strategy II can work. We do not encounter this issue in our experiments. With the help of function relationship, we can identify the functions indirectly.

Strategy-III: : Function structure. If one function has more than one caller, callee or sibling, it cannot be located solely using the function relationship. The third strategy takes the function structure, including logic or arithmetic operations, return value, the number of basic blocks, and the number of callee functions. Fig. 5.10a shows the example that the function performs the logic operation on some specific values (i.e., `a = a|0x300`) and return a specific value (i.e., -22) , the compiler will generate the instructions that contain the specific values (e.g., `orr r0,r0, #0x300`, `mvn r0,#0x15`). Besides, the callee number and basic block number will also be considered to filter out the candidate. Fig. 5.10b shows that function *foo* has two callees (i.e., *callee_foo_one* and *callee_foo_two*), which map to two instructions at foo_offset+0x18 and foo_offset+0x1c. Basic block number works with the same



(a) Logic operation: The constants (i.e., 0x300, -22) of logic operation or return value in source code map to the operands in assembly code.



(b) Callee Number: The two callee functions (i.e., callee_foo_one, callee_foo_two) map to the two bl instruction at offset foo_offset+0x18 and foo_offset+0x1c. Basic Block Number: The three basic blocks in source code maps to three basic blocks in assembly code.

Figure 5.10: Strategy-III: Function structure

rule.

Summary: With the above three strategies, we can automatically and successfully identify ECMO Pointers for all the Linux kernels (815 ones in 20 kernel versions) used in the evaluation (Section 5.5.2).

5.4.3 Generate ECMO Drivers

The process to generate *ECMO Drivers* is similar with developing a kernel module. However, we need to make the driver self-contained as much as possible and invoke the APIs in the Linux kernel through *ECMO Backward Pointers*. In particular, we compile the source code into an object file (i.e., `ECMO_Driver.o`). To make this

```
1  0x10000: ldr r3, [pc, #72]
2  0x10004: blx r3
3  0x10050: "Pointer value of called function"
```

Figure 5.11: `ECMO Driver` indirectly invokes functions in Linux kernel. In offset 0x10000, the memory address pointed by [pc, #72] is 0x10000 + 8 + 72 = 0x10050. In this case, functions with arbitrary address can be invoked.

driver work, we need to setup the base address and fix up the function calls to *ECMO Backward Pointers*. Moreover, we need to ensure that this driver does not occupy the physical memory region that the kernel can perceive, which is achieved by allocating the opaque memory.

Fixup the driver. Note that the compiled object file's base address is `0x0`. Given a new load address at runtime, our system calculates new values of the data pointers and function pointers and automatically rewrites the corresponding values in the driver.

Furthermore, due to the limitation of the jump range for the `BL Label` instruction, the driver may not be able to invoke the functions (*ECMO Backward Pointers*) in the original Linux kernel with direct calls, if the offset between them is far from the range of the `BL` instruction. To make it work, we rewrite the direct calls with indirect calls. For example, Fig. 5.11 shows a code snippet of the assembly code. At the offset `0x10000`, it loads the value stored at the offset `0x10050` into the register `R3`, which is the jump target. We can rewrite the value in the offset `0x10050` to invoke arbitrary function (*ECMO Backward Pointers*) in the Linux kernel, without being limited by the direct call.

Allocate the opaque memory. The *ECMO Driver* is loaded into the memory for execution. However, if we directly inject the driver into the free physical memory pages, the pages could be allocated for other purposes. This is because the kernel does not explicitly know the existence of the *ECMO Driver* and it is hard to change the allocated physical memory pages due to the complex memory management strategy of

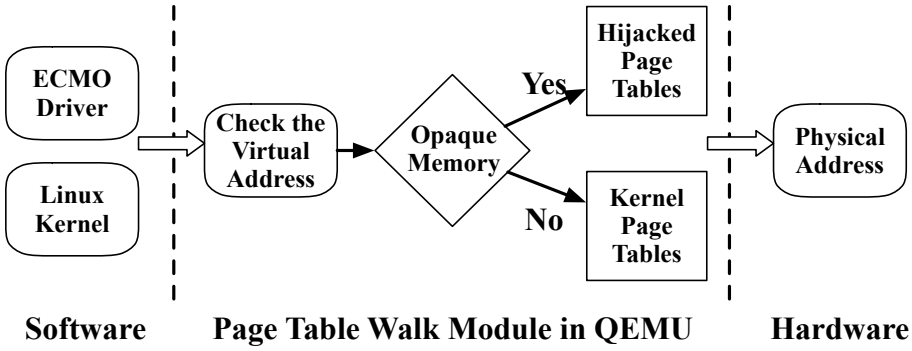


Figure 5.12: The overall design of opaque memory.

Linux kernel. In this case, the ECMO driver may be overwritten and the functionality cannot be guaranteed. Thus, we need to ensure that the driver should reside inside a memory region that cannot be affected by the Linux kernel.

To solve this problem, we propose the concept of *opaque memory*, a memory region that is not perceived by the Linux kernel but can be used at runtime. We implement the opaque memory by hooking the emulated MMU in QEMU. Fig. 5.12 shows how opaque memory works. Specifically, the emulated MMU walks through the page table to translate virtual addresses to physical addresses. ECMO changes the MMU module in QEMU to check whether the virtual address being translated is in the region of the opaque memory. If so, it will walk through our hijacked page table for the opaque memory to get the physical address. Otherwise, the original kernel page tables will be used. We ensure that the virtual address in the opaque memory always has a valid entry in the page table. By doing so, the *ECMO Driver* can be loaded and executed in the opaque memory, without affecting the memory view of the rehosted Linux kernel. By default, we set the opaque memory starting from 0xd0008000 and the length is 0x10000. Meanwhile, we check whether the address conflicts with the one allocated by Linux kernel. If so, we will change the start address.

5.4.4 Implementation Details

We implement ECMO based on LuaQEMU [30]. LuaQEMU is a dynamic analysis framework based on QEMU and it exposes several QEMU-internal APIs to LuaJIT [29] core, which is injected into QEMU. We port LuaQEMU based on old QEMU (version 2.9.50) to support the QEMU in new version (4.0.0) and expose more designated APIs for initializing the peripheral models. With LuaQEMU, we are able to hijack the execution process of rehosted Linux kernel at runtime and manipulate the machine states, e.g., accessing registers and memory regions, through Lua scripts, at specified breakpoints. For example, we can specify a breakpoint at any particular address. Inside the breakpoint, we can execute our own Lua script for different purposes. This eases the implementation of the opaque memory, dumping the decompressed Linux kernel, and installing the ECMO Pointers.

The module to identify ECMO Pointers (Section 5.4.2) is implemented in Python. We utilize Capstone [13] to disassemble the decompressed Linux kernel. For the function identification, we re-implement the algorithm described in Nucleus [72] and angr [3] in Python. We further extract the required function information, which is the function signature based on the generated functions and their control flow graphs. Finally, we integrate all these code with our strategies for identifying ECMO Pointers, which takes 2290 lines of Python code. All the above mentioned procedures can be done automatically except that the *ECMO Driver*, which consists of the drivers of transplanted peripherals. It is developed using the C language manually, which takes less than 600 lines of code, and cross-compiled by GCC. Note that it is a one-time effort to develop the *ECMO Driver* (Section 5.6). One *ECMO Driver* can be used by different Linux kernel versions if the related functions and structures are not changed.

5.5 Evaluation

In this section, we present the evaluation result of our system. Note that, the main purpose of our work is to rehost Linux kernels in QEMU so that we can build different dynamic analysis applications and install drivers for more peripherals. In the following, we first introduce the dataset of firmware images used in the evaluation and then answer the following research questions.

- **RQ1:** Is ECMO able to identify ECMO Pointers?
- **RQ2:** Is ECMO able to rehost the Linux kernels of embedded devices with different kernel versions and device models?
- **RQ3:** Are the rehosted Linux kernels stable and reliable?
- **RQ4:** Can ECMO support more peripherals and be used to develop dynamic analysis applications?

5.5.1 Dataset

As our system targets embedded Linux kernels, we have collected the firmware images from both third-party projects (i.e., OpenWRT [34]) and device vendors (i.e., Netgear [32]). Our evaluation targets Linux kernels in ARM devices, since they are the popular CPU architectures in embedded devices [41]. However, the overall methodology can also be applied to other architectures (e.g., MIPS).

During the experiment, we focuses on transplanting three early-boot peripherals, i.e., interrupt controller (IC), timer, and UART, which are required to boot a Linux kernel. Once the Linux kernel is rehosted, we can install different peripheral drivers to support other peripherals with kernel modules. Specifically, we use the PrimeCell Vectored Interrupt Controller (PL190) [37] and ARM Dual-Timer Module (SP804) [4]. We use the ns16550 UART device in our system. In total, we evalu-

Table 5.1: The ECMO Pointers, identification strategy, and the Linux kernel versions that the ECMO pointers used by.

Forward Pointers	Strategy	Kernel Version
mach_desc->init_irq	I	ALL
mach_desc->init_time	I	ALL
Backward Pointers	**Strategy**	**Kernel Version**
irq_set_chip_and_handler_name	III	3.18.x/4.4.x/4.14.x
irq_set_chip_data	III	ALL
handle_level_irq	II	ALL
__handle_domain_irq	III	3.18.x/4.4.x/4.14.x
setup_machine_fdt	I	3.18.x/4.4.x/4.14.x
set_handle_irq	III	3.18.x/4.4.x/4.14.x
irq_domain_add_simple	III	3.18.x/4.4.x/4.14.x
irq_create_mapping	I	3.18.x/4.4.x/4.14.x
of_find_node_by_path	II	3.18.x/4.4.x/4.14.x
setup_irq	I	ALL
clockevents_config_and_register	III	3.18.x/4.4.x/4.14.x
irq_domain_xlate_onetwocell	I	3.18.x/4.4.x/4.14.x
clockevent_delta2ns	I	2.6.x
clockevents_register_device	II	2.6.x
set_irq_flags	I	2.6.x/3.18.x
set_irq_chip	I	2.6.x
irq_to_desc	II	2.6.x
__do_div64	II	2.6.x
platform_device_register	I	ALL
lookup_machine_type	I	2.6.x
_set_irq_handler	I	2.6.x
irq_modify_status	III	4.4.x/4.14.x

ate 815 (720 in OpenWRT and 95 in Netgear) firmware images that contain Linux kernels.

5.5.2 Identify ECMO Pointers (RQ1)

ECMO Pointers are important to peripheral transplantation. In this section, we evaluate the success rate of identifying ECMO Pointers. Among all the 815 Linux kernels, there are 20 different kernel versions.

Table 5.1 lists the required ECMO Pointers, the strategies we used, and the Linux kernel versions that these ECMO Pointers are used. In total, we need to identify 24 different ECMO Pointers for all the 20 Linux kernel versions. Among them, two (i.e.,

Table 5.2: The decompressed Linux kernel size and the disassembled function numbers for our dataset.

	Maximum	Minimum	Mean	Median
Size (Bytes)	8,526,240	4,134,392	7,297,977	8,478,848
Functions (#)	48,412	18,455	29,910	23,872

`mach_desc->init_time`, and `mach_desc->init_irq`) are data pointers. Identifying the data pointers is rather more difficult than the function pointers as we need to identify symbols in each function and infer the right ones. Fortunately, these two data pointers are the return values of `setup_machine_fdt` and `lookup_machine_type`, respectively. According to the ARM calling convention, the return value is saved in register `R0`. In this case, we can identify these two data pointers by identifying function pointers `setup_machine_fdt` and `lookup_machine_type`.

Identifying ECMO Pointers requires us to disassemble the decompressed Linux kernel. Table 5.2 lists the information of these kernels. The decompressed Linux kernel is about 730k bytes on average, with thousands of functions. Among these functions, we successfully identify the required ECMO Pointers for all Linux kernels.

Table 5.3: The overall result of ECMO on rehosting the Linux kernel of OpenWRT. "Downloaded Images" represents the number of downloaded images. "Format Supported" represents the number of images whose formats are supported by firmware extraction tool (i.e., Binwalk). "Kernel Extracted" represents the number of images extracted from the downloaded image, which are rehosted by ECMO. "Peripherals Transplanted" represents the number of the images that peripheral can be transplanted successfully (e.g., IC can handler the interrupt well). "Ramfs are not Mounted" represents the number of images that cannot mount the given ramfs. "Shell" represents the images that we can rehost and spawn a shell. Success Rate of Transplantation = (Peripherals Transplanted)/(Images); Success Rate of Rehosting = (Shell)/(Images).

Kernel Version	Downloaded Images	Format Supported	Kernel Extracted	Peripherals Transplanted	Success Rate of Transplantation	Ramfs are not Mounted	Shell	Success Rate of Rehosting
3.18.20	23	23	21	21	100%	8	13	61.9%
3.18.23	29	29	29	29	100%	8	21	72.4%
4.4.42	37	37	37	37	100%	8	29	78.4%
4.4.47	37	37	37	37	100%	8	29	78.4%
4.4.50	45	45	45	45	100%	16	29	64.4%
4.4.61	39	39	37	37	100%	8	29	78.4%
4.4.71	40	40	38	38	100%	8	30	78.9%
4.4.89	40	40	38	38	100%	8	30	78.9%
4.4.92	41	41	38	38	100%	8	30	78.9%
4.4.140	41	41	38	38	100%	8	30	78.9%
4.4.153	40	38	38	38	100%	8	30	78.9%
4.4.182	40	38	38	38	100%	8	30	78.9%
4.14.54	54	54	42	42	100%	0	42	100%
4.14.63	66	66	42	42	100%	0	42	100%
4.14.95	66	66	42	42	100%	0	42	100%
4.14.128	66	66	42	42	100%	0	42	100%
4.14.131	66	66	42	42	100%	0	42	100%
4.14.151	66	66	42	42	100%	0	42	100%
4.14.162	66	66	42	42	100%	0	42	100%
Overall	902	898	720	720	100%	96	624	86.7%

Answer to RQ1: ECMO can identify all the required ECMO Pointers from thousands of functions inside decompressed Linux kernel.

5.5.3 Rehost Linux Kernels (RQ2)

In this section, we evaluate the capabilities of ECMO on rehosting the Linux kernels. During this process, we use our system to boot the kernel and provide a root file system (rootfs) in the format of ramfs. We use our own rootfs because we can include different benchmark applications into the rootfs to conduct security analysis. For example, we include PoCs of kernel exploits to conduct the root cause analysis (Section 5.5.5). Furthermore, we can include different peripheral drivers to support more peripherals. The rootfs extracted from the firmware image can also be used.

Firmware Images from Third Party Projects

Table 5.3 shows the overall result and the success rate of peripheral transplantation and kernel rehosting for OpenWRT. We define the success of peripheral transplantation as that the transplanted IC, timer and UART devices function well in the kernel. If the rehosted kernel enters into the user-space and spawns a shell, we treat it as a successful kernel rehosting. In total, we download 902 firmware images from OpenWRT. However, four images' formats are not supported by Binwalk and the Linux kernel cannot be extracted (if there is). For the left 898 firmware images, 720 of them contain Linux kernels while the left ones contain only user-level applications. The 720 ones will be evaluated by ECMO.

Linux Kernel Versions. The kernels in the 720 OpenWRT firmware images consist of 19 different kernel versions. Our evaluation shows that we can transplant the peripherals for all the 720 Linux kernels. However, some Linux kernels cannot be booted. This is because they cannot recognize our pre-built root file system (in the ramfs file format) as the support of ramfs is not enabled when being built. Without

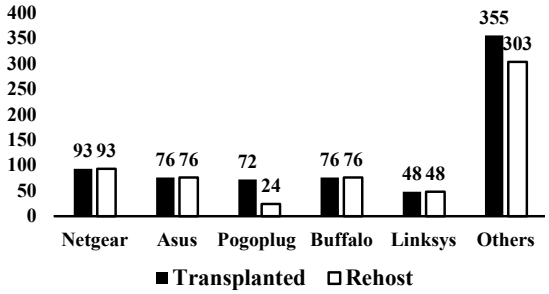


Figure 5.13: Vendor Distribution of Linux Kernels.

Table 5.4: The overall result of ECMO on rehosting the Linux kernel of Netgear Devices.

Device Name	Kernel Version	Images	# of Peripherals Transplanted	Shell
R6250	2.6.36	21	21	15
R6300v2	2.6.36	22	22	19
R6400	2.6.36	20	20	20
R6700	2.6.36	16	16	16
R6900	2.6.36	16	16	16
Overall	-	95	95	86

the root file system, we cannot launch the shell. However, all of them enter into the function (i.e., `init_post`) to execute the `init` program. In summary, among 720 kernels, our system can rehost 624 of them, which is shown in Table 5.3.

Vendors and Device Models. As the OpenWRT project supports devices from multiple vendors, we calculate the supported vendors and there are 24 different vendors. Figure 5.13 shows the result of the top five vendors, i.e., `Netgear`, `Asus`, `Pogoplug`, `Buffalo`, and `Linksys`, in the OpenWRT dataset. Among them, `Pogoplug` has a relatively low success rate of rehosting. That's because most kernels from that vendor cannot recognize our pre-built root file system. We also count the number of device models for the successfully rehosted Linux kernels. In total, 32 device models are identified.

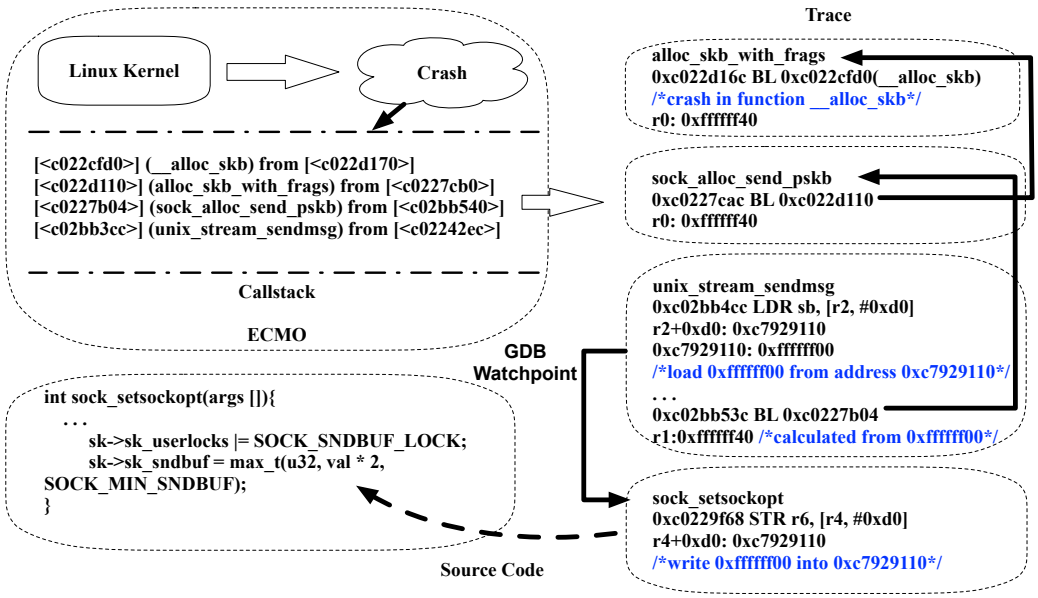


Figure 5.14: Root cause analysis of CVE-2016-9793.

Firmware Images from Official Vendors

Besides third-party firmware images, we also apply ECMO on the official images released by `Netgear`. We collect the firmware images for five popular devices, including R6250, R6300v2, R6400, R6700, R6900, from the vendor's website [32]. In total, we manage to collect 95 firmware images, and the latest one is released on 2020-09-30. Table 5.4 shows the result. We noticed that all the Linux kernels of these devices are in the version 2.6.36. We can successfully transplant the peripherals to all the 95 different firmware images. Among them, we can launch the shell for 86 images while the left 9 cannot be rehosted due to the same root file system problem.

Answer to RQ2: ECMO can rehost the Linux kernel of embedded devices from 20 kernel versions and 37 (32 in OpenWRT and 5 in Netgear) device models. Peripherals can be transplanted to all the Linux kernels while 87.1% (710/815) Linux kernels can be successfully rehosted (i.e., launch the shell).

Table 5.5: The category of the failed syscall test cases.

Category of Failed cases	Number
Testing the bug or vulnerability of Linux kernel	16
Network is not enabled	15
The function is not implemented	25
Others	10
Total	66

5.5.4 Reliability and Stability (RQ3)

We use the LTP (Linux Test Project [27]) testsuite to evaluate the reliability and stability of the rehosted kernel. In total, there are $1,257$ test cases for system calls. Among them, 148 are skipped as the testing environment (e.g., the CPU architecture and the build configuration) does not meet the requirement. For the left $1,109$ test cases, $1,043$ passed while the left 66 ones failed.

We further analyze the reason for the failed test cases. Table 5.5 lists the category of the reason. Among them, 15 cases are due to the lack of network devices. This is expected since our system does not add the support of network device initially. However, all the 15 test cases are passed after installing the Ethernet device driver with kernel modules on the rehosted Linux kernel (Section 5.5.5). Also, 16 cases aim to test whether the Linux kernel fixes a bug or vulnerability. For instance, the test case (timer_create03 [28]) is to check whether CVE-2017-18344 [16] is fixed. If the vulnerability is not fixed, the test case will fail. They are also expected since the testing kernel does not fix these vulnerabilities. The other 25 cases return back the `ENOSYS` error number, which means the functionalities are not implemented. For the remaining 10 cases, the reason is adhoc, such as the kernel version is old and timeout.

In summary, 94% of the system call test cases passed. This evaluation shows the rehosted kernel is reliable and stable. We further demonstrate the usage scenarios of the rehosted Linux kernel in Section 5.5.5.

Answer to RQ3: The rehosted Linux kernel can pass 94% system call test cases in LTP, which demonstrates its reliability and stability.

5.5.5 Applications and Other Peripherals (RQ4)

Our system can rehost Linux kernels, which provides the capability to install different peripheral drivers with kernel modules to support more peripherals. Furthermore, the rehosted Linux kernel lays the foundation of applications relying on the capability to introspect the runtime states of the target system. In this section, we successfully install the Ethernet device driver (i.e., smc91x) for all the rehosted Linux kernels. We also leverage our system to build three applications, including kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, to demonstrate the usage scenarios of ECMO. Other applications that rely on QEMU can be ported. Note that, we only use these applications to demonstrate the usage of our system. The applications are not the main contribution of

Other Peripherals

Linux kernel module is an object file that can be loaded during the runtime to extend the functionality of the Linux kernel. In this case, peripheral drivers can be built as kernel modules and loaded into the kernel dynamically. To demonstrate that our rehosted Linux kernel is able to support more peripherals. we select one rather complex peripheral (i.e., smc91x [42]) and build the driver code into kernel module (i.e., smc91x.ko). We then inject this kernel module into the ramfs that is fed to rehosted Linux kernel. After the embedded Linux kernel is rehosted by ECMO, we use the command *insmod smc91x.ko* to install the peripheral driver for smc91x. Meanwhile, QEMU has already provided the peripheral model for smc91x and we can integrate this model into the machine model directly. Finally, we successfully install the peripheral driver of smc91x for all the 710 rehosted Linux kernels, which

Table 5.6: CVEs that can be triggered on the rehosted Linux kernel by ECMO.

CVE ID	CVE Score	CVE Type	Fix Version
CVE-2018-5333	5.5	Null Pointer Dereference	4.14.13
CVE-2016-4557	7.8	Double Free	4.5.5
CVE-2017-10661	7.0	Race Condition	4.10.15
CVE-2016-0728	7.8	Interger Overflow	4.4.1
CVE-2016-9793	7.8	Type Confusion	4.8.14
CVE-2017-12193	5.5	Null Pointer Dereference	4.13.11

demonstrate the capability of ECMO to support the other peripherals.

Crash Analysis

In the following, we show the process to utilize ECMO to understand the root cause of the crash on rehosted kernels.

To this end, we collect the PoCs that can trigger the crash for six reported bugs and vulnerabilities (as shown in Table 5.6). We then boot the Linux kernel and run the PoCs to crash the kernel. During this process, we use the QEMU to collect the runtime trace. We also leverage the remote GDB in QEMU to debug the rehosted kernel. We detail the procedures on how to conduct the crash analysis for one case (CVE-2016-9793 [15]) with the collected runtime trace. Figure 5.14 shows the whole procedure.

Specifically, when the rehosted Linux kernel crashes, the detailed call stack will be printed out. The call stack includes the function name and the addresses of these functions. With the runtime trace provided by QEMU, we can get the information including the register values and the execution path. By analyzing the trace, we noticed that a negative value (i.e., `0xffffff40`) is the first parameter of the function `__alloc_skb`. This negative value results in the crash.

We then analyze the propagation of this negative value within the trace. This value is propagated by the first parameter of the function `sock_alloc_send_pskb`. Finally, we notice that the negative value `0xffffff40` is calculated from `0xffffff00`,

which is loaded by the function `unix_stream_sendmsg` from the address `0xc7929110`. We then use the `GDB` to set a watchpoint at this memory address and capture that the instruction at the address `0xc0229f68` was writing the negative value (i.e., `0xffffff00`) into this memory location.

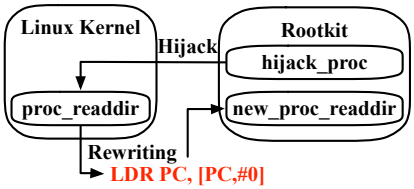
We further analyze the function that contains the instruction at the address `0xc0229f68`. It turns out that the root cause of the crash is because of the type confusion. In the function `sock_setsockopt`, the variable `sk→sk_sndbuf` will be set by the return value of `max_t` (maximum value between two values in the same type). However, due to the wrong type `u32`, the return value can be a negative value, which triggers the crash.

This analysis shows the usage of ECMO by providing the capability introspect the runtime states of the rehosted kernel.

Rootkit Forensic Analysis

Rootkit forensic analysis requires the ability to monitor the runtime states of the kernel [82, 94]. We demonstrate this ability by conducting the rootkit forensic analysis with one (i.e., *Suterusu* [45]) popular rootkit in the wild.

Specifically, `Suterusu` is able to hide specific processes by hijacking the kernel function `proc_readdir`, which is used to get the process information. As shown in Figure 5.15a, it hijacks the function `proc_readdir` by rewriting the function's first instruction to `LDR PC,[PC,#0]`. As a result it redirects the execution to the function *new_proc_readdir* inside the rootkit. With ECMO, we can monitor the changes to the kernel code sections (a suspicious behavior) by setting up memory watchpoints to the Linux code section (Figure 5.15b).



(a) Workflow of rootkit *Suterusu*

```
gef >  c
Continuing.

Hardware watchpoint 1: *0xc00fc078

Old value  = 0xe92d4038
New value = 0xe59ff000
0xbf00116c in ?? ()     LDR PC,[PC, #0]
```

(b) ECMO observes how the rootkit *Suterusu* works.

Figure 5.15: The workflow of rootkit *Suterusu* and how ECMO analyzes the behavior

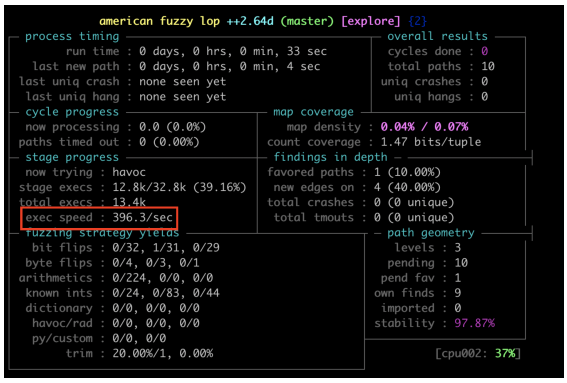


Figure 5.16: `UnicornFuzz` can be run on the rehosted Linux kernel

Fuzzing

Fuzzing has been widely used to detect software vulnerabilities. We ported one of the most popular kernel fuzzers (i.e.,`UnicornFuzz` [109]) into ECMO and fuzzed the example kernel modules provided by `UnicornFuzz`. As shown in Fig. 5.16, `UnicornFuzz` can work under ECMO and the fuzzing speed can reach to 396 instances per second. This demonstrates the usage of ECMO for kernel fuzzing.

Answer to RQ4: Applications, e.g., crash analysis, forensic analysis, kernel fuzzing, can be built upon the rehosted Linux kernel by our system.

5.6 Discussion

Manual efforts. ECMO provides mostly automated approach and only developing the *ECMO Driver* requires manual efforts. However, this is a one-time effort. Furthermore, one *ECMO Driver* can be transplanted to different kernel versions if the related functions and structures are not changed. Even if the functions are changed, we just need to change a few APIs and compile it again to create a new *ECMO Driver*. For example, the 815 Linux kernels consist of 20 different kernel versions. For the kernel in version 2.6.36, it takes 385 lines of C code. This driver can be used for all the kernel images of Netgear (Table 4). For the kernel in version 3.18.20 and 3.18.23, it takes 534 lines of C code while 180 lines of new code are added. For kernels in all the left 17 versions, they share the same driver code. 60 lines of new code are added compared with the one used in 3.18.20. Note that the driver code for the transplanted peripherals does not need to be developed. Instead, we reuse the existing code. For example, the driver code for VIC (PL190) is open source [38]. Thus, we just reuse the existing driver code, merge the driver code into one file, and finally compile it to generate the ECMO driver. In total, it takes less than one person-hour to build a new customized driver.

Functionality of peripherals. We successfully boot the Linux kernel by transplanting designated peripherals (e.g., IC, Timer, and UART). We admit that the original peripherals may not work property as they are not emulated (or transplanted) in QEMU. However, the functionalities of the transplanted peripherals are guaranteed. With the transplanted peripherals, ECMO can provide the capability to introspect the runtime states of the Linux kernel that dynamic analysis applications can be built upon. Without our system, it's impossible to build such applications since the target Linux kernel cannot be booted in QEMU. The three applications used in the evaluation have demonstrated the usage scenarios of our system. We

may build or port more complicated applications, e.g., dynamic taint analysis [134], to further evaluate our system.

Other Peripherals. Currently, ECMO is evaluated based on transplanting three early-boot peripherals (i.e., IC, timer, and UART) as they are required to boot a Linux kernel. In general, peripheral transplantation works on all kinds of peripherals. The transplanting process depends on the identification of ECMO pointers. Fortunately, to support the other peripherals, users can install the kernel modules directly on the rehosted Linux kernel, which does not need to identify pointers. In this case, all kinds of peripherals can be supported. Our experiments show that the driver of Ethernet device, which is rather complex, can be successfully installed and the network functionality can be guaranteed.

Other architectures Currently, ECMO only supports ARM architecture, which is the most popular one in embedded systems [41]. However, the technique peripheral transplantation can be easily extended to the other architecture as it does not rely on any particular architecture feature. Specifically, developers need to implement the module for identifying ECMO Pointers for the new architecture. This requires additional engineering efforts and algorithm 2 is provided.

5.7 Summary

In this work, we propose a novel technique named peripheral transplantation to rehost the Linux kernel of embedded devices in QEMU. This lays the foundation for applications that rely on the capability of runtime state introspection. We have implemented this technique inside a prototype system called ECMO and applied it to 815 firmware images, which consist of 20 kernel versions and 37 device models. ECMO can successfully transplant peripherals for Linux kernels in all images. Among them, 710 kernels can be successfully rehosted, i.e., launching the user-space shell

(87.1% success rate). Furthermore, we successfully install one Ethernet device driver (i.e., smc91x) on all the rehosted Linux kernels to demonstrate the capability of ECMO to support more peripherals. We further build three applications to show the usage scenarios of ECMO.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we conduct the empirical study on ARM disassembly tools, and propose two powerful tool named EXAMINER and ECMO, to lay the foundation of the firmware analysis.

Our empirical study reveals the observations that have not been systematically summarized and/or confirmed before and explore the implementation bugs of the state-of-the-art ARM disassembly tools. Furthermore, our study sheds light on the limitations of state-of-the-art disassembly tools and points out the potential directions for improvement. With this study, the static firmware analysis framework can be more accurate and robust.

To enhance the dynamic firmware analysis frameworks, which usually based on state-of-the-art emulators (i.e., QEMU). We propose EXAMINER, which can automatically locate the inconsistent instructions between emulators and hardware devices. With EXAMINER, we are able to locate 12 different bugs of three emulators (i.e., QEMU, Unicorn, and Angr), which cover commonly used instructions (e.g., BLX). All of these bugs are confirmed by the developers. Furthermore, we locate a huge number of inconsistent instructions and demonstrate the capability of these instructions on detecting emulators, anti-emulation, and anti-fuzzing. With our findings,

the dynamic firmware analysis frameworks can be more stable and reliable.

To further increase the scalability and capability of the state-of-the-art dynamic firmware analysis platforms, we propose a technique named peripheral transplantation and implement it inside a prototype system called ECMO. ECMO aims to transplant the device drivers of designated peripherals into the Linux kernel and replace the peripherals of Linux kernel that are not supported by QEMU with supported ones. Our evaluation demonstrate that ECMO can successfully rehost 710 different firmware images out of 815 ones, covering 20 kernel versions, 37 device models, and 24 vendors. With ECMO, the Linux kernel of embedded devices can be rehosted and different applications (e.g., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing) can be conducted, boosting the capability of the state-of-the-art dynamic firmware analysis platforms.

Overall, our works are helpful to nearly all the dynamic and static analysis tools. Almost all the static analysis tools rely on the disassembly technique and may reuse the state-of-the-art disassemblers, which are our study targets. With our study, the accuracy and reliability of these static analysis tools can be enhanced. The same impact also applies to the dynamic analysis tools. Most of the dynamic analysis tools are based on the state-of-the-art emulators. Our work successfully find the bugs of the emulators and reveal the fact that there are many inconsistent instructions between emulators and real devices. This can make the dynamic analysis tools more accurate and robust. Furthermore, our peripheral transplantation technique and the proposed system ECMO can enhance the scalability of the current dynamic analysis tools and boost the capability of analyzing embedded Linux kernels. Thus, the works proposed in this thesis can impact the field of general program analysis.

6.2 Future Work

In the future, we aim to explore the following directions. First, we would like to design new algorithms for disassembling ARM binaries more accurately. Based on the implication from our empirical study, ARM disassembly tools do not have very good support on detecting the right instruction set and identifying function boundaries. Combining the static analysis and machine-learning based mechanism may help improve the accuracy. Furthermore, conflict analysis can be utilized to identify the usage of `BL label` instruction, which can greatly improve the result of identifying function boundary.

Second, we aim to improve the capability of EXAMINER on testing instructions in privileged environments. This can help us to test the reliability of virtual machines. Apart from this, testing instruction stream sequences will also be our future work. This is because emulators may have optimization strategy towards the specific instruction stream sequences, whether these optimizations may influence the emulation accuracy is unknown and deserve our testing. Furthermore, we aim to extend EXAMINER to support more emulators and architectures.

Third, we aim to improve ECMO so that it can support more peripherals. Rehosted Linux kernel not only lays the foundation for building different applications, but also provides us with the capability on installing kernel modules. Currently, we inject the other peripheral drivers (i.e., NIC) on the rehosted Linux kernel with the loadable kernel modules (.ko files). We aim to extend the supported peripherals and apply ECMO on analyzing the security issues of kernel drivers.

All of the above mentioned future works can complement our existing works and boost the capability and reliability of the state-of-the-art firmware analysis techniques.

References

[1] 64 bit juno r2 arm® development platform. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Juno%20r2%20datasheet.pdf.

[2] Android Open Source Project. https://source.android.com/.

[3] angr. https://angr.io/.

[4] Arm dual-timer module (sp804). https://developer.arm.com/documentation/ddi0271/d/.

[5] ARM Exploration tools. https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.

[6] ARM Mapping Symbols. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0474f/CHDGFCDI.html.

[7] Arm Mbed OS. https://www.mbed.com/en/.

[8] ARM SIMD Instructions. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-SIMD-/ARM-SIMD-instructions.

[9] ARM WFE Instruction. https://developer.arm.com/documentation/ddi0360/e/programmer-s-model/additional-instructions/wait-for-event-wfe.

[10] B, BL, BX, BLX, and BXJ. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/Cihfddaf.html.

[11] Binary Ninja: A New Kind Of Reversing Platform. https://binary.ninja/.

[12] Buildroot: Making Embedded Linux Easy. https://buildroot.org.

[13] Capstone: The Ultimate Disassembly. http://www.capstone-engine.org/.

[14] Clang: Documentation. https://clang.llvm.org/docs/CommandGuide/clang.html.

[15] CVE-2016-9793. https://nvd.nist.gov/vuln/detail/CVE-2016-9793.

[16] CVE-2017-18344. https://nvd.nist.gov/vuln/detail/CVE-2017-18344.

[17] DDoS attack that disrupted internet was largest of its kind in history, experts say'. https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet.

[18] Debian Popularity Contest. https://popcon.debian.org/by_inst.

[19] GCC: Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[20] Ghidra: A Software Reverse Engineering(SRE) Suite of Tools Developed by NSA. https://ghidra-sre.org/.

[21] Hopper Disassembler. https://www.hopperapp.com/.

[22] IDA Pro. https://www.hex-rays.com/products/ida/.

[23] Issues submitted to BAP. https://github.com/BinaryAnalysisPlatform/bap/issues/951.

[24] Issues submitted to Binary Ninja. https://github.com/Vector35/binaryninja-api/issues/1359.

[25] Issues submitted to Ghidra. https://github.com/NationalSecurityAgency/ghidra/issues/657.

[26] Issues submitted to Radare2. https://github.com/radareorg/radare2/issues/14223.

[27] Linux Test Project. http://linux-test-project.github.io/.

[28] Linux test project test case timer_create03. https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/syscalls/timer_create/timer_create03.c.

[29] Luajit. http://luajit.org/luajit.html.

[30] Luaqemu. https://github.com/Comsecuris/luaqemu.

[31] Mirai IoT botnet blamed for 'smashing Liberia off the internet'. https://www.theregister.com/2016/11/04/liberia_ddos/.

[32] Netgear. https://www.netgear.com/.

[33] Objdump - Display Information from Object Files. https://linux.die.net/man/1/objdump.

[34] OpenWRT. https://openwrt.org/.

[35] Panda.re. https://panda.re/.

[36] Paradyn Project. Dyninst: Putting the Performance in High Performance Computing. https://www.dyninst.org/.

[37] Primecell vectored interrupt controller (pl190). https://developer.arm.com/documentation/ddi0181/e/introduction/about-the-vic.

[38] Primecell vectored interrupt controller (pl190) source code. https://elixir.bootlin.com/linux/v3.18.20/source/drivers/irqchip/irq-vic.c#L445.

[39] Psutil. https://psutil.readthedocs.io.

[40] Radare2. https://rada.re/r/.

[41] The roadshow of arm. https://group.softbank/system/files/pdf/ir/presentations/2019/arm-roadshow-slides_q4fy2019_01_en.pdf.

[42] Smc91x source code. https://elixir.bootlin.com/linux/v3.18.20/source/drivers/irqchip/irq-vic.c#L445.

[43] Soc (system on a chip). https://openwrt.org/docs/techref/hardware/soc.

[44] Source code for iot botnet 'mirai' released. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/.

[45] Suterusu. https://github.com/mncoppola/suterusu.

[46] The FreeRTOS Kernel. https://www.freertos.org/.

[47] Triforceafl. https://github.com/nccgroup/TriforceAFL.

[48] Unicorn. https://www.unicorn-engine.org/.

[49] Vulnerability statistics of linux kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[50] Z3Prover. https://github.com/Z3Prover/z3.

[51] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 23th ACM Conference on Computer and Communications Security*, 2016.

[52] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[53] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[54] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE symposium on reliable distributed systems*, 2010.

[55] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23th USENIX Conference on Security Symposium*, 2014.

[56] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, Ahmad M Mustafa, Gbadebo Ayoade, Khaled Al-Naami, Latifur Khan, Kevin W Hamlen, Bhavani M Thuraisingham, Frederico Araujo, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Network and Distributed Systems Security Symposium*, 2018.

[57] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Conference on Usenix Annual Technical Conference*, 2005.

[58] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016.

[59] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

[60] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[61] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[62] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[63] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[64] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2-3):171–188, 2001.

[65] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[66] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[67] Andrei Costin, Jonas Zaddach, Aurelien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[68] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In

Proceedings of the 11th ACM Asia Conference on Computer and Communications Security, 2016.

[69] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *Proceedings of the 13th {USENIX} Symposium on Operating Systems Design and Implementation*, 2018.

[70] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23rd Symposium on Network and Distributed System Security*, 2016.

[71] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[72] Andriesse Dennis, Asia Slowinska, and Bos Herbert. Compiler-agnostic function detection in binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[73] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE symposium on security and privacy*, 2011.

[74] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. Qemu-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[75] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017.

[76] Michael J. Eager. Introduction to the DWARF Debugging Format. `http://www.dwarfstd.org/doc/DebuggingusingDWARF-2012.pdf`.

[77] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[78] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[79] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[80] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[81] Yangchun Fu and Zhiqiang Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Transactions on Information and System Security*, 2013.

[82] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Annual Network and Distributed System Security Symposium*, 2003.

[83] Xinyang Ge, Ben Niu, and Weidong Cui. Reverse debugging of kernel failures in deployed systems. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020.

[84] Daniel M German and Jesús M González-Barahona. An empirical study of the reuse of software licensed under the gnu general public license. In *IFIP International Conference on Open Source Systems*. Springer, 2009.

[85] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *Proceedings of the 28th {USENIX} Security Symposium*, 2019.

[86] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[87] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[88] Grant Hernandez, Farhaan Fowze, Tuba Yavuz, Kevin RB Butler, et al. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017.

[89] Anoirel Issa. Anti-virtual machines and emulations. *Journal in Computer Virology*, 2012.

[90] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011.

[91] Daehee Jang, Yunjong Jeong, Sungman Lee, Minjoon Park, Kuenhwan Kwak, Donguk Kim, and Brent Byunghoon Kang. Rethinking anti-emulation techniques for large-scale software deployment. *Computers & Security*, 2019.

[92] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[93] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based" out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[94] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security*, 2010.

[95] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *Proceedings of the 30th {USENIX} Security Symposium*, 2021.

[96] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment*, 2019.

[97] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *Proceedings of the 28th {USENIX} Security Symposium*, 2019.

[98] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, 2015.

[99] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[100] Nikos Karampatziakis. Static analysis of binary executables using structural svms. In *Proceedings of the 23rd Advances in Neural Information Processing Systems*, 2010.

[101] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Proceedings of the 2020 Annual Computer Security Applications Conference*, 2020.

[102] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 12th USENIX Security Symposium*, 2004.

[103] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in PDF readers and files. *Empirical Software Engineering*, 2018.

[104] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[105] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 2014.

[106] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 2015.

[107] Cătălin Valeriu Liţă, Doina Cosovan, and Dragoş Gavriluţ. Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in upa packers. *Journal of Computer Virology and Hacking Techniques*, 2018.

[108] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[109] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proceedings of the 13rd {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[110] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[111] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 19th international symposium on software testing and analysis*, 2010.

[112] Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, and Danilo Bruschi. A methodology for testing CPU emulators. *ACM Transactions on Software Engineering and Methodology*, 2013.

[113] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[114] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[115] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[116] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[117] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.

[118] Marius Muench, Dario Nisi, Aurelien Francillon, and Davide Balzarotti. Avatar2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research*, 2018.

[119] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.

[120] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[121] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[122] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[123] Rui 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*, 2017.

[124] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. idev: exploring and exploiting semantic deviations in arm instruction processing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[125] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Proceedings of the 2007 International Conference on Information Security*. Springer, 2007.

[126] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[127] Alastair Reid. Trustworthy specifications of arm® v8-a and v8-m system level architecture. In *Proceedings of the 16th Formal Methods in Computer-Aided Design*, 2016.

[128] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11st International Workshop on Recent Advances in Intrusion Detection*, 2008.

[129] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60, 2009.

[130] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, 2008.

[131] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 2008.

[132] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008.

[133] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[134] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE symposium on Security and privacy*, 2010.

[135] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002.

[136] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[137] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice: Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22th Annual Symposium on Network and Distributed System Security*, 2015.

[138] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[139] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[140] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[141] Kim Taegyu, Chung Hwan Kim, Choi Hongjun, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu less. Revarm: A platform-agnostic arm binary rewriter for security applications. In *Proceedings of the 37th Annual Computer Security Applications Conference*, 2017.

[142] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[143] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, 2016.

[144] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security*, 2017.

[145] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the mirror: evaluating iot device security through mobile companion apps. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[146] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11st International Workshop on Recent Advances in Intrusion Detection*, 2008.

[147] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2020.

[148] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[149] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017.

[150] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[151] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks*, 2008.

[152] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[153] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st {USENIX} Security Symposium*, 2012.

[154] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[155] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, 2014.

[156] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[157] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[158] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[159] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[160] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRMAFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *Proceedings of the 28th USENIX Security Symposium*, 2019.