2026

Improving the security of Linux-based IoT firmware code via re-hosting and dynamic analysis techniques

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

IMPROVING THE SECURITY OF LINUX-BASED IOT FIRMWARE CODE VIA RE-HOSTING AND DYNAMIC ANALYSIS TECHNIQUES

by

IOANNIS ANGELAKOPOULOS

B.S., National Technical University of Athens, 2018

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2026

Approved by

First Reader

Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Gianluca Stringhini, PhD
Associate Professor of Electrical and Computer Engineering

Third Reader

David Starobinski, PhD
Professor of Electrical and Computer Engineering
Professor of Systems Engineering
Professor of Computer Science

Fourth Reader

Alan Zaoxing Liu, PhD
Assistant Professor of Computer Science
University of Maryland

Acknowledgments

First, I would like to express my sincere gratitude and appreciation to my advisor, Prof. Manuel Egele and co-advisor Gianluca Stringhini, for their invaluable guidance and support throughout my PhD journey. I would also like to thank Prof. David Starobinski and Prof. Alan Zaoxing Liu for their guidance early on during my PhD.

In particular, I would like to thank my friends, who I also consider family, for the immeasurable support and understanding I received from them during the hardships I faced in PhD: Alexandra Zarifis, Aris Skandamis, Giorgos Karantonis, Ioannis Gasteratos, and Evangelos Bitsikas. I am also grateful for my friends in the SeclaBU research group for all the insightful discussions and help they provided to me throughout the years.

Finally, I want to thank my parents and my siblings for their unconditional love and support. All the success I have achieved in my life, I owe it to them.

IMPROVING THE SECURITY OF LINUX-BASED IOT FIRMWARE CODE VIA RE-HOSTING AND DYNAMIC ANALYSIS TECHNIQUES

IOANNIS ANGELAKOPOULOS

Boston University, College of Engineering, 2026

Major Professors: Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

Gianluca Stringhini, PhD
Associate Professor of Electrical and Computer Engineering

ABSTRACT

The Internet of Things (IoT) comprises of billions of Internet-connected devices and gadgets that run dedicated software, called firmware. Even though these devices are at the forefront of innovation, their firmware is notoriously famous for its numerous security weaknesses. For this reason, IoT firmware has become a prime target for malicious actors. Despite the contributions of prior research works in identifying and remediating the security bugs and vulnerabilities in IoT firmware, through static, dynamic and hybrid analysis techniques, new security issues arise on a daily basis (Claroty, 2022).

When it comes to Linux-based IoT firmware vulnerability analysis, dynamic vulnerability analysis techniques that rely on firmware re-hosting (emulation) are the most popular. In this thesis, I focus on addressing challenges in both the firmware re-hosting and dynamic vulnerability analysis domains and propose novel contributions to improve the

security posture of IoT firmware. Specifically, I introduce new techniques for firmware re-hosting which enable the application of downstream (dynamic) analysis systems on binary Linux-based kernel-level firmware code. I also present methods to enhance the scope of firmware dynamic analysis techniques to detect diverse bugs and vulnerabilities (e.g., memory corruption and deadlock bugs) in binary privileged (kernel-level) Linux-based firmware code. When it comes to improving firmware re-hosting, I propose three approaches; `FirmSolo`, `Pandawan`, and `FirmDiff`. I first present `FirmSolo`, a framework designed to re-host the Linux-based binary IoT kernel modules within firmware images, at scale, and expose these modules to existing downstream analysis. The evaluation of `FirmSolo` not only shows that `FirmSolo` can successfully re-host the majority of the binary kernel modules in firmware images, but also enable dynamic analysis systems to detect multiple previously-unknown bugs in these modules. Next, I detail `Pandawan`, a framework whose contribution is twofold; 1) The ability to objectively compare full-system re-hosting approaches based on their emulation capabilities, and 2) The ability to holistically (both the user and privileged level) re-host and analyze IoT firmware. `Pandawan`'s evaluation demonstrates that `Pandawan` is able to facilitate the objective comparison of the state-of-the-art re-hosting systems based on emulation-specific metrics (e.g., code coverage) defined by the user. Furthermore, due to its holistic re-hosting and analysis capabilities, `Pandawan` is able to discover both previously known and unknown bugs in binary IoT kernel modules. Next, I discuss `FirmDiff`, an automated binary diffing framework that enables analysts to improve the fidelity of the IoT kernel module re-hosting process and achieve a more effective analysis. These contributions to firmware re-hosting (specifically of privileged firmware code) instigate the development of novel firmware analysis techniques. Thus finally, I propose `Lock 'n Load` (`LL`) as my contribution to the firmware analysis landscape. `LL` is a dynamic analysis framework that uses my re-hosting frameworks to enable the analysis of proprietary binary-only IoT kernel modules for deadlock-related bugs. My analysis with

LL reveals multiple previously unknown deadlock bugs in multiple binary-only targets.

In summary, I improve the security of Linux-based IoT firmware code by introducing (holistic) re-hosting techniques which expose Linux-based privileged IoT firmware code to downstream analysis, at scale. Furthermore, I apply novel dynamic analysis techniques to privileged Linux-based IoT firmware code to detect diverse bugs (e.g., deadlock bugs). These contributions are only possible due to the firmware re-hosting and analysis frameworks developed during my research.

Contents

List of Tables

List of Figures

List of Abbreviations

DDoS	.....	Distributed Denial of Service
DMA	.....	Direct Memory Access
GVT	.....	Ghidra Version Tracking
IoT	.....	Internet of Things
LKM	.....	Loadable Kernel Modules
MMIO	.....	Memory-Mapped Input/Output
OS	.....	Operating Systems
OSI	.....	Operating Systems Introspection
VM	.....	Virtual Machine
VMM	.....	Virtual Machine Monitor

Chapter 1

Introduction

The Internet of Things (IoT) has been growing rapidly over the past few decades without any signs of slowing down. By 2030, the number of these smart gadgets and devices connected to the Internet will reach the 40 billion mark (IoT Analytics GmbH, 2024). Despite their massive popularity and widespread adoption, IoT devices consistently fail to adhere to the appropriate and best security practices and design specifications (Microsoft, 2023), thus these devices are attractive targets for malicious actors. Attackers have been compromising these devices, by exploiting their firmware (software) security weaknesses, to conduct high profile attacks such as the Deadbolt (ransomware) attack (Ellzey et al., 2022), Stuxnet (Malwarebytes, 2025), the Verkada (smart cameras) hack (Verkada, 2021), and the infamous Mirai botnet DDoS attack (Antonakakis et al., 2017) which became the catalyst for the emergence of various IoT botnets over the years. IoT vendors tend to overlook the reasons behind these attacks such as the use of trivial passwords, outdated software, and the lack of authentication. To make matters worse, vendors often develop and use their own custom software and/or rely on third party solutions to support the myriads of device-exclusive peripherals used by the IoT devices. Unfortunately, the newly introduced proprietary code frequently does not undergo thorough security testing during its development cycle. In addition, this proprietary code is commonly distributed in closed-source binary form, hindering the analysis attempts from researchers and independent analysts.

To combat the negligence of vendors to secure their products against IoT focused attacks, researchers have devised techniques to analyze the firmware code that runs on dif-

ferent types of IoT devices for bugs and vulnerabilities while urging the vendors to take prompt action. Muench et al. (Muench et al., 2018) and Fasano et al. (Fasano et al., 2021), provide a clear distinction on the different types of IoT devices based on the type of firmware (operating system) they use. Specifically, IoT devices can be organized into three classes; *Type-I*, *Type-II*, and *Type-III*.

Type-I: General Purpose OS devices. These types of devices run firmware that is based on general purpose operating systems (e.g., Linux) modified to fit embedded devices. Similar to general purpose operating systems, general purpose firmware features both a user and kernel space component. However, the user space environments (e.g., busybox) used by Type-I devices are generally more lightweight than the ones running on full-fledged systems (e.g., desktops and servers).

Type-II: Embedded OS devices. These are low-power devices, such as IP cameras and MP3 players that run single-purpose custom operating systems called embedded firmware (e.g., uClinux). While a Memory Management Unit (MMUs) is frequently absent from these devices, their embedded firmware still adopts a basic distinction between the user and kernel space components.

Type-III: Devices without an OS abstraction. These devices run custom single-purpose operating systems called "monolithic firmware", which combine both the user and kernel space components into a single program. Internally, this type of firmware implements an infinite loop which handles interrupts triggered by the device's hardware peripherals in response to events occurring in the physical world.

The firmware analysis approaches that analyze these three types of IoT firmware can be further classified into three categories depending on their analysis type; static analysis, hybrid analysis, and dynamic analysis. Static analysis methods analyze the firmware without executing it. While these techniques are highly scalable due to their independence on the physical hardware, they suffer from false positives during the analysis. Hybrid approaches

combine the physical IoT device and an emulator to conduct their analysis. Typically, the emulator dynamically executes and analyzes the firmware code while I/O operations (e.g., MMIO and DMA) are forwarded to the physical device. While these techniques are very effective and accurate, the requirement for the IoT device to be available renders these approaches non-scalable and cost-inefficient. Finally, dynamic methods analyze the IoT firmware code either while it executes directly within a physical device or within a re-hosting environment (i.e., an emulator). Specifically, dynamic analysis techniques that rely on re-hosting (emulation) provide a reasonable tradeoff between the scalability, effectiveness and accuracy of the analysis, hence these techniques are generally more preferable compared to static and hybrid methods.

This thesis focuses on dynamic analysis techniques that rely on (full-system) firmware re-hosting to analyze Linux-based firmware code, which runs on Type-I IoT devices, for bugs and vulnerabilities.

To this end, I propose novel contributions to improve the security of Linux-based IoT firmware through full-system firmware re-hosting and dynamic analysis techniques.

1.1 Thesis Contributions

This thesis proposes innovative solutions that benefit from both source code and binary analysis, to address the limitations of prior solutions and ultimately improve the security posture of Linux-based IoT firmware. In particular, I present two novel fully automated re-hosting techniques, and a method to improve firmware re-hosting fidelity. Furthermore, I introduce the first fully automated deadlock detection approach for binary-only kernel-level firmware code. My thesis is summarized in the following statement:

Combining holistic firmware re-hosting with a comprehensive analysis of IoT firmware enables the detection of diverse types of bugs and vulnerabilities in open and closed-source privileged binary firmware code, thus improving the security of IoT firmware.

Here, I note that I consider firmware analysis techniques that focus exclusively on either the kernel or user space aspects of IoT firmware code as non-comprehensive. Conversely, I consider firmware analysis methods whose scope encapsulates both firmware code aspects as comprehensive. To address the above thesis statement, my Ph.D. research consists of four novel contributions; three approaches focus on firmware re-hosting, and one introduces a new dynamic firmware analysis method. I first detail my contributions in improving Linux-based IoT firmware re-hosting. Next, I discuss my contribution in expanding the scope of dynamic firmware analysis techniques to detect availability bugs.

1.1.1 Firmware Re-hosting

Firmware re-hosting (emulation) techniques have gained a lot of popularity in recent years due to their scalability (there is no need for the physical hardware to be available) and their compatibility with widely-used dynamic analysis techniques (e.g., fuzz testing). Before my contributions, works in this area were categorized in two groups; 1) Approaches that re-host only user-level firmware code, and 2) Approaches that re-host open-source kernel-level firmware code. Given the high complexity and dependency of kernel-level firmware code to the physical hardware (e.g., hardware peripherals), the majority of prior works focus on user firmware application re-hosting and analysis (Chen et al., 2016; Kim et al., 2020; Vetterl and Clayton, 2019; Zheng et al., 2022). Only a handful of existing approaches (Pustogarov et al., 2020; Liu et al., 2022) focus on re-hosting and analyzing kernel-level firmware code (e.g., kernel modules and drivers), albeit the firmware code has to be open-source for these systems' analysis to be fruitful. Since the binary IoT kernels used by the firmware images are closely dependent on the hardware that they are built for, current emulators fail to emulate these kernels due to the lack of support for the myriads of hardware peripherals used by the IoT devices. To make matters worse, IoT firmware often incorporates insufficiently tested proprietary kernel code in the form of kernel modules and drivers. Due to the absence of the source code of these kernel modules, it is very challeng-

ing for analysts to test these modules for bugs and vulnerabilities. In general, to be able to re-host binary(-only) firmware kernel modules and drivers, re-hosting techniques have to address the following challenges:

1. Must use their own custom kernels, that are built using the source files available in the upstream (open-source) repositories.

2. Ensure that the custom kernels provide all the symbols (i.e., functions and data structures) required by the binary firmware kernel modules to load in these kernels.

3. Ensure that the memory layout of data structures that are shared between the custom kernel and the binary firmware kernel modules is consistent. This requirement is mandatory to ensure that the firmware kernel modules do not incorrectly access the members of these data structures while being loaded or executing in the custom kernel. Misaligned data structure accesses would inevitably lead to kernel module crashes during the emulation. Since the binary kernel modules are immutable all the changes regarding the layout of the data structures must occur in the context of the custom kernel.

My works address these challenges to enable the re-hosting of both open and closed-source privileged firmware code (in the form of kernel modules and drivers) and expose that code to various dynamic analysis approaches. Initially, I discuss my contributions to binary IoT kernel module re-hosting and holistic (both the user and kernel level) firmware re-hosting and analysis. Next, I present a novel method to improve the emulation fidelity of re-hosting frameworks that target the privileged (kernel-level) aspect of IoT firmware.

FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules

The Linux-based firmware running on Internet of Things (IoT) devices is complex and consists of user level programs as well as kernel level code. Both components have been shown

to contain serious security vulnerabilities. As mentioned previously, the existing firmware re-hosting and analysis systems are unable to analyze neither the IoT kernel proper nor the binary kernel modules used by IoT gadgets. This limitation is troublesome, because a vulnerability in a kernel module or the kernel proper can risk compromise of the entire system. While upstream Linux kernel modules receive fixes from their corresponding maintainers, vendors often use outdated kernel versions and do not backport these fixes in their products. In addition, the presence of untested binary-only proprietary modules in IoT firmware makes matters even worse. A notable example of a vulnerable binary-only kernel module is Kcodes' `NetUSB.ko` which exposes USB printers connected to a WiFi router to the network and features a Remote Code Execution vulnerability (CVE-2015-3036). Despite the serious consequences that such a vulnerability can have on affected devices, the research community lacks techniques capable of automatically detecting bugs that are located in the privileged code of Linux-based IoT systems.

As my first contribution, I propose `FirmSolo`, the first scalable firmware re-hosting system to automatically expose IoT privileged kernel-level code (in the form of Linux kernel modules) to downstream analysis. A prerequisite to any dynamic analysis (i.e., analysis of the code while it is executing) of a kernel module is that the module is successfully loaded by the kernel, a process guarded by the kernel's module loading facility. All but the most trivial modules will require interaction with the kernel proper through kernel symbols; functions or kernel-defined data structures (see the re-hosting challenges in Section 1.1.1). As type information is removed during compilation, it is imperative that the data structures defined in the kernel have the same memory layout expected by the kernel modules. Given that the IoT firmware kernel modules are immutable (distributed in binary form), the layout of the data structures can only be modified on the kernel side. The availability of symbols and the layout of data structures in the kernel are controlled by options that constitute the configuration (i.e., `.config`) of the kernel. Thus, `FirmSolo`'s goal is to reverse engineer the

IoT kernel's configuration such that an open-source kernel that is built with this configuration can successfully load, and subsequently analyze, the kernel modules distributed in real-world firmware images. To this end, `FirmSolo` implements a fully-automated two-pronged hybrid analysis process called **K***ernel* **C***onfiguration* **R***everse* **E***ngineering*. K.C.R.E. first leverages static symbol and dependency information from an image's kernel modules to infer an approximate kernel configuration. Subsequently, K.C.R.E. iteratively augments this configuration with a dynamic analysis used to align the data structures shared between the modules and the kernel proper, yielding a kernel that can load the binary kernel modules within firmware images of real-world IoT devices.

To demonstrate `FirmSolo`'s main utility, I integrated `FirmSolo` with two representative analysis systems; the TriforceAFL (NCC Group Plc, 2017) kernel fuzzer and the Firmadyne (Chen et al., 2016) dynamic firmware analysis system. While initially unable to analyze Linux IoT firmware kernel modules, combined with `FirmSolo` both systems can analyze kernel modules in IoT firmware. The evaluation of `FirmSolo` on 1,470 firmware images containing 56,688 kernel modules, shows that `FirmSolo` can successfully load 64% of these kernel modules. In addition, the fuzzing experiments with TriforceAFL on 75 kernel modules discovers 19 previously unknown bugs in 11 distinct proprietary kernel modules. Through Firmadyne, I confirm the presence of these previously-unknown bugs in 84 firmware images. Finally, by using `FirmSolo`, Firmadyne confirms a previously-known memory corruption vulnerability in five different versions of the closed-source Kcode's *NetUSB* module across 15 firmware images. I provide in-depth details about the design and implementation of `FirmSolo` in Chapter 4.

Pandawan: Quantifying Progress in Linux-based Firmware Rehosting

Full-system re-hosting (or emulation) techniques are the most popular when it comes to Linux-based firmware vulnerability analysis. Even though these techniques revolutionized IoT firmware analysis, by extending bug and vulnerability testing towards increasingly

sophisticated firmware, the field is characterized by the absence of two important aspects. First, the community lacks a mechanism to objectively compare the capabilities of different re-hosting systems. Specifically, we need answer the following question:

> ***Q:*** *How can we identify whether a re-hosting framework performs better than the others?*

The second missing capability of full-system Linux-based re-hosting is that current state-of-the-art systems target only one aspect of IoT firmware; either its user level (Chen et al., 2016; Kim et al., 2020; Vetterl and Clayton, 2019; Tay et al., 2023; Zheng et al., 2022) or its privileged (kernel) level aspect (Angelakopoulos et al., 2023; Zheng et al., 2022; Pustogarov et al., 2020). In real world devices, though, both the user and privileged aspect of the IoT firmware run cooperatively. To effectively analyze IoT firmware code for bugs and vulnerabilities it is therefore important to follow a more *holistic* approach that includes both aspects of the IoT firmware in the re-hosting and analysis process.

To address these shortcomings, I present `Pandawan` a firmware re-hosting framework whose contribution is twofold; `Pandawan` implements the `Firmware Initialization Completion Detection` (`FICD`), a novel technique that enables the objective comparison of different re-hosting frameworks based on their emulation capabilities. Specifically, `FICD` detects the reference point at which the firmware's initialization phase is complete during emulation. This point is conceptually common for all full-system re-hosting approaches. Crucially, `FICD` makes it possible to quantify the forward progress in firmware re-hosting, by enabling the meaningful comparison of different full-system re-hosting approaches on metrics such as the number of user programs executed, user and kernel code coverage, or number of kernel modules loaded, up to the reference point. By using the reference point the comparison is independent of the firmware re-hosting frameworks' performance.

To fill the void in holistic full-system firmware re-hosting `Pandawan`, introduces a novel

re-hosting approach that builds atop of (Py)PANDA (Craig et al., 2021; Dolan-Gavitt et al., 2015) and FirmSolo's firmware re-hosting capabilities to provide Operating System (OS) level introspection and dynamic analysis of both user and privileged (i.e., kernel module) firmware code. To achieve holistic firmware re-hosting and analysis, `Pandawan` features the `Kernel Augmentation` (`KA`) technique. `KA` first isolates the kernel functionality typically included by vendors in their IoT firmware kernels (in the form of kernel symbols). Then, it builds `Pandawan`'s "augmented" kernels which contain this functionality to enable the successful re-hosting of firmware user level code and kernel modules alike.

I use `FICD` to compare the `Pandawan`, Firmadyne, FirmAE, and FirmSolo re-hosting frameworks. For my experiments, I use a dataset consisting of 1,520 firmware images. My investigation shows that `Pandawan` outperforms Firmadyne, FirmAE, and FirmSolo, by executing up to 6% more user level programs and 21% more user code basic blocks, on average, than these systems. In addition, `Pandawan` loads 9% more IoT kernel modules and executes 26% more kernel module basic blocks on average than `FirmSolo`.

Finally, I evaluate `Pandawan`'s holistic analysis on the same dataset of 1,520 firmware images. Fuzzing the entire system call interface would not be efficient, since the majority of system calls (depending on the arguments they are given) would target the core kernel code instead of kernel module code. Thus, `Pandawan` targets only the subset of system calls that result in the execution of code in IoT kernel modules. To this end, `Pandawan`'s holistic analysis approach fundamentally enables the tracing of system calls invoked by user level programs which lead to kernel module code execution. In turn, the traces expose entry points from user level to kernel (module) level code. `Pandawan` uses the entry point information to generate system call chains in a format (seeds) compatible with the TriforceAFL (NCC Group Plc, 2017) kernel fuzzer to fuzz the kernel modules within the firmware images. `Pandawan` collects traces for 479 firmware images. After transforming these traces into seeds and providing them to TriforceAFL, the fuzzer triggers 16 bugs on 8

closed and 4 open source kernel modules. Twelve of the bugs found are previously known and six are previously unknown bugs. I provide detailed information about `Pandawan` and its inner working in Chapter 5.

FirmDiff: Improving the Configuration of Linux Kernels Geared Towards Firmware Re-hosting

Ideally, the advances in firmware re-hosting have lead to a situation where the analyst can trigger the emulation and analysis of arbitrary firmware images, only with the press of a button. However, reality shows that due to the diversity of IoT firmware images, fully automated solutions only work occasionally. Thus, analysts often have to manually intervene with the analysis process and invest an extensive amount of engineering effort and time to address the issues and ensure further progress. Unfortunately, there are few automated solutions that can lift some of the manual investigation burden and improve the efficiency and effectiveness of firmware analysis.

FirmSolo (and other frameworks built upon FirmSolo such as Pandawan) is the only scalable system that automatically configures and builds custom emulation-ready kernels, whose configuration approximates the corresponding IoT kernels. FirmSolo uses these kernels to load and dynamically analyze the IoT kernel modules within firmware images. However, its kernel configuration process is inherently dependent on the metadata (i.e., kernel symbols) embedded within the IoT privileged code (kernel and kernel modules) and the kernel source code used to compile the custom kernels (FirmSolo uses only open source kernel versions). To address the re-hosting challenges discussed in Section 1.1.1, FirmSolo primarily ensures that kernel symbols (functions and data structures) are available to the IoT kernel modules in order for the latter to load into the custom kernel. FirmSolo focuses on lower level implementation semantics, such as ensuring that the IoT kernel modules access the members of the data structures in the custom kernels at the correct offset, only when issues arise during re-hosting. In particular, inconsistencies in the memory layout

of these data structures can lead to misaligned data structure memory accesses during the analysis, which in turn can lead to these IoT kernel modules crashing.

Generally, the memory layout of kernel data structures is affected by two causes: the configuration options used in the kernel build process and any modifications directly applied by the vendors in their kernels' source code. While FirmSolo attempts to fix the layout of data structures when IoT kernel modules crash while being loaded into the custom kernel, the layout alignment algorithm used by FirmSolo is limited. FirmSolo's layout alignment algorithm must meet certain requirements, such as that the crashing IoT kernel modules must be open source and not stripped and that the IoT kernels' source code must not be modified by vendors. If these prerequisites are not met, the layout recovery algorithm will create incorrect solutions. Specifically, in the cases where data structures are modified by vendors at source (e.g., extra members added), FirmSolo cannot detect and apply the correct modifications in these data structures. Since FirmSolo uses the open-source versions of the IoT kernels and is limited only to adjusting the layout of data structures through toggling configuration options during the kernel build process, modified data structures remain misaligned. To successfully load and analyze the kernel modules in these cases, the analyst has to manually detect the modifications and configuration options that align the offending data structures within FirmSolo's kernels.

To address this issue, as my third contribution to firmware re-hosting, I present `FirmDiff`, a binary diffing framework that compares the memory layout of data structures common to both the IoT kernel modules (and IoT kernel) and FirmSolo's custom kernels. `FirmDiff`'s goal is to help the analyst identify differences in the memory layout of data structures common between the IoT kernel modules and FirmSolo's custom kernels, and also provide sufficient information to the analyst to address these differences. Consequently, by aligning the layout of these data structures, the analyst can build custom kernels that closely resemble their IoT counterparts. The premise behind building these re-

fined kernels is to load and analyze the IoT kernel modules without misaligned data structure accesses that occur during emulation, thus improving the overall firmware re-hosting. Specifically, the outcome of `FirmDiff`'s analysis are the names of data structures and their members (i.e., their offsets and names) that are misaligned between the IoT kernel modules and FirmSolo's custom kernel. Analysts can use this information to enhance FirmSolo's kernel configuration and build process to produce refined custom kernels that can load and analyze the IoT kernel modules without crashes thwarting the analysis.

I evaluate `FirmDiff` on a dataset of 7 firmware images, originating from FirmSolo. Furthermore, I also analyze 3 additional modern images from the Greenhouse dataset (published in 2023) to assess whether `FirmDiff`'s analysis capabilities work for modern firmware. Overall, these firmware images contain 148 kernel modules that crash during the analysis with FirmSolo. I showcase how the information produced by `FirmDiff` aids analysts to supplement the configuration process of FirmSolo to produce custom kernels that better approximate the IoT kernels used by the firmware images in our dataset. `FirmDiff` identifies 37 misaligned data structures in FirmSolo's kernels for these images. After aligning the layout of 35 of these data structures, FirmSolo's refined kernels successfully load 28 previously crashing kernel modules. I discuss the design and implementation details of `FirmDiff` in Chapter 6.

1.1.2 Firmware Analysis

Prior to my contributions in firmware re-hosting, only few works (Pustogarov et al., 2020; Liu et al., 2022) were capable of analyzing privileged Linux-based firmware code for bugs and vulnerabilities. As mentioned in Section 1.1.1, these works are severely limited both in terms of scalability and their analysis capabilities. My re-hosting frameworks are the first to expose (open and closed source) Linux-based kernel-level firmware code (i.e., kernel modules) to dynamic bug and vulnerability analysis, at scale. Crucially, my firmware re-hosting contributions also foster the development of new dynamic firmware analysis

techniques (such as deadlock detection analysis) that were previously not possible. On that note, I discuss my contribution on the expansion of the analysis scope of current firmware analysis techniques.

Lock 'n Load: Detecting Deadlocks in Linux-based Binary-only Firmware

Current firmware analysis techniques suffer from one important limitation; their analysis scope. The majority of works focus mainly on the detection of memory corruption bugs (i.e., violations of *Confidentiality* and *Integrity*), and largely eschew *Availability* bugs, such as those induced by deadlocks (e.g., threads blocking due to contention for shared resources).

Nowadays, deadlocks are prevalent in systems, such as IoT (Elwel, 2018), where multiple CPUs, threads, and task parallelism have become the norm. These types of bugs can be devastating when they occur within the context of the kernel, as they usually end up crashing or freezing the entire system. Currently there are no kernel-level mechanisms to recover from deadlocks, as the only solution is to reboot the operating system. To this end, IoT devices often implement a (software or hardware) watchdog timer that restarts the system if it becomes unresponsive for a predefined amount of time. Fortunately, IoT firmware borrows the majority of its kernel (module) code from the upstream version of the Linux kernel, which is thoroughly tested for deadlocks by fuzzers (e.g., syzkaller (Google, 2023)) and test suites (e.g., xfstests (XFS team, 2025)), using Linux's bespoke Lockdep deadlock detection subsystem (see Section 2.6.1). As mentioned in Section 1.1.1, closed-source proprietary kernel modules are prevalent in IoT firmware. Thus, kernel-level deadlocks in binary-only IoT kernel modules and drivers can become an attractive alternative for attackers to exploit and reach a form of Denial of Service, preventing the normal operation of IoT devices.

While deadlock detection systems are not new, prior works on deadlock analysis do not support closed-source kernel level firmware code. Both static and dynamic analysis ap-

proaches depend on the availability of their target's source code and cannot analyze binary-only kernel module code by design. The state-of-the-art kernel dynamic deadlock detection mechanism is Lockdep (Molnar and Van de Ven, 2025) (i.e., the Runtime Locking Correctness Validator) which has been part of the Linux kernel for almost two decades.

Lockdep operates on specific meta-information (so-called *lockdep maps*) that are inlined within the locking (and other miscellaneous) data structures during compilation, provided that Lockdep is enabled during the kernel configuration process. Regrettably, firmware re-hosting approaches cannot simply enable Lockdep in their custom kernels to analyze the binary IoT kernel modules for deadlock-based bugs. First, due to its high performance impact, Lockdep is primarily used during the kernel development phase and not in production systems such as the Commercial Off-the-Shelf (COTS) IoT devices. Thus, the IoT kernels and by extension their modules are not compiled with Lockdep enabled, and in turn they do not contain Lockdep's metadata. Second, it is necessary to ensure the memory layout consistency of the kernel data structures between the custom re-hosting kernels and IoT kernel modules (see the challenges in Section 1.1.1), to avoid kernel module crashes during emulation. Consequently, both the IoT kernel and modules must be compiled with Lockdep to enable the analysis. Unfortunately, when it comes to binary-only kernel modules, their source code is not publicly available and thus a recompilation is not possible in this case. These mutually exclusive requirements necessitate that the custom re-hosting kernels must be compiled without Lockdep enabled to be able to re-host the binary(-only) IoT kernel modules.

To address the aforementioned challenges and enable the detection of deadlocks in IoT firmware, I propose *Lock 'n Load* (`LL`), the first deadlock analysis framework targeting Linux kernel-level (binary-only) firmware code. `LL` introduces `Shadow Lockdep`, a subsystem (complementary to Lockdep) which decouples Lockdep's metadata (usually inlined into the kernel data structures) into a (shadow) memory, called `Shadow Lockdep`

`Maps` (*SLoMs*), separate from the kernel data structures themselves. By decoupling Lockdep's metadata from the kernel data structures, `LL` ensures that the memory layout of the data structures in the custom re-hosting kernel(s) aligns with the memory layout of the immutable data structures in the binary-only IoT kernel modules. Consequently, `LL` can apply Lockdep's deadlock detection logic (unmodified) to these kernel modules, without the requirement for Lockdep's metadata to be embedded in the modules' data structures. I evaluate `LL` on 80 binary-only IoT kernel modules that make use of locks. `LL`'s deadlock detection analysis discovers 6 potential deadlocks in 6 binary-only IoT kernel modules.

1.2 Contributions Summary

Prior works on firmware re-hosting and analysis target primarily user-level IoT firmware code while kernel-level (proprietary) firmware code is usually left out of the analysis, due to its overall complexity. Furthermore, the difficulty in overcoming the challenges of firmware re-hosting (see Section 1.1.1), greatly limits the scope of existing dynamic firmware analysis approaches. Most of these works focus on identifying *Integrity* and *Confidentiality*, such as memory corruption bugs, while *Availability* bugs (e.g., deadlock-related bugs) are left unexplored. This thesis proposes that to improve the security of IoT firmware, it is crucial to combine holistic firmware re-hosting with the comprehensive analysis of IoT firmware code to detect diverse types of bugs and vulnerabilities in open and closed-source privileged firmware code. To support my thesis statement, I make the following contributions:

- I propose a fully-automated firmware re-hosting framework that exposes Linux-based privileged firmware code (in the form of kernel modules) to downstream analysis (Angelakopoulos et al., 2023).

- I introduce a method for the objective comparison of full-system re-hosting approaches based on their emulation capabilities, and a holistic firmware re-hosting

and analysis framework that targets both the user and kernel-level aspect of Linux-based IoT firmware (Angelakopoulos et al., 2024b).

- I present a binary diffing framework which aids analysts to improve the fidelity of Linux-based IoT re-hosting frameworks and thus increase the soundness of the analysis (Angelakopoulos et al., 2024a).
- I propose a dynamic firmware analysis framework that focuses on detecting deadlock-based bugs on binary-only IoT kernel modules.

Chapter 2

Background

In this chapter, I provide background information on the Linux kernel, the Loadable Kernel Modules (LKMs), and kernel module loading process. In addition, I elaborate on (full-system) firmware re-hosting, Operating Systems (OS) introspection frameworks (e.g., Panda/PyPANDA), re-hosting frameworks (e.g., Firmadyne and FirmAE), binary analysis tools (e.g., Ghidra and Ghidriff), and fuzzers (e.g., TriforceAFL, syzkaller) that my frameworks build upon. Finally, I discuss the definition of deadlocks and the functionality of Linux's Lockdep deadlock detection subsystem. These concepts and systems constitute the building blocks for my firmware re-hosting and analysis frameworks, which I detail in the next chapters.

2.1 Linux Kernel

```
1 config BRIDGE_NETFILTER
2   bool "Bridged IP/ARP packets filtering"
3   depends on BRIDGE && NETFILTER && INET
4   depends on NETFILTER_ADVANCED
5   default y
```

Listing 2.1: Linux-2.6.30 BRIDGE_NETFILTER option definition

The Linux kernel provides an elaborate configuration system reflected in thousands of configuration options that, if selected by the user, enable certain functionality (e.g., networking support). These options and their values constitute the configuration file `.config`, which is a prerequisite to compile a Linux kernel through its build system. The definitions

of the options are present in the hundreds of (*Kconfig*) files in the kernel source tree, and each option has one of five types (`bool`, `tristate`, `string`, `hex`, or `int`). The majority of options are of type `bool` or `tristate` and can be assigned a value from the set {n,y} or {n,m,y}, respectively. These values translate to *not selected (n)*, *compile as part of the main kernel (y)*, and *compile as a Loadable Kernel Module (LKM) (m)*. Furthermore, the availability of an option can be guarded by dependencies in the form of logic expressions over other options and their values. Intuitively, an option can only be assigned a value if its dependencies (also expressed in the *Kconfig* files) are fulfilled. For example, Listing 2.1 shows the definition of the `BRIDGE_NETFILTER` option which enables the kernel's netfilter subsystem to see bridged IP and ARP traffic and is commonly enabled on devices that provide firewall capabilities, such as WiFi home routers. This option in turn depends on `NETFILTER_ADVANCED` and a logical expression of three other options.

Users can interact with the kernel's configuration system through various make targets and other tools. The result of all these mechanisms is the `.config` configuration file which contains the configuration options and their corresponding values in the format `CONFIG_<option_name>=<value>`. Next, I elaborate on Linux's kernel modules and the process of loading kernel modules into the kernel at runtime.

2.1.1 Loadable Kernel Modules

To avoid turning the Linux kernel into a large monolithic binary, a user can opt to compile a portion of the kernel source code as Loadable Kernel Modules (LKM). A LKM is an `ELF` binary which, once loaded, extends the functionality of the main kernel at runtime. Typical uses of LKMs include support for new hardware peripherals (i.e., device drivers), file-systems, and supplements to the kernel network stack.

To ensure interoperability between the kernel and a LKM, it is essential that each LKM is compiled using the same configuration that was used for the kernel proper. Using the same configuration ensures that symbols (functions and data structures), which are common

to the kernel and the LKMs, have the same type and layout, and that the LKM will not cause an error after it is loaded into the kernel. In addition, LKMs must also match the *vermagic* (version magic) of the kernel that captures information on the kernel's version, *Symmetric Multiprocessing*, *Preemption*, and *Module Versioning*.

Comparable to shared libraries, LKMs can access externally defined functions and variables (i.e., symbols) beyond their own code, as long as these symbols are exported either by the kernel proper or another module (i.e., via the `EXPORT_SYMBOL[_GPL]` macros). Furthermore, if the user enables the `CONFIG_KALLSYMS` option, the compiler will preserve information about all exported symbols and their addresses in the kernel binary.

2.1.2 Module Loading Process

The module loading process is initiated by the `init_module` system call, commonly invoked by `insmod` or `modprobe`. Loading a module consists of six stages: 1) check if the user has permission to load modules, 2) copy the module binary code from user-space to a temporary kernel-space buffer, 3) check if the module and kernel *vermagic* match, 4) allocate the actual kernel memory for the module, 5) perform symbol resolution and relocation, and finally 6) call the module's initialization function (`module_init`).

2.2 Full-system Firmware Re-hosting

Full-system firmware re-hosting (or emulation) (Fasano et al., 2021) is a technique used to run a binary firmware image without the need for the IoT hardware it was designed for. Particularly, the firmware (image) code is executed within an emulated environment (e.g., QEMU (Bellard, 2005)), where hardware accesses are handled by the emulator. Full-system firmware re-hosting has become popular mainly due to two major benefits; 1) allowing the emulation of IoT firmware at scale, and 2) exposing firmware code to dynamic analysis (e.g., fuzzing) and operating system level introspection. However, full-system

firmware re-hosting can be challenging. A recurring issue in the IoT infrastructure is the *lack of standarization* (D. Ganesan, 2017). Particularly, the absence of common specifications and protocols leads to the production of billions of devices with diverse and proprietary hardware (e.g., NVRAM and Wireless NICs). Hence, IoT firmware and specifically its kernel component is dependent on each particular IoT device's System on Chip (SoC) and hardware peripherals.

Unfortunately, state-of-the-art emulators such as QEMU only support a fraction of the hardware peripherals used by IoT and embedded devices (Fasano et al., 2021) and thus are unable to emulate the IoT firmware kernels. As a result, full-system re-hosting approaches either target only user level IoT firmware code (Chen et al., 2016; Kim et al., 2020; Vetterl and Clayton, 2019) or the IoT kernel modules (Pustogarov et al., 2020), by substituting the original firmware kernel with a custom kernel compatible with the emulators. Next, I discuss two state-of-the-art re-hosting systems that my re-hosting frameworks use and I compare my re-hosting frameworks against.

2.2.1 Firmadyne & FirmAE

Firmadyne (Chen et al., 2016) and FirmAE (Kim et al., 2020) are among the state-of-the-art when it comes to full-system firmware re-hosting and dynamic analysis approaches. Specifically, they are designed to emulate the user level code of IoT firmware images and subject it to various bug and vulnerability analyses, such as testing against exploits from the Metasploit framework (Rapid7, 2023) or fuzzing. By design, both systems replace the original firmware kernels, which are incompatible with QEMU, with custom pre-built kernels that can be booted under QEMU. Since both the original IoT kernels and the pre-built kernels are based on Linux, they share the same stable system call interface required by user level firmware code to run.

By relying on custom pre-built kernels, Firmadyne and FirmAE only support user level firmware code. The Linux kernel modules contained within the IoT firmware file-systems

cannot be loaded by the pre-built kernels. These kernels do not meet the requirements of Linux's module loading process, such as exporting kernel symbols (i.e., functions and data structures) required by the kernel modules to load into the kernel. As a result, the analysis of privileged IoT firmware code is out of scope for both Firmadyne and FirmAE.

2.3 Operating Systems Introspection

Operating Systems Introspection (OSI) is a technique used to inspect and modify the state (e.g., memory and registers) of a virtual machine (VM). Specifically, the OSI operations are conducted through the Virtual Machine Monitor (VMM) or hypervisor (e.g., QEMU), which has direct access to the guest's memory and registers. Next, I provide information about the PyPANDA OSI framework that `Pandawan` builds upon.

2.3.1 (Py)PANDA

PyPANDA (Craig et al., 2021) is a Python frontend to PANDA (Dolan-Gavitt et al., 2015), an Operating System (OS) level introspection framework built on top of QEMU. Besides emulating IoT related architectures such as ARM and MIPS, PANDA and PyPANDA also enable users to control the guest code execution and inspect the guest's internal state (e.g., memory). The OS level guest introspection is possible through a set of PANDA's existing C/C++ plugins and through PyPANDA's Python plugin interface. These plugins unlock unique capabilities not available in the upstream version of QEMU, such as system call and basic block tracing, and guest OS function hooking.

2.4 Binary Reverse Engineering

Binary reverse engineering is a process that aims to reveal the functionality of a binary target without accessing the target's source code. In this section, I provide information about two binary reverse engineering tools, Ghidra and Ghidriff that are used by my re-

hosting and analysis frameworks.

2.4.1 Ghidra

Ghidra (NSA Research Directorate, 2025) is a state-of-the-art binary reverse engineering tool. Similar to the other reverse engineering systems (Vector 35, 2024; Hex-Rays, 2024; Radare, 2024), Ghidra offers various types of analyses, such as disassembling and decompiling binary code, extracting metadata information (e.g, the endianness, the architecture and debugging information – DWARF), binary rewriting, etc. In addition, alongside the main analysis tool, Ghidra also offers a built-in binary diffing utility, called `Ghidra Version Tracking` (GVT). GVT's primary functionality is to compare two (similar) binaries and by using different correlators (e.g., *Exact Function Bytes Match*, *Exact Function Instructions Match*, *Similar Symbol Name Match*, etc), match the functions between the compared binaries. As the names suggest, these example correlators match functions (between the compared binaries) if the bytes comprising their body are identical, match functions if they consist of the same instructions, and if the function names match, respectively. I note here that the use of these correlators enables GVT to match functions that are not 100% identical, and might contain changes (e.g., patches) between the two binaries.

Furthermore, users have the option to instrument Ghidra's whole analysis pipeline through Python. Specifically, they can execute Ghidra in headless mode (via Python scripts) and run either the native and/or their own custom analysis on the target binary without invoking Ghidra's GUI. Of course this capability benefits scalability since multiple binaries can be analyzed automatically without human intervention.

2.4.2 Ghidriff

Ghidriff (Clearbluejar, 2023) is a command line binary diffing tool designed for fast and effective comparison of similar binaries (i.e., multiple versions of the same binary). Essentially, Ghidriff is a Python wrapper on top of Ghidra's headless mode that also leverages

GVT for binary diffing. Originally, GVT was tied to Ghidra's GUI and was unavailable in Ghidra's headless mode. Ghidriff is the first system that disassociates GVT's functionality from Ghidra's GUI to enable binary diffing in the headless mode.

Note that Ghidriff goes beyond merely running GVT in headless mode and also generates additional information. For example, Ghidriff also provides the similarity ratio of matched functions (i.e., decompiled code, instruction, basic block similarity ratio, etc), code diffs of modified functions, the names and code of added and removed functions between the compared binaries.

2.5 Fuzzing

Fuzzing is a dynamic analysis technique that feeds random input to a program in order to trigger unexpected behavior, such as crashes. It is a very popular technique that gained a lot of attention in recent years, especially when it comes to analyzing the Linux kernel for bugs and vulnerabilities. Below, I provide details about TriforceAFL and syzkaller, the two most popular kernel fuzzers that my frameworks used to analyze the IoT kernel modules for bugs and vulnerabilities.

2.5.1 TriforceAFL

TriforceAFL (NCC Group Plc, 2017) is a kernel fuzzer designed for system call fuzzing on Linux x86 and ARM kernels. It consists of two components, the fuzzer and the user-level agent that runs within a QEMU guest (that uses the target kernel) and is responsible for executing system call sequences provided by the fuzzer. The communication between the user-level agent and the fuzzer is conducted through the QEMU hypervisor (Bellard, 2005). The user-level agent uses an invalid instruction (hypercall) that is trapped by QEMU, to request input (e.g., the system call chain to execute). QEMU in turn, is responsible for tracing the kernel level Translation Blocks (TBs) corresponding to the executed system

calls and provide this coverage feedback to the fuzzer through a shared memory.

2.5.2 Syzkaller

Syzkaller (Google, 2023) is the most popular fuzzer for the Linux kernel and part of Google's OSS-Fuzz platform (Google, 2016) designed to continuously fuzz-test open-source projects. As a grammar-based fuzzer, syzkaller depends on manually crafted or automatically generated system call descriptions (i.e., instructions on how syzkaller should execute system calls) to fuzz-test Linux kernel's subsystems. Similar, to other greybox fuzzers, such as AFL/AFL++ (Fioraldi et al., 2020; Zalewski, 2017) syzkaller benefits from coverage feedback to improve its mutation strategies and increase the code covered in the kernel. Specifically, syzkaller uses the source-based KCOV (Vyukov, 2025) instrumentation to obtain the essential coverage feedback.

2.6 Deadlocks

Deadlocks are scenarios where a group of two or more (user or kernel) threads are blocked waiting for each other to release shared resources (e.g., locks), and none of them can proceed (Litux, 2005). Deadlocks occur only when the "Coffman conditions" are met; *Mutual Exclusion*, *Circular Wait*, *Hold and Wait*, and *No pre-emption* (GeeksforGeeks, 2025). The simplest case of a deadlock is the *ABBA* deadlock, as shown in Figure 2·1. In this scenario two (or more) threads (T_A and T_B) acquire two (or more) locks in reverse order (T_A: $Lock_A$->$Lock_B$, T_B: $Lock_B$->$Lock_A$), leading to a circular wait and a deadlock. There are also other types of deadlocks, such as double locking (i.e., acquiring the same lock twice without releasing it first) and IRQ-unsafe locking (i.e., acquiring a lock in and out of an IRQ context without disabling IRQs first).

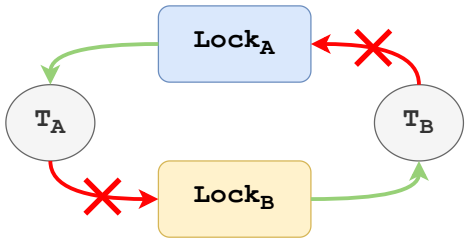


Figure 2·1: Simple case of an ABBA deadlock. T_A acquires $Lock_A$ (green) and T_B acquires $Lock_B$ (yellow). Next, both T_A and T_B will block indefinitely while trying to acquire $Lock_B$ and $Lock_A$, respectively, since these locks are not available for acquisition.

2.6.1 Lockdep

Introduced in 2006, Lockdep (Molnar and Van de Ven, 2025) is part of the Linux kernel and provides mature mechanisms to detect potential deadlocks throughout the kernel at runtime. Internally, Lockdep mainly consists of two components; 1) the meta-information, called *lockdep maps* which are used to trace which locks (e.g., `mutexes`, `spinlocks`, etc) are *acquired/released* and in what order. Once enabled in the kernel's configuration (usually only for debug builds due to significant performance overheads), the lockdep maps are embedded as members within all locking data structures (e.g., `struct mutex`). 2) the lockdep annotations (code specific to Lockdep), such as the `mutex_<acquire/release>` macros, which trigger Lockdep's deadlock detection logic based on the locking operation (e.g., *acquire/release*). The lockdep annotations and in turn Lockdep's deadlock detection logic process the lockdep metadata. Interestingly, Lockdep has no notion of the locks (instances) used in the kernel, but operates on *lock classes* instead.

A *lock class* represents a category of locks that are conceptually identical (i.e., have the same type, variable name, and originate from the same context – e.g., are members of a specific data structure), even though there may be thousands of individual instances of these locks. For instance, the mutex (type) lock named `param_lock` (name) in `struct module` (origin) represents one lock class. As the kernel associates one `struct module` with each loaded kernel module, generally multiple `struct modules` exist at the same

time. However, all instances of `param_lock` belong to the same class. Within the kernel code, a lock class corresponds to a unique identifier called “lock class key”. Each lockdep map corresponding to a lock stores the lock’s lock class key (or lock class). Upon the use of a lock (class), Lockdep detects if the necessary conditions for a deadlock (e.g., a ABBA locking scenario) are met, and if so, issues a warning, called “splat” (similar to a kernel "Oops") featuring detailed debug information (e.g., dependency chains and identifiers of the involved locks). Note that Lockdep does not have to trigger an actual deadlock to generate a warning but merely requires the kernel code of two (or more) regions that might contend for a shared resource (i.e., a lock) to execute in any order, in a race-free context. Hence, the bugs found by Lockdep are considered potential deadlocks (the actual deadlock does not have to occur for Lockdep to detect it). Lockdep maps must be compiled into the lock data structures used by the kernel proper and all modules. Since the source-code of proprietary binary-only IoT kernel modules is by definition not available for re-compilation, Lockdep as implemented in the Linux kernel cannot be applied to such modules preventing security researchers from identifying deadlocks therein.

Chapter 3

Related Work

In this chapter, I discuss prior work on firmware re-hosting and analysis. I structure the related work as follows. First, I provide information about prior works that focus on reverse engineering the configuration and data structure memory layouts of data structures in binary kernels. Next, I elaborate on prior firmware analysis studies, as well as benchmarking techniques that aim to compare different analysis approaches with each other. Finally, I present information about prior works on deadlock detection.

3.1 Kernel Reverse Engineering

There are plenty of prior works that aim to reverse engineer the configuration of pre-compiled binary kernels or identify the lost information about the memory layout of data structures. In this section, I present works in both categories and compare them with `FirmSolo` (see Chapter 4) and `FirmDiff` (see Chapter 6), which also reverse engineer the binary IoT kernels to recover their configuration and the memory layout of their data structures.

3.1.1 Kernel Configuration Recovery.

Layout Expert (Socała and Cohen, 2016) and Autoprofile (Pagani and Balzarotti, 2022) leverage memory footprints of live kernels and memory dumps, respectively, to recover the configuration of pre-compiled kernels. Unfortunately, it is not only costly to purchase thousands of IoT devices but it would also require extensive manual effort to generate

the memory dumps. In addition, experimental evidence shows that the original kernels used in IoT firmware cannot directly be booted in a full-system emulator such as QEMU. The main reason is that these kernels are compiled for Systems-on-Chip (SoCs) that are not supported by the emulator. In particular QEMU does not support the peripherals in the SoCs, thus causing any attempt of booting a device's kernel in the emulator to fail. `FirmSolo` overcomes the above limitations by using K.C.R.E. to generate its own kernels that are supported by QEMU to load the IoT kernel modules.

3.1.2 Data Structure Inference.

Dynamic analysis systems, such as REWARDS (Lin et al., 2010) , HOWARD (Slowinska et al., 2011) and DSIbin (Rupprecht et al., 2017) mostly rely on memory access patterns with the help of symbolic execution to deduce the layout of data structures. Static analysis systems, such as OSPREY (Zhang et al., 2021) and hybrid systems such as TIE (Lee et al., 2011) and ORIGEN (Feng et al., 2016) rely on data and control flow analyses to recover the layout of data structures in binaries. Unfortunately, both the absence of semantic information in the IoT binaries and/or the requirement to emulate the IoT kernels (which are not supported by state-of-the-art emulators), pose great limitations in discovering the layout of the data structures used by the IoT kernel modules. Specifically, none of the systems mentioned above, except ORIGEN, recover semantic information about data structures (i.e., names and members) like `FirmSolo` does (i.e., through inter-module debugging information transferring). Without this feature `FirmSolo` would not be able to recover or align the layout of data structures in its custom kernels. However, not even ORIGEN can completely replace `FirmSolo`'s data structure layout recovery facility since ORIGEN's dynamic analysis component, essential to its functionality, suffers from the dynamic analysis limitations stated previously. In addition, my other contribution `FirmDiff`, provides a fast and effective static analysis approach, which overcomes the aforementioned limitations. First, `FirmDiff` does not require to emulate the IoT firmware kernels and

second by using the debugging information embedded in the open-source kernel modules compiled by `FirmSolo`, `FirmDiff` is able to recover the type and (partially) the layout of data structures used by the IoT kernel modules.

3.2 Firmware Analysis

In this section, I detail prior studies on firmware analysis, grouped under three categories; static, hybrid (hardware-in-the-loop), and dynamic analysis approaches. Unlike static analysis techniques, which do not require executing the target firmware, both hybrid and dynamic firmware analysis techniques rely on re-hosting to conduct their analysis (e.g., fuzzing) on the target firmware.

3.2.1 Static Firmware Analysis

There are plenty of academic works focusing on this research area. Static analysis and symbolic execution (Davidson et al., 2013; Renzelmann et al., 2012; Shoshitaishvili et al., 2015; Hernandez et al., 2017; Shoshitaishvili et al., 2016) are popular techniques used for analyzing Linux based drivers and binary firmware. Static analysis is essential to guide symbolic execution engines (Cadar et al., 2008; Chipounov et al., 2011; Cha et al., 2012), which serve as a replacement mechanism in the absence of real hardware. These systems require extensive manual effort from the analyst to integrate the symbolic execution engines with different types of embedded devices. In contrast, `FirmSolo`'s (see Chapter 4), `Pandawan`'s (see Chapter 5), and `LL`'s full-system emulation and fully automated analysis pipelines make these frameworks applicable to many different types of IoT devices and their privileged firmware code.

3.2.2 Hardware-In-The-Loop

Systems in this category follow a hybrid approach which relies on both re-hosting and the physical IoT device to analyze IoT firmware code. AVATAR (Zaddach et al., 2014),

PROSPECT (Kammerstetter et al., 2014), Inception (Corteggiani et al., 2018), and SURROGATES (Koscher et al., 2015) are examples of systems that forward I/O operations to the physical device through its debugging interface (e.g., JTAG) while firmware code is execute and analyzed within the emulator (e.g., QEMU). While these systems can effectively re-host and analyze IoT firmware code, they are intrinsically limited in terms of scalability due to their dependence on the IoT hardware.

3.2.3 Firmware Re-hosting

Costin et al. (Costin et al., 2016) is one of the first works that pioneered firmware re-hosting. They employed QEMU's (Bellard, 2005) userland emulation and `chroot` to execute and analyze the services (e.g., webservers) within the file-systems of firmware images. Chen et al. (Chen et al., 2016) and Kim et al. (Kim et al., 2020) developed Firmadyne and FirmAE, respectively, two dynamic analysis frameworks that use firmware re-hosting to discover bugs and vulnerabilities in user space applications of firmware images. In a similar fashion Vetterl et al. (Vetterl and Clayton, 2019) developed Honware, a honeypot framework that leverages firmware image re-hosting to imitate the behavior of real-network connected devices and study real world attack scenarios by deploying these devices on the Internet. All these works focus on user space analysis while lacking support for firmware kernel modules. FirmGuide (Liu et al., 2022) is a re-hosting framework that semi-automatically creates QEMU peripheral models to successfully re-host the Linux IoT kernels of open-source firmware images (e.g., OpenWRT). However, both the need for human intervention and source code availability renders FirmGuide unable to re-host proprietary firmware. In contrast my firmware re-hosting contributions, produce kernels that are able to load open and closed-source kernel modules and enable downstream analysis on both user and privileged firmware code.

3.2.4 Firmware Fuzzing

Both Greenhouse (Tay et al., 2023) and EQUAFL (Zheng et al., 2022) rely on user mode QEMU to emulate user level firmware applications and analyze these applications through fuzzing. EASIER (Pustogarov et al., 2020) is a framework that loads binary kernel modules on pre-configured Android kernels within an emulated environment and analyzes these modules with AFL (Zalewski, 2017). FirmAFL (Zheng et al., 2019) combines a hybrid firmware re-hosting technique (user and full system mode) with fuzzing to analyze user level firmware code. Fuzzware (Scharnowski et al., 2022) is a system that relies on dynamic symbolic execution (DSE) to infer hardware created values (provided to the firmware through MMIO) which are used to drive a fuzzer while analyzing the embedded firmware code. On a similar fashion, Halucinator (Clements et al., 2020) uses Hardware Abstraction Layers (HALs) to model hardware peripherals along with a fuzzer to dynamically analyze embedded firmware.

These systems are unable or only suitable of analyzing specific categories of firmware kernel modules. In contrast, my firmware re-hosting and analysis frameworks can load and analyze a variety of binary Linux IoT kernel modules.

3.2.5 Benchmarking

These studies are important for establishing reliable metrics for evaluating the contributions of works and also setting the standards for future directions in a research area. Both (Klees et al., 2018; Li et al., 2021) introduce metrics to evaluate the performance and effectiveness (i.e., bug finding) of fuzzing tools as well as guidelines that should be followed by future fuzzing applications. AIR (Zhang and Sekar, 2013) proposes a metric to quantify the protection offered by Control Flow Integrity techniques on binary executables. Similar to these works, the `FICD` technique, implemented by `Pandawan` (see Chapter 5), uses specific metrics (i.e., the number of user level, user and kernel module code coverage and number

of kernel modules loaded) to evaluate different approaches in the full-system re-hosting domain.

3.3 Deadlock Detection

A subset of works (Muller, 2024; Ye et al., 2024; Bai et al., 2022; Cai et al., 2022; Cai et al., 2023; Brotherston et al., 2022; Engler and Ashcraft, 2003) in this category use static analysis to discover deadlock-based bugs in multithreaded user level or kernel level code. Dynamic approaches (Eslamimehr and Palsberg, 2014; Bartocci et al., 2013; Seyster et al., 2011; Zhou et al., 2017; Joshi et al., 2010; Cai et al., 2020) opt to dynamically execute the target code and detect potential deadlocks at runtime. However, when it comes to deadlock analysis in the kernel, Lockdep (Molnar and Van de Ven, 2025) is the most preferred system due to its maturity and out-of-the box compatibility with the kernel code. Unlike `LL` (see Chapter 7) though, all these approaches are not capable of analyzing binary-only IoT kernel modules due to their dependency to source code availability.

Chapter 4

Enabling dynamic analysis of binary Linux-based IoT kernel modules

In this chapter, I discuss my first contribution to improving firmware re-hosting. First, I present the assumptions that need to be satisfied so that my framework successfully re-hosts the IoT kernel modules within firmware images. Next, provide information about the design of my re-hosting framework in Section 4.2, and then I detail `FirmSolo`, the prototype implementation of my re-hosting approach in Section 4.3. Finally, I detail the evaluation of `FirmSolo`'s ability to re-host binary IoT firmware kernel modules, and expose these modules to dynamic vulnerability analysis.

4.1 Assumptions

Experimental evidence shows that the original kernels used in IoT firmware cannot directly be booted in a full-system emulator such as QEMU. The main reason is that these kernels are compiled for Systems-on-Chip (SoCs) that are not supported by the emulator. In particular QEMU does not support the peripherals in the SoCs, thus causing any attempt of booting a device's kernel in the emulator to fail.

Furthermore, I assume that IoT vendors rarely modify the code of their open-source LKMs also present in the upstream kernel source tree of the kernel version used in their products. This assumption is essential for the Data Structure Layout Correction step (see Section 4.3.2) and implies that open-source IoT firmware LKMs share the majority of their source code with their upstream counterparts in the kernel's repository. It also implies

that data structures defined in the kernel and used by *any* module, solely depend on the configuration options (i.e., the `.config` file) used to compile the kernel.

4.2 System Overview

In this section I provide the system overview of `FirmSolo`. Provided a target firmware image (I), the goal of `FirmSolo` is to produce a Linux kernel (K_{FS}) capable of loading the kernel modules ($\mathbb{D}_{KM}$) distributed within the file-system of I. K_{FS} can then be used by downstream systems to analyze these kernel modules for bugs and vulnerabilities.

A kernel module can only be loaded if the following three requirements are fulfilled. First, the kernel and the module must be of the same version. Second, all external symbols that a module depends on must be provided, either by the kernel proper or by another module that must be loaded first. Third, the layout of data structures shared between the kernel and the module must be consistent. Note that `FirmSolo` can only adjust K_{FS}, as the goal is to load the binary (proprietary and open-source) modules in $\mathbb{D}_{KM}$ unmodified. In the remainder of this section I will discuss how `FirmSolo` satisfies these three requirements. On the one hand, finding the right kernel version for a given module is easily possible due to the open-source nature of the Linux kernel via its git repository. On the other hand, the availability of symbols and layout of data structures depend entirely on the kernel's configuration, a challenge addressed by `FirmSolo`'s K.C.R.E. process.

Figure 4·1 illustrates `FirmSolo` as a fully-automated three-stage process: 1. ① The Information Gathering stage extracts static meta information about a firmware image (I), such as its kernel version and the set of modules $\mathbb{D}_{KM}$. 2. ② The Kernel Configuration Reverse Engineering (K.C.R.E.) stage then follows a hybrid and iterative approach to infer the configuration for a kernel (K_{FS}) that matches the configuration in the compilation of I's original kernel (K_0) 3. ③ The Downstream Analysis stage leverages the kernel (K_{FS}) from the above step to subject the modules found in image F_{image} to various security analyses.

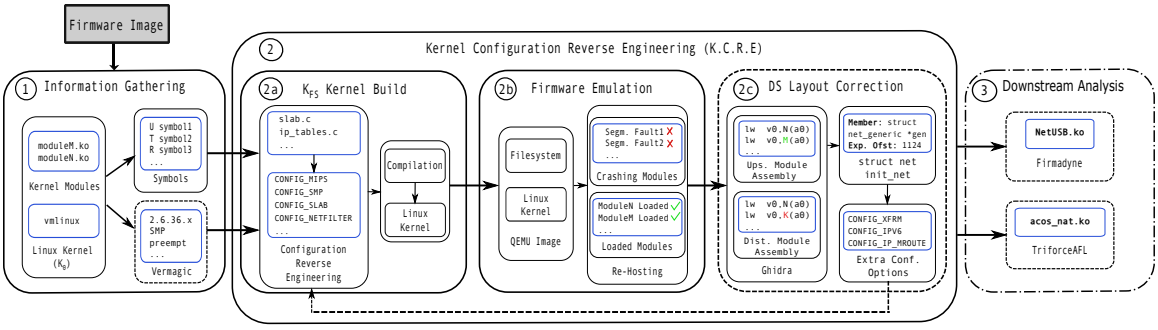


Figure 4·1: System overview of `FirmSolo`: The dashed boxes indicate that a step or information are optional. Also while stage 3 is not part of `FirmSolo`, I feature two representative downstream analysis.

As representative examples of such analyses I discuss the integration with TriforceAFL and Firmadyne. Clearly, further analysis systems can easily leverage the K_{FS} kernels generated by `FirmSolo`.

Note that while $\mathbb{D}_{KM}$ refers to the set of modules *distributed* in a firmware image, I use the symbol $\mathbb{U}_{KM}$ for the modules compiled by `FirmSolo` as a product of K.C.R.E. using the source code of the *upstream* Linux kernel. Per my assumptions in Section 4.1, a *subset of open-source modules* in $\mathbb{D}_{KM}$ will have corresponding modules in $\mathbb{U}_{KM}$, an insight I will exploit to align the layout of data structures in Section 4.2.2.

4.2.1 Information Gathering

In stage ①, `FirmSolo` gathers metadata about the image I and its $\mathbb{D}_{KM}$ modules. Examples of these metadata are the symbols (functions and data structures) required by the $\mathbb{D}_{KM}$ modules and the optional vermagic, which indicates what kernel source tree (version) `FirmSolo` should configure. I provide more detailed information about the metadata collected by `FirmSolo` in Section 4.3.1. The metadata collected in this stage are used in stage ② to configure and compile the K_{FS} kernel.

4.2.2 K.C.R.E.

The Kernel Configuration Reverse Engineering (K.C.R.E.) stage is the core of `FirmSolo` and aims to infer a configuration for K_{FS} that is an approximation of the configuration of K_0 to allow the modules found in image I to load and operate. Note that configuration options in K_0 that do not affect symbols used by the $\mathbb{D}_{KM}$ modules are not necessary for the modules' operation and hence are of no concern to `FirmSolo`. K.C.R.E. consists of three iterative steps: a) The K_{FS} Kernel Build, b) the Firmware Emulation and c) the Data Structure Layout Correction. I describe each step below.

(2a) K_{FS} ***Kernel Build:*** At its first iteration, this step infers an approximate kernel configuration based on the static metadata extracted in stage (1). This initial configuration will be refined in subsequent iterations.

`FirmSolo` performs seven tasks when building a kernel image in step (2a): 1) It consumes the information produced in stage (1). 2) It searches the source code of the target kernel version and identifies the C-source files that provide the implementations for the required symbols. 3) It detects the configuration options responsible for compiling these C-source files. 4) It also parses the C-source files and pinpoints additional configuration options that might guard the symbol definitions. 5) It enables these options in the configuration for K_{FS} and, if necessary, resolves any option dependencies. 6) K.C.R.E. identifies which (tristate) options guard the creation of the kernel modules ($\mathbb{D}_{KM}$) in image I and assigns these options the value m to emit the corresponding $\mathbb{U}_{KM}$ modules. Of course, this step only applies to modules with available source code in the kernel source tree. The outcome of the previous tasks is a kernel `.config` file. 7) Finally, `FirmSolo` compiles the K_{FS} kernel and its modules ($\mathbb{U}_{KM}$), using the generated `.config` file. As the kernel must be compiled for a specific hardware platform, `FirmSolo` chooses the `Malta` (MIPS) and `Versatile` and `Realview` (ARMv5 and ARM[v6/v7], respectively) development boards[1],

[1] Choosing a real-SoC will yield a kernel that runs only on said SoC, but will not be supported by QEMU (see Section 4.1)

as these are well-supported by QEMU (Bellard, 2005).

This procedure ensures that the symbols required by the modules in $\mathbb{D}_{KM}$ are provided either by other modules in $\mathbb{D}_{KM}$ or the kernel proper. However, the existence of all required symbols is necessary but not sufficient. Recall that for a binary module to work correctly, the memory layout of any data structures shared between the kernel proper and the module must be consistent. As the above process cannot guarantee the correct memory layout of shared symbols, step (2b) will identify the modules where mismatches exist and the Data Structure Layout Correction in (2c) will address them.

(2b) ***Firmware Emulation:*** To validate the accuracy of the `.config` produced in step (2a), `FirmSolo` assesses which modules in $\mathbb{D}_{KM}$ can be loaded correctly by K_{FS}. This is an iterative process where `FirmSolo` tries to load the modules in $\mathbb{D}_{KM}$ after it has resolved any module dependencies.

When a module $d \in \mathbb{D}_{KM}$ fails to load (e.g., due to a crash) it can prevent the loading of additional modules, if they rely on symbols exported by the offending module d. I refer to the set of modules that fail to load as $\mathbb{C}_{KM} \subseteq \mathbb{D}_{KM}$. To minimize this issue and ensure that a larger number of modules are available to downstream analysis in stage (3), `FirmSolo` implements a substitution mechanism where it substitutes $\mathbb{C}_{KM}$ modules with their counterparts from $\mathbb{U}_{KM}$. Of course, `FirmSolo` can only substitute modules that have a correspondence in the source tree of K_{FS}, and hence binary-only proprietary modules cannot be substituted. This mechanism allows me to assess the overall quality of the inferred kernel configuration, as `FirmSolo` attempts to load as many modules as possible. The output of step (2b) are the modules in $\mathbb{D}_{KM}$ that could be loaded successfully, the substitutions `FirmSolo` implemented during this step, along with the module-load-order of both.

(2c) ***Data Structure Layout Correction:*** The kernel modules interact with the kernel through symbols of typed data structures defined in the kernel proper. However, while step (2a) makes these symbols available in the K_{FS} kernel, it makes no effort to ensure that

```
1   struct net {
2         atomic_t                count;
3   ...
4   #if defined(CONFIG_NF_CONNTRACK) ||
5       defined(CONFIG_NF_CONNTRACK_MODULE)
6         struct netns_ct         ct;
7   #endif
8  ...
9         struct net_generic      *gen;
10  };
```

Listing 4.1: Code snippet of the implementation of `struct net` in the Linux kernel.

their memory layout (i.e., type) matches the layout the modules in $\mathbb{D}_{KM}$ expect. Specifically, if K_{FS} and a $\mathbb{D}_{KM}$ module were not compiled with the same configuration options, and by extension the same `.config` file, there might be a mismatch between the layout of a shared data structure within K_{FS} and the layout that the $\mathbb{D}_{KM}$ module expects. These symbol layout differences are one of the main reasons why modules in $\mathbb{C}_{KM}$ fail to load in step (2b).

For example, Listing 4.1 illustrates the definition of `struct net` in the Linux kernel. This `struct` contains several members that are guarded by configuration options, such as `ct` (line 9) which is guarded by the `CONFIG_NF_CONNTRACK` option (line 8). An example of a kernel variable (symbol) of type `struct net` is `init_net` (not shown in the Listing) which is exported to all networking modules. The layout of `init_net` (e.g., size and offset of its members) depends directly on the options in the `.config` that was used to compile the kernel. Thus, a module can only safely use `init_net` if it has been compiled with the same setting of `CONFIG_NF_CONNTRACK` as the kernel that is trying to load the module. Specifically, the offset of member `init_net->gen` (line 12) would be different if the `CONFIG_NF_CONNTRACK` settings were inconsistent and accessing that member would likely result in a crash.

`FirmSolo` addresses these errors by recovering the correct layout of the target data

structure s (e.g., `struct net` in Listing 4.1) by applying this step to all the modules ($\mathbb{C}_{KM}$) that crashed in step (2b). The purpose of step (2c) is twofold. First, it fixes crashing modules and reintroduces them into the analysis pipeline, subsequently making them available to downstream analysis (see Section 4.2.3). Second, it helps to prevent any additional errors later in the emulation related to misaligned data structures. If a module fails to load for an unrelated reason (e.g., because it tries to interact with a peripheral that is not emulated by QEMU), it gets discarded and `FirmSolo` proceeds with the next crashing module.

Step (2c) relies on binary static analysis to infer the data structure s that caused the crash along with its unaligned members. Afterwards, `FirmSolo` infers a set of configuration options that are used to refine the `.config` in step (2a) and build a new K_{FS} with s correctly aligned. Note that step (2c) works only for $\mathbb{C}_{KM}$ modules with an upstream counterpart in $\mathbb{U}_{KM}$ and not for proprietary modules (see Section 4.3.2). If successful, the new kernel will be able to load the modules that crashed due to differences in the memory layout of shared data structures.

4.2.3 Firmware Analysis

`FirmSolo`'s utility stems from the fact that it allows existing downstream analyses to broaden their scope into the analysis of Linux-based IoT firmware kernel modules. I demonstrate this aspect by integrating `FirmSolo` with two representative analysis systems; TriforceAFL (NCC Group Plc, 2017) and Firmadyne (Chen et al., 2016). TriforceAFL is a kernel fuzzer designed for system call fuzzing on Linux x86 and ARM kernels. By extending TriforceAFL with support for MIPS and combining it with `FirmSolo`, TriforceAFL can fuzz the kernel modules (open and closed-source) found in IoT firmware. Firmadyne is a vulnerability analysis system targeting user space applications in firmware images but lacks any kernel-level analysis capabilities. With the assistance of `FirmSolo`, Firmadyne supports the analysis of closed and open source kernel modules found in IoT firmware.

4.3 Implementation

I next discuss the implementation details of `FirmSolo`.

4.3.1 Information Gathering

Stage ① consumes the extracted file-system of the original firmware image F_{image}, collects the set of $\mathbb{D}_{KM}$ kernel modules from F_{image} and infers 1) the endianess and the architecture of the target system, 2) the symbols that the $\mathbb{D}_{KM}$ modules depend on and export, 3) the module vermagic and 4) the `KALLSYMS` entry from the K_0 kernel. Both vermagic and the `KALLSYMS` entry are optional information that `FirmSolo` takes advantage of if they are present. The endianess and architecture dictate the compiler-toolchain to compile the K_{FS} kernel and is extracted from the modules' ELF headers via `readelf`. The remaining information allows `FirmSolo` to infer the configuration options used by the vendor when compiling the original K_0 kernel.

As explained in Section 2.1.1, kernel modules can only be loaded and used if all symbols they rely on are either provided by another module (that must be loaded first) or by the kernel proper. Thus, to load modules, `FirmSolo` must infer these symbols and ensure they exist. To this end, `FirmSolo` first extracts two sets of symbols from the set of modules $\mathbb{D}_{KM}$ contained in F_{image}. The set $\mathbb{S}_{und}$ corresponds to the set of symbols imported by (or undefined in) any of the modules in $\mathbb{D}_{KM}$. Analogously, $\mathbb{S}_{exp}$ is the set of symbols exported by any of the modules in $\mathbb{D}_{KM}$. Of course, a given symbol can be undefined in one module and exported by another module and thus end up in both sets. For example, the symbol `gpl_usb_alloc_urb` marked as undefined by the proprietary `NetUSB` kernel module is defined and exported by the `GPL_NetUSB` module. Thus, $\mathbb{S}'_{und} = \mathbb{S}_{und} \setminus \mathbb{S}_{exp}$ is the set of symbols not provided by any module and these symbols must be provided by the kernel proper. `FirmSolo` extracts the status of symbols (undefined or exported) from the kernel modules' ELF headers via `nm`.

```
1  #define VERMAGIC_STRING \
2    UTS_RELEASE " "\
3    MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT\
4    MODULE_VERMAGIC_MODULE_UNLOAD MODULE_VERMAGIC_MODVERSIONS     \
5    MODULE_ARCH_VERMAGIC
```

Listing 4.2: Linux-2.6.30 vermagic macro defined in include/linux/vermagic.h

The vermagic contains information about the target kernel version and is embedded as a zero-terminated string in kernel modules which `FirmSolo` extracts with a simple regular expression. It also embeds 5 configuration options related to the target system (see Section 2.1.1 and Listing 4.2). Specifically, as illustrated in Listing 4.2 these options are related to `Symmetric Multiprocessing`, `Preemption`, `Module Versioning` support, etc. The `UTS_RELEASE` macro specifies the full kernel version (base version + extension). In case the vermagic is not present in a module, `FirmSolo` infers the kernel version from I's file-system at `/lib/modules/<kernel-version>`. Also, it attempts to (partially) recover the vermagic options in step (2a) of stage (2). Finally, if available, `FirmSolo` extracts the `KALLSYMS` entry from K_0 via `vmlinux-to-elf`.[2] The information collected above determines which configuration options must be enabled in K_{FS} in step (2a).

4.3.2 K.C.R.E.

The goal of stage (2) (K.C.R.E.) is to configure and build a K_{FS} Linux kernel that is compatible with the $\mathbb{D}_{KM}$ modules.

(2a) K_{FS} Kernel Build: In this step, `FirmSolo` performs three tasks: First, it checks out the correct version of the Linux kernel from `git.kernel.org`. Second, it configures the kernel pursuant to the static information extracted in stage (1). Third, and finally, `FirmSolo` uses the appropriate cross compiler toolchain to build the K_{FS} kernel and the modules in $\mathbb{U}_{KM}$. To preserve valuable information for subsequent steps (e.g., step (2c)), `FirmSolo` compiles

[2] https://github.com/marin-m/vmlinux-to-elf

the K_{FS} and all $\mathbb{U}_{KM}$ modules with debug symbols.

The `.config` generated by `FirmSolo` should satisfy the following two constraints: (1) the compiled K_{FS} kernel should export all symbols in S'_{und}, and (2) each module $d \in \mathbb{D}_{KM}$ that has an implementation in the kernel source tree should have its corresponding module $u \in \mathbb{U}_{KM}$ built too.

Enable Symbols. For each symbol $s \in \overline{\mathbb{S}}$ ($\overline{\mathbb{S}} = \mathbb{S}'_{und} \cup$ `KALLSYMS`[3]), `FirmSolo` must infer which configuration option enables s in the kernel. Thus, given s, `FirmSolo` first identifies the source file f that defines s. As the kernel exports symbols to its modules via the `EXPORT_SYMBOL[_GPL]` macros, `FirmSolo` uses `cscope` (Bell Labs, 2012) to search for expressions of the form `EXPORT_SYMBOL[_GPL](<s>)` to identify which file f implements s. Subsequently, `FirmSolo` analyzes the kernel's build system to identify which configuration option guards the compilation of f. That is, `FirmSolo` identifies the `Makefile` in the kernel's source tree that is responsible for building f and extracts the configuration option o (i.e., `CONFIG_<option name>`) that guards the compilation of f. Next, `FirmSolo` enables o in the `.config` file (initially being a minimal template `.config`). Finally, the definition of s can be further affected by additional configuration options within f. `FirmSolo` addresses these cases by parsing f and detecting any options that encapsulate (guard) the definition of s within f's body. As with o, `FirmSolo` also enables these new options in the `.config` file. While some configuration options can be directly enabled in the `.config` file, others are guarded by dependencies. Thus, to satisfy the constraints enforced over configuration options (expressed as logical formulae in the `Kconfig` files), `FirmSolo` leverages `Kconfiglib` (Magnusson, 2021) to first transform the `Kconfig` files into a constraint tree. In this tree, configuration options are the leaf-nodes and internal nodes are the logical operators (&&, ||, !). `FirmSolo` then enables (or disables for negations) the constraining options while walking the tree in pre-order traversal and finally enables the target option o.

Listing 4.3 shows an example for the symbol `malloc_sizes`. First `FirmSolo` detects

[3] If `KALLSYMS` could not be extracted, this set is the empty set

```
1  mm/slab.c:
2    struct cache_sizes malloc_sizes[]
3    EXPORT_SYMBOL(malloc_sizes);
4  mm/Makefile:
5    obj-$(CONFIG_SLAB) += slab.o
```

Listing 4.3: `CONFIG_SLAB` option detection. The listing shows code snippets from two files `mm/slab.c` (1-3) and `mm/Makefile` (4-5) with only the relevant lines retained.

that the `malloc_sizes` symbol is exported by the *mm/slab.c* file, via *cscope* (lines 1 and 4). Afterwards, `FirmSolo`, detects that the `Makefile` responsible for the compilation of *slab.c* is the *mm/Makefile* (line 5) and by parsing it detects the compilation of *slab.o*, is guarded by the option `CONFIG_SLAB`, in *mm/Makefile* (line 6).

Multiple Symbol Definitions. While the above mechanism works for most scenarios, some symbols have multiple definitions throughout the kernel and the configuration dictates which implementation will be used during compilation. For example the symbol `kmem_cache_alloc` is defined in both `mm/slab.c` and `mm/slub.c`, guarded by the mutually exclusive options `CONFIG_SLAB` and `CONFIG_SLUB`, respectively. To resolve such multiple definitions, `FirmSolo` relies on a heuristic that works as follows. While processing a symbol $s \in \overline{\mathbb{S}}$, let f_n be the set of files that each define s. `FirmSolo` will select whatever file provides the most *additional* symbols in $\overline{\mathbb{S}}$. That is, if $def(f)$ returns the set of symbols defined in file f, then `FirmSolo` chooses f according to $\max_{f \in f_n} |def(f) \cap \overline{\mathbb{S}})|$.

Enable $\mathbb{U}_{KM}$Modules. In addition to the configuration options that control the symbols in $\overline{\mathbb{S}}$, `FirmSolo` also configures the kernel to build the $\mathbb{U}_{KM}$ modules that correspond to the $\mathbb{D}_{KM}$ modules found in F_{image}. The modules in $\mathbb{U}_{KM}$ play an important role in the following steps. As the `Makefiles` in the kernel source tree indicate which configuration option guards the generation of each kernel module, `FirmSolo` simply sets the corresponding options to the value *m* (build as module) to trigger the compilation of the corresponding module. Of course, `FirmSolo` skips this step for proprietary modules. Finally, `FirmSolo`

selects the configuration options coupled to the vermagic extracted in stage ①. The vermagic encodes five configuration options that are not always present in the set of options related to symbols in $\overline{\mathbb{S}}$ (see Listing 4.2), since the vermagic options might not have any correspondence in these symbols. Thus, I manually encoded the relation between vermagic values and their configuration options into `FirmSolo`. In case the vermagic is not available, `FirmSolo` attempts to reconstruct it by searching the set of options related to symbols in $\overline{\mathbb{S}}$ for the presence of the encoded vermagic options. K.C.R.E. is an iterative process. Hence, during the iterative steps, additional information about configuration options that must be enabled to align data structures from step (2c) are included here. For the final phase of this step, `FirmSolo` compiles K_{FS} and its $\mathbb{U}_{KM}$ modules based on the generated `.config`.

(2b) Firmware Emulation: The goal of this step is to assess which modules in $\mathbb{D}_{KM}$ load in K_{FS} and which do not. To this end, `FirmSolo` first infers the correct module load-order induced by the dependencies between various modules (i.e., from the modules' `.modinfo` section). In a second step, `FirmSolo` loads these modules (according to the load-order) after booting a generic file-system based on buildroot (Petazzoni, 2021) in the QEMU system emulator. This file-system contains the $\mathbb{D}_{KM}$ modules and their $\mathbb{U}_{KM}$ counterparts compiled previously.

Module Substitution. If a module $c \in \mathbb{C}_{KM}$ causes an error during loading, `FirmSolo` deals with the offending module, by invoking the substitution mechanism which works as follows. If c does not load due to a failure in one of the module loading stages (see Section 2.1.2), `FirmSolo` simply loads the $u \in \mathbb{U}_{KM}$ counterpart of c instead (if it exists). However, if c triggers a kernel `Oops`, `FirmSolo` proceeds to restart the QEMU instance. This restarting mechanism is necessary as the `Oops` could bring the kernel into an inconsistent state (e.g., leave drivers in a semi-initialized state). In these cases `FirmSolo` applies the substitution process after the restart. If there is no substitution for c, `FirmSolo` skips loading the module after the restart.

Except enabling more $\mathbb{D}_{KM}$ modules to load (see Section 4.4.2), the substitution mechanism also helps to unearth additional offending modules since more modules reach step 6 (invoking `init_module`) of the module loading stages. `FirmSolo` attempts to address the module crashes in step (2c). Finally, `FirmSolo` preserves the information about the successfully loaded modules in $\mathbb{D}_{KM}$ and their substitutions for use by downstream analysis (see Section 4.3.3).

(2c) Data Structure Layout Correction: In step (2c), `FirmSolo` addresses crashes for $\mathbb{C}_{KM}$ modules that arise from the misaligned data structures these modules rely on. Specifically, using as basis the data structure layout that a module c expects, `FirmSolo` attempts to match the same layout in the K_{FS} kernel. Unfortunately, the $\mathbb{C}_{KM}$ modules are in binary form and stripped, whereas their $\mathbb{U}_{KM}$ counterparts are compiled with debugging information (i.e., DWARF) which preserves the layout of data structures as metadata. Using this observation I can address the above challenge as follows: 1) transfer debugging information from the $\mathbb{U}_{KM}$ to the $\mathbb{C}_{KM}$ modules to pinpoint the data structure misalignment and 2) adjust the data structure layout in K_{FS} and its $\mathbb{U}_{KM}$ modules, via their mutual build process. Note that this step is not intended to recover the memory layout of all the data structures that the $\mathbb{C}_{KM}$ modules use but only align the ones responsible for the crashes. Finally, this step also addresses two special cases not addressed by the generic algorithm described below: 1) for `struct module` and 2) for `struct kmem_cache`. `FirmSolo` implements the data structure alignment process via the following automated analysis.

Layout Recovery. Given a crashing module $c \in \mathbb{C}_{KM}$ and the corresponding module $u \in \mathbb{U}_{KM}$, 1) Identify the function $f_c \in c$ that caused the crash. 2) Identify the corresponding function $f_u \in u$. 3) Extract the access patterns to all variables in f_c and f_u. 4) Match the variables between f_c and f_u. 5) Identify `struct` variables. 6) Find mismatching accesses to members in these `structs`. 7) Identify the options that affect the offset of mismatched members. 8) Choose the options such that the misaligned members become aligned.

To implement these steps, FirmSolo first identifies the crashing function f_c from the kernel's Oops message. It is straightforward to identify f_u as it shares the same name with f_c. To transfer the debugging information from f_u to f_c, FirmSolo extracts variable accesses similarly to the static analysis presented in ORIGEN (Feng et al., 2016). Specifically, FirmSolo uses a Ghidra (NSA Research Directorate, 2025) script to lift the assembly code (MIPS or ARM) of f_c to PcodeOP (i.e., Ghidra's Intermediate Language). PcodeOP enables FirmSolo to directly extract all the variables used in f_c along with their def-use chains. A def-use chain consists of a variable definition and all the instructions accessing the variable which can be reached from its definition. FirmSolo processes the chains and only retains the memory access instructions, since they indicate a variable member access (through a constant displacement off of the base of the variable). The rationale to focus on these instructions is that compilers translate the access to a struct's member as accesses with constant displacements.

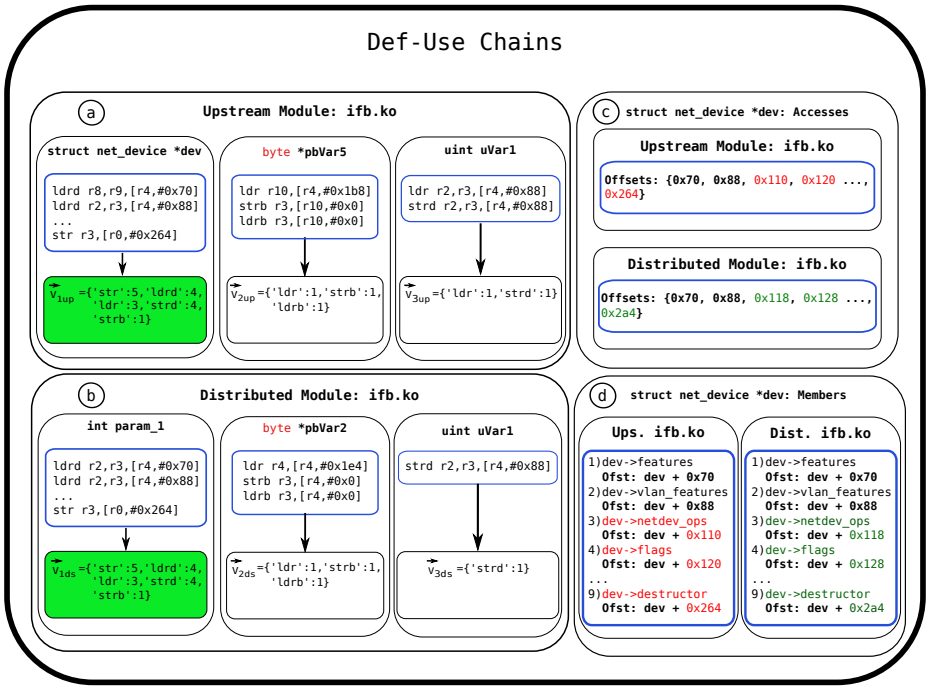
FirmSolo applies the same process to function f_u, however note that f_u's variables are annotated with debugging information which preserves their data types. Figure 4·2 provides an example of how FirmSolo extracts the variable def-use chains of the crashing function ifb_setup in both the $\mathbb{D}_{KM}$ and $\mathbb{U}_{KM}$ versions of module ifb.ko. Listings 4·2a and 4·2b illustrate the Ghidra decompilation view of the crashing function ifb_setup in the $\mathbb{U}_{KM}$ and $\mathbb{D}_{KM}$ versions of module ifb.ko, respectively. It is clear that in the case of the $\mathbb{U}_{KM}$ ifb.ko, Ghidra retains the debugging information for the variable dev of type struct net_device, which is used during the alignment process. In turn, parts ⓐ and ⓑ in Figure 4·2c show the extraction and abstraction of def-use chains to vectors $\vec{v}$ of memory instructions. In detail, the def-use chain of the variable dev in the $\mathbb{U}_{KM}$ ifb.ko is abstracted to the vector $\vec{v}_{1up}$ of five store instructions, four load double word instructions, etc. In my example, the variables dev, pbVar5 and uVar1 of the $\mathbb{U}_{KM}$ ifb.ko are matched to variables param_1, pbVar2 and uVar1 of the $\mathbb{D}_{KM}$ ifb.ko, based on the similarity of

```
1  void ifb_setup(net_device *dev)
2  {
3    undefined4 uVar1;
4    uint uVar2;
5    uint uVar3;
6    uint uVar4;
7    byte *pbVar5;
8    uint uVar6;
9
10   dev->destructor = free_netdev;
11   dev->netdev_ops = &ifb_netdev_ops;
12   ether_setup();
13   uVar3 = *(uint *)&dev->features;
14   uVar4 = *(uint *)((int)&dev->features + 4);
15   uVar6 = dev->flags;
16   uVar2 = dev->priv_flags;
17   pbVar5 = dev->dev_addr;
18   uVar1 = *(undefined4 *)((int)&dev->
           vlan_features + 4);
19   ...
20   dev->tx_queue_len = 0x20;
21   dev->flags = uVar6 & 0xffffefff | 0x80;
22   dev->addr_assign_type = '\x01';
23   dev->priv_flags = uVar2 & 0xffbefbff;
24   get_random_bytes(pbVar5,6);
25   *pbVar5 = *pbVar5 & 0xfe | 2;
26   return;
27 }
```

(a) $\mathbb{U}_{KM}$ Ghidra snippet

```
1  void ifb_setup(int param_1,undefined4 param_2,undefined4
       param_3,undefined4 param_4)
2  {
3    uint uVar1;
4    byte *pbVar2;
5
6    *(code **)(param_1 + 0x2a4) = free_netdev;
7    *(undefined **)(param_1 + 0x118) = &DAT_000107f0;
8    ether_setup();
9    *(undefined4 *)(param_1 + 0x250) = 0x20;
10   *(uint *)(param_1 + 0x70) = *(uint *)(param_1 + 0x70) |
         0x1900e9;
11   *(uint *)(param_1 + 0x74) = *(uint *)(param_1 + 0x74) |
         0x80;
12   uVar1 = *(uint *)(param_1 + 0x88) | 0x190069;
13   *(uint *)(param_1 + 0x88) = uVar1;
14   *(undefined4 *)(param_1 + 0x8c) = *(undefined4 *)(
         param_1 + 0x8c);
15   *(uint *)(param_1 + 0x128) = *(uint *)(param_1 + 0x128)
         & 0xffffefff | 0x80;
16   *(uint *)(param_1 + 300) = *(uint *)(param_1 + 300) &
           0xffbefbff;
17   *(undefined *)(param_1 + 0x164) = 1;
18   pbVar2 = *(byte **)(param_1 + 0x1e4);
19   get_random_bytes(pbVar2,6,uVar1,1,param_4);
20   *pbVar2 = *pbVar2 & 0xfe | 2;
21   return;
22 }
```

(b) $\mathbb{D}_{KM}$ Ghidra snippet



(c) Variable and member matching during the Data Structure Layout Correction step.

Figure 4·2: Ghidra code snippets for the upstream and distributed versions of the `ifb.ko` kernel modules. The figure also presents the process behind transferring DWARF information between the upstream and distributed version of the `ifb.ko` kernel module.

their instruction vectors (i.e., $\vec{v}_{1up} \approx \vec{v}_{1ds}$, $\vec{v}_{2up} \approx \vec{v}_{2ds}$ and $\vec{v}_{3up} \approx \vec{v}_{3ds}$), Next, `FirmSolo` discards all variable pairs without a valid `struct` type. Since only variable `dev` has a `struct net_device` type, only the pair `dev` and `param_1` will be retained by `FirmSolo`. Thus, `struct net_device` is the offending data structure that `FirmSolo` has to "fix". Part ⓓ of Figure 4·2c showcases which members of `struct net_device` are misaligned in my example (the figure only showcases three misaligned members). In particular, members `netdev_ops`, `flags` and `destructor` are located at offsets 0x110, 0x120 and 0x264 (hex), respectively, from the base of the `struct net_device`, while they should be at offsets 0x118, 0x128 and 0x2a4 (hex). Finally, `FirmSolo` invokes the backtracking search algorithm providing the offsets mentioned above. The algorithm searches for options that need to be added or removed from the K_{FS} kernel's `.config` file so that `struct net_device` is aligned in K_{FS} and its $\mathbb{U}_{KM}$ modules after a new iteration of K.C.R.E..

`FirmSolo` proceeds to map the variables and thus data types from f_u to f_c by comparing the `def-use` chains of the variables between functions f_u and f_c. In this case each variable's `def-use` chain can be viewed as a vector of memory instruction types and their number of occurrences within the chain (e.g., three load-byte `lb`, two store-word `sw` instructions, etc). Variables with the most similar `def-use` chains between the two functions are mapped together. Since the data types of variables in f_u are known, `FirmSolo` is in turn able to transfer this information to the variables in f_c based on the newly found mappings. As layout differences only occur in `struct` data-types, `FirmSolo` only preserves matches between variables that have such a type.

Next, `FirmSolo` identifies which of the remaining variable pairs represents the offending data structure s. `FirmSolo` focuses on the offsets (i.e., the member accesses) at which a variable v_c and its match v_u are accessed in f_c and f_u, respectively. Different offsets indicate a layout misalignment. In particular, for each `struct` variable pair v_c and v_u, `FirmSolo` reduces their `def-use` chains into the constant displacements used in the memory instruc-

tions that constitute the chains, forming two integer sets. After sorting both sets, FirmSolo identifies a misaligned member access where the displacements in both sequences differ (see Figure 4·2c). When this phenomenon occurs for a pair v_c and v_u the struct type assigned to that pair is considered the offending data structure s that FirmSolo has to fix. Again, the debugging information available for variable v_u is used to infer which member of v_c was accessed with a different offset in f_c.

With the misaligned members inferred, FirmSolo analyzes the kernel source to identify which configuration options affect the layout of the struct s that contains the members (and recursively any contained structs). The set of these configuration options induce the search space that FirmSolo must explore to find an assignment of options that correctly align s's members that caused the Oops to begin with. To this end, FirmSolo implements a backtracking search algorithm to efficiently and repeatedly 1) toggle a configuration option, 2) build a new kernel module u, and 3) confirm whether the struct members are now aligned.

Here I denote that a module c can invoke functions from its dependencies in $\mathbb{D}_{KM}$, thus the crashing function f_c might not be contained in c. FirmSolo handles these cases, using the same process as described above. In addition, despite the observation that most $\mathbb{C}_{KM}$ modules are stripped, function names are readily available from section names, when the ffunction-sections compiler flag was used by the vendors to compile the $\mathbb{C}_{KM}$ modules and their dependencies. In my dataset (see Section 4.4.1) 56% of the images feature this characteristic. Using the name of f_c, FirmSolo can easily identify the corresponding f_u.

Special Case: `struct module`. Before proceeding with s's layout recovery process described above, FirmSolo first checks if the struct module is aligned between c and u. This struct is a kernel module's representation within the kernel. Its correct alignment between c and u is necessary since a misaligned access might cause a crash even before the invocation of init_module (see Section 2.1.2). Luckily, struct module is directly

encoded in the module binary under the `.gnu.linkonce.this_module` section. Again using Ghidra, `FirmSolo` is able to extract the size and the offsets of two pointers, members, of `struct module`; the module's `init_module` and `cleanup_module` functions. The size indicates the last member of `struct module`. This information is sufficient for `FirmSolo` to align `struct module` between c and u, using the same process described above.

Special Case: `kmem_cache_alloc`. Finally, `FirmSolo` handles a special case if an error occurs in the `kmem_cache_alloc` kernel function, responsible for allocating memory for kernel objects. This function takes an argument of type `struct kmem_cache`. Through manual inspection I observed that only two options (`ZONE_DMA` and `SLUB_DEBUG`) affect the layout of `struct kmem_cache`, so `FirmSolo` simply checks if any of the two options correct the `struct`'s layout. Once `FirmSolo` infers the configuration options that align the data structures in the $c \in \mathbb{C}_{KM}$ module, this information is provided to step (2a) for the next iteration.

Proprietary modules. This alignment only works for modules that have an open-source counterpart in $\mathbb{U}_{KM}$, and thus not for proprietary modules. However, any proprietary module that relies on data structures also used by *any* of the modules in $\mathbb{U}_{KM}$ would transitively benefit from aligning the layout in these modules. This is due to the requirement that the data structure layout between *all* (proprietary and open-source) modules in the firmware image must be consistent.

4.3.3 Firmware Analysis

To demonstrate the utility of `FirmSolo`, I provide two representative use cases of dynamic analysis tools that benefit from `FirmSolo`; TriforceAFL (NCC Group Plc, 2017) and Firmadyne (Chen et al., 2016).

TriforceAFL. The TriforceAFL kernel fuzzer only supports x86 and ARM. However, and consistent with prior work, the majority of IoT firmware in our dataset runs on the MIPS architecture. Thus, I modified the QEMU sources for MIPS to support the AFL

instrumentation, fork-server, and hyper call, thereby enabling the use of TriforceAFL with MIPS. Furthermore, I modified the user-space agent that assists the fuzzer within the guest, to only fuzz the IOCTL system call. The rationale is that the vast majority of system calls are handled by the kernel proper. However, `FirmSolo` focuses on functionality contained in the $\mathbb{D}_{KM}$ kernel modules commonly invoked through the IOCTL system call.

```
1  struct ifreq {
2  #define IFHWADDRLEN     6
3          union
4          {
5              char    ifrn_name[IFNAMSIZ];
6          } ifr_ifrn;
7
8          union {
9              struct  sockaddr ifru_addr;
10 ...
11             struct  if_settings ifru_settings;
12         } ifr_ifru;
13 };
```

Listing 4.4: Code snippet of the implementation of *struct ifreq* in the the Linux kernel.

The IOCTL is a flexible system call, which exposes an interface that can be used by user-space applications to interact with the kernel proper or modules that implement the system call. Thus I use these available interfaces to execute and analyze the privileged code of firmware kernel modules. Traditionally, the IOCTL has the following prototype:

```
int ioctl(int fd, unsigned int cmd, void *arg),
```

The `fd` argument is related to the device file registered by the module under the `/dev` directory and the `cmd` is a number that acts as an index to a dispatch mechanism (e.g., via a `C-switch` statement) within the IOCTL implementation. Finally `arg` holds a pointer to user data passed to the kernel module. The `void` pointer type offers a measure of flexibility for the module to interpret the data in any specific format it requires (e.g., as a `struct`). I use the `void` pointer to pass the fuzzer mutated data to the kernel module. In the case

of a networking module I use the pointer to pass a `struct ifreq` to the module (see Listing 4.4). This `struct` consists of two `unions` of 16 bytes each. The fist `union` stores the name of the network interface associated with the module, used by the kernel to find and execute the IOCTL of the target kernel module and the second `union` stores configuration data about the network interface. Since the network interface name must be correctly placed in the first 16 bytes of `struct ifreq`, for the kernel to use and find the target kernel module, I only have the remaining 16 bytes available in the `struct` to replace with mutated data from the fuzzer.

Next to detect the `cmd` numbers used for the dispatch mechanism, `FirmSolo` uses the Ghidra (NSA Research Directorate, 2025) software reverse engineering tool. While several previous works on kernel driver IOCTL fuzzing and IOCTL `cmd` excavation (Corina et al., 2017; Pustogarov et al., 2020; Lu et al., 2020) exist, these tools either require the source code of the modules and the kernel or they are not suited for large scale analysis, thus they are not applicable to our analysis pipeline. Instead, I use Ghidra's Python scripting to lift the assembly of the $\mathbb{D}_{KM}$ kernel modules (ARM and MIPS) into Ghidra's intermediate representation (IR) PcodeOp, to extract integers used in conditional branch instructions or as jump table indexes, both of which are an indicator of the dispatch mechanism within the IOCTL's implementation. However, this approach introduces a large number of integers unrelated to the target module's IOCTL dispatch. To filter out most of the unwanted cases I use AFL's `afl-cmin` tool which keeps the minimum number of seeds that produce distinct instrumentation results during a fuzzing dry run.

Firmadyne. Firmadyne relies on a single modified kernel version and heavily relies on the Kprobes(Keniston, 2021) function hooking mechanism to collect runtime data on various system calls. However, Kprobes was introduced into the Linux kernel after version `2.6.36-rc1` and thus it is not supported by older versions used by IoT devices present in my data set. Thus, to support Firmadyne on older and newer kernels, `FirmSolo` obviates

the need for Kprobes and automatically patches the implementation of the system calls within the kernel's source directly resulting in identical hooking functionality.

As Firmadyne relies on the original startup scripts contained in each image, these scripts control which modules actually get loaded. However, `FirmSolo` must ensure that Firmadyne does not attempt to load any of the crashing modules from $\mathbb{C}_{KM}$. To this end, `FirmSolo` simply replaces the $\mathbb{C}_{KM}$ modules with their $\mathbb{U}_{KM}$ counterparts, or deletes the respective module in the Firmadyne file-system image if no replacement module exists. Thus, instead of loading a crashing module, the startup scripts in Firmadyne will transparently load either a replacement module or no module at all.

4.4 Evaluation

In this section I evaluate `FirmSolo`'s three stage approach and especially the K.C.R.E. process on a large set of firmware images and the kernel modules they contain. Specifically, I evaluate `FirmSolo` along three dimensions.

Q1 *Efficacy:* What is the contribution of the first two stages of `FirmSolo` to successfully loading IoT kernel modules (§ 4.4.2)?

Q2 *Precision:* How close is the kernel configuration that `FirmSolo` infers to a ground-truth kernel configuration (§ 4.4.3)?

Q3 *Broader Applicability:* How does `FirmSolo` aid downstream systems to analyze kernel modules (§ 4.4.4)?

In the following, I first introduce the dataset that I use to evaluate `FirmSolo`. I then present the experiments that I conduct to evaluate my approach in detail.

4.4.1 Evaluation Dataset

My dataset consists of 8,737 images used by Firmadyne and shared with us by its authors. Additionally I downloaded 50 (at the time of writing) up-to-date images belonging to 7 vendors, to demonstrate `FirmSolo`'s applicability to modern firmware as well. Of these 50 downloaded images, I were able to use 15 for the evaluation, as for 33 images I were not able to extract the file-system (e.g., due to encrypted images) and 2 images target the `AArch64` architecture, currently not supported by my prototype implementation. Only 5 out of these 15 images use kernels of the `4.x` series or above, illustrating that IoT vendors continue to rely on outdated software in their devices, and thus validating the continued relevance of the Firmadyne dataset despite its age. I also refined the Firmadyne dataset and removed 4,020 images belonging to open-source projects such as OpenWRT (Openwrt Project, 2021) and DD-WRT (Project, 2021) as the repetitive analysis of essentially thousands of duplicates would distort the results. Furthermore, I removed 254 images targeting architectures not supported by `FirmSolo`, 711 images that use kernels prior to `2.6.18` which require ancient toolchains to compile, and finally 2,282 images whose file-system did not contain any kernel modules. Eventually, my evaluation dataset consists of 1,470 firmware images, where 969 target MIPS and 501 target ARM platforms. The firmware images in my dataset span across 95 unique kernel versions, between version `2.6.18` and version `4.4.198`. The compiler tool chains I use to compile the K_{FS} kernels in my experiments are `gcc-3.4` (to support kernels $\leq$ `2.6.23`), `gcc-4.3` (for kernels $\geq$ `2.6.23`) and `gcc-5.5` (for kernels $\geq$ `4.4.x`) for both MIPS and ARM.

Finally, Table 4.3 summarizes the different categories of the kernel modules that comprise my dataset. I categorize the modules based on their pathname in the firmware image file-system. The majority of the modules belong to the networking category (46% of the total modules), followed by driver modules (26% of the total modules). I consider miscellaneous all the modules whose category is unknown.

Type of Data	#
Symbols (Unique - $\mathbb{D}_{KM}$ ELF headers only)	9,028
Symbols (Unique - KALLSYMS included)	95,646
C-source files exporting required symbols (Unique)	7,159
Configuration options (Unique)	1,164

Table 4.1: Symbol, C-source file and option information collected by FirmSolo.

4.4.2 Efficacy

In this section I evaluate the individual stages of FirmSolo and answer Q1.

1) Information Gathering: While processing the 1,470 images, FirmSolo extracted the names of 95,646 distinct undefined symbols averaging 8,557 undefined symbols per image. As explained in Section 4.3.1, these originate from the ELF headers of the kernel modules and the KALLSYMS entry of K_0 (if available). In my dataset, 948 images (65%) contained a K_0 kernel, 871 of which (92%) contain the KALLSYMS information. This highlights that for the remaining 599 images, identifying symbols via ELF headers is essential. On average, FirmSolo recognizes 14,123 undefined symbols for images whose K_0 has KALLSYMS defined, and an average of 463 if KALLSYSMS is not available. In the absence of KALLSYMS, FirmSolo only concerns itself with symbols imported by the kernel modules. As most symbols in KALLSYMS pertain to kernel functionality not used by modules, these symbols' corresponding configuration options are irrelevant to FirmSolo. Table 4.1 summarizes the symbol information obtained through the Information Gathering stage ①.

2) K.C.R.E. I evaluate K.C.R.E. along three dimensions: 1) its capability to successfully map the symbol information collected in stage ①, to C-source files and the corresponding options in the target kernel tree, 2) the capability of the K_{FS} kernels to successfully load the $\mathbb{D}_{KM}$ modules of the firmware images, and 3) the ability of the Data Structure Layout Correction step to address the errors of the $\mathbb{C}_{KM}$ modules and reintroduce these modules in the analysis pipeline.

Experiment	$\mathbb{D}_{KM}$ Load.
Base `.config` only	19
Base `.config` + vermagic	5,276
K.C.R.E. - (2c)	35,354
K.C.R.E.	36,178

Table 4.2: `FirmSolo` experiments with different settings.

Categories	Total	Loaded
net	25,820	20,422
drivers	14,608	6,942
misc	8,792	3,968
fs	3,965	2,784
lib	1,807	1,326
crypto	972	324
arch	724	412
Total	**56,688**	**36,178**

Table 4.3: Module dataset categorization.

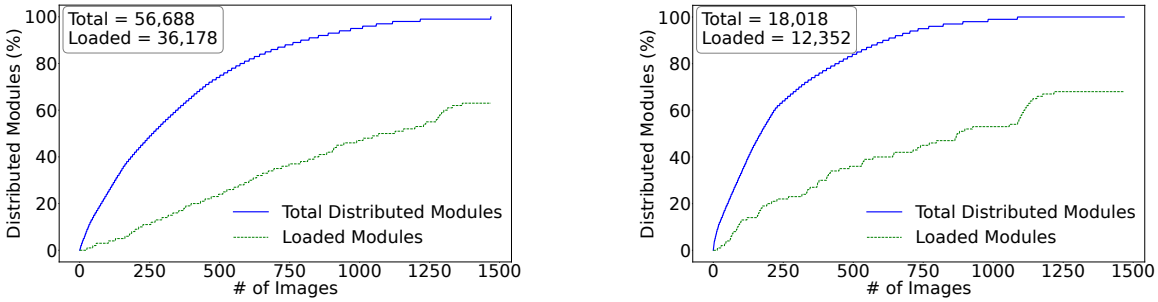
Symbols to Configuration Options. On average, `FirmSolo` maps the 8,557 undefined symbols of each image F_{image} to 391 C-source files per image, and in turn to 123 configuration options. `FirmSolo` then attempts to enable these options, and succeeds for 111 (92%) options on average, during the construction of the `.config` file. Table 4.1 contains information about the number of distinct C-source files and unique configuration options `FirmSolo` detected throughout this stage.

Of course, not all symbols identified in stage (1) can be mapped to C-source files in the target kernel sources and then to configuration options. For example, any symbol that appears in the kernel due to a vendor's modifications to the kernel will not be present in the upstream kernel repository and hence is not identified by `FirmSolo`. On average `FirmSolo` fails to map 2,657 (31%) symbols per image to C-source files in the target kernel tree. This phenomenon is mostly prevalent in the case of images that have a `KALLSYMS` entry, where `FirmSolo` tries to resolve all the symbols exported by K_0, including those not relevant to any modules. In fact, `FirmSolo` fails to map 4,441 (32%) symbols on average in the case of images with a `KALLSYMS` entry and 63 (14%) on average when the entry is not available.

Module Loading. `FirmSolo` is the first system that automatically builds kernel images that can load, and subsequently subject to dynamic analysis the $\mathbb{D}_{KM}$ kernel modules. To support this claim I first tested Qiling (Qiling, 2022), a binary emulation framework with

kernel module support, against a random sample of ten kernel modules from my dataset. Unfortunately, Qiling aborted with errors for both MIPS and ARM modules in all cases. Specifically, Qiling does not support symbol relocation types, such as `R_MIPS_26` and `R_ARM_CALL`, commonly used by IoT kernel modules. I contacted the Qiling developers and the reply indicated that MIPS and ARM are not well tested. In light of these events I consider improving Qiling outside the scope of this work.

Instead, to provide a complete evaluation of `FirmSolo` and better showcase the contribution of K.C.R.E. I test my system under four scenarios with different settings. In detail, I run these experiments using: 1) only the template `.config`, 2) the template `.config` plus the vermagic setting, 3) K.C.R.E. without step (2c) and 4) K.C.R.E. These experiments clearly highlight the improvement introduced by K.C.R.E. in the kernel module loading over the base cases (i.e., Experiments 1 and 2). The results in Table 4.2 clearly show that with K.C.R.E. my system is substantially more successful as it loads 85% more modules than Experiment 2.



(a) `FirmSolo` firmware emulation step results

(b) Firmadyne firmware emulation step results

Figure 4·3: Cumulative Distribution Functions representing the modules in $\mathbb{D}_{KM}$ loaded by `FirmSolo` and Firmadyne, respectively.

Specifically, `FirmSolo` loads 36,178 (64%) of the 56,688 $\mathbb{D}_{KM}$ modules in my dataset, while implementing the substitution mechanism. Figure 4·3a shows the cumulative distribution function over the number of successfully loaded $\mathbb{D}_{KM}$ modules when applying `FirmSolo` on my dataset. As illustrated in Table 4.3, these modules fall into various cate-

Module Failure	**# of Modules**
Unknown symbol	13,039
Image not emulated	3,265
Relocation overflow	1,917
Modules crashed	629
Duplicate symbol export	337
File exists	258
Invalid module format	230
Module timeout	198
No such device	170
No vermagic match	123
Operation not permitted	109
Invalid argument	97
Device or resource busy	88
Cannot allocate memory	27
Resource temporarily unavailable	9
Module has no symbols	5
Bad address	5
No buffer space available	3
Relocation out of range	1
Total	**20,510**

Table 4.4: Module load failure reasons during the `FirmSolo` experiments. The special cases where firmware images were not emulated, modules timed out during loading (after 60 seconds) and modules crashed are also included in the table with *italics*.

gories, the majority of which belong in the networking category (57% of the loaded modules). Therefore `FirmSolo` is not only a scalable system but also diverse as it supports kernel modules from multiple categories.

Of course, I do not count as success the 4,601 module substitutions that `FirmSolo` implements in the course of the experiments, in my results. The substitutions account for 8% of the total $\mathbb{D}_{KM}$ modules in my set and are related to both $\mathbb{D}_{KM}$ modules that crash during their initialization process (311 substitutions) or fail to pass all steps of the module loading process explained in Section 2.1.2 (4,290 substitutions). Crucially, the substitution system enabled the loading of 5,290 (9%) additional $\mathbb{D}_{KM}$ modules in my set which would

otherwise not load and not be available to downstream analysis.

Failed Cases. The remaining 20,510 modules did not load in my experiments, due to various reasons. My analysis showed that the majority of these modules (64 %) fail to pass the symbol resolution step (see Section 2.1.2), due to required symbols not exported by the kernel proper (see Section 4.4.2). I present these failure reasons in Table 4.4 . While most of these required symbols are entirely missing from the upstream kernel source trees (e.g., `osl_malloc`), there are cases where `FirmSolo` fails to add symbols into the kernel proper even when they are present in its kernel tree.

A notable example is the symbol `__pci_register_driver`, tied to the configuration option `CONFIG_PCI` and required by 1,037 modules that were not loaded. Unfortunately, `FirmSolo` is unable to set `CONFIG_PCI` for ARM images compiled for the `Realview` platform, since the latter does not support a PCI bus. Thus, any module that requires PCI functionality will not load in a `Realview` platform-based kernel. Another interesting failure category is the `Relocation overflow` error. This error is related to the availability of a limited bit range (i.e., 28 bits) for addressing when the kernel and kernel modules are placed far apart in memory. It only affects MIPS images in my experiments and based on my manual analysis it is specific to modules compiled from an open source distribution (e.g., OpenWRT), that is loaded in my upstream kernels. Since I removed all the OpenWRT images from my dataset, I conclude that vendors borrow code from open source projects like OpenWRT and include it in their distributed firmware. Finally, `FirmSolo` might fail to load a module because the module code is already compiled as part of the kernel proper. This is a limitation of my system related to K.C.R.E. not being able to set options to value `m` because of dependency constraints. For example, errors in categories `Duplicate symbol export` and `File exists` are related to this limitation (see Table 4.4).

Emulation Failures. While `FirmSolo` compiles its kernels to be supported by QEMU, there are cases where QEMU fails to emulate the kernels. Specifically, `FirmSolo` fails to

emulate 48 images in my set with a total of 3,265 (6%) modules. I do not consider these cases as a limitation of my system, since 29 images require a kernel with `MIPS Thread Context` (a feature for parallelism) which is not supported by QEMU and in the rest of the cases either the kernel is not uncompressed or it cannot mount my generic file-system. I also include these cases in Table 4.4.

Finally, loading all modules in a file-system comprises the worst-case scenario for `FirmSolo`, as few real-world systems will show this behavior. Section 4.4.4 illustrates that real IoT firmware only attempts to load a subset of the modules contained in their images.

Data Structure Layout Correction. Out of the 56,688 $\mathbb{D}_{KM}$ modules in my set, 1,080 (2%) crash ($\mathbb{C}_{KM}$ modules) while being loaded in step (2b). By invoking step (2c), `FirmSolo` "fixes" 451 (42%) of the $\mathbb{C}_{KM}$ modules that crashed, due to a misaligned data structure member access. Delving deeper into the results revealed that `FirmSolo` was able to successfully recover the layout of three core data structures, `structs net` in 272 modules, `kmem_cache` in 171 modules using my `kmem_cache_alloc` heuristic and `module` as my first special case in 118 modules (55 cases together with `struct net` and 55 cases together with `struct kmem_cache`).

Unfortunately, `FirmSolo` is unable to fix the remaining 629 $\mathbb{C}_{KM}$ modules. My manual analysis upon 232 out of these $\mathbb{C}_{KM}$ modules, revealed that the majority of their crashes are not related to a misaligned data structure. Instead, in 110 cases the modules execute kernel code that crashes when calling the `BUG_ON` macro. This macro performs a check and if it fails, the macro *traps* causing an `Oops`. After inspecting the kernel code and the assembly of the $\mathbb{C}_{KM}$ modules I found out that the vendors have modified their kernels' source code regarding the check, thus the check fails on my upstream kernels. In addition, in 95 cases the modules make a memory access related to a hardware peripheral that is not supported by QEMU and thus an *unhandled memory access* occurs causing the crash.

Since peripheral emulation is out of scope for this work I do not consider these crashes a limitation of `FirmSolo`. I suspect that more $\mathbb{C}_{KM}$ modules that `FirmSolo` cannot fix, crash due to one of the reasons mentioned above. Finally, for the remaining 27 modules `FirmSolo` is actually able to recover the misaligned data structures but not recover their correct layout (`structs bonding` (3), `device` (2), `net` (17), `net_device` (1) and `module` (4)). My manual analysis revealed that in 5 cases `FirmSolo` incorrectly recovered the accessed members (i.e, offsets) of the offending data structures, causing the layout recovery algorithm to fail. For the remaining cases, dependency constraints prevented the layout recovery algorithm from successfully testing all the possible solutions, since `FirmSolo` could not set options in these solutions to their required value (either `y` or `n`).

4.4.3 Precision

To assess the precision of the K.C.R.E. process when inferring the kernel configuration, I compare its results against a ground-truth firmware image built for the open source Backfire OpenWRT release (OpenWrt Project, 2021).[4] The open source nature of OpenWRT means I have access to complete ground-truth of the configuration used to build the kernel and its modules, allowing us to answer Q2. This experiment aims to quantify K.C.R.E.'s reverse engineering accuracy.

Specifically, I used OpenWRT's image builder to create an image from the heavily modified (compared to the upstream Linux kernel) OpenWRT source,[5] for a MIPS-based Broadcom device. The resulting image targets the `2.6.32.27` kernel and contains 530 kernel modules.

I measure `FirmSolo`'s precision along two dimensions and perform each measurement without the use of the optional `KALLSYMS`. First, on a macro-scale I quantify how many of the 530 modules can be loaded by the K_{FS} kernel produced by `FirmSolo`. The K_{FS}

[4] OpenWRT is an after market Linux-based software distribution supporting dozens of system-on-chip platforms and hundreds of IoT devices.

[5] `git://git.openwrt.org/openwrt/svn-archive/archive.git`

kernel successfully loads 484 (91%) of the 530 modules. In total there are 8 cases where modules in $\mathbb{D}_{KM}$ crash during loading. Unfortunately, step (2c) is unsuccessful in addressing the module errors. Specifically, while `FirmSolo` detects the offending `struct` in 2 cases, no solution is applicable since the offending data structures (`structs sk_buff and net_device`) are modified at their source code. For the remaining 6 cases Ghidra is not able to correctly recover the variable `def-use` chains, from the crashing functions, thus causing the layout recovery process to abort. In addition, 9 substitutions were implemented by the substitution mechanism in step (2b), 7 of which replace crashed modules in $\mathbb{C}_{KM}$. Finally, 19 modules fail to load because the code of their $\mathbb{U}_{KM}$ counterparts is already compiled as part of the kernel proper. Dependency constraints prevent `FirmSolo` from setting the corresponding options to value `m`, during K.C.R.E. (see Section 4.3.2). The remaining modules did not load into the K_{FS} kernel, since symbols are missing from the upstream kernel, due to the modifications in the OpenWRT's kernel source code.

Second, on a micro-scale I assess whether `FirmSolo` sets the *relevant* configuration options correctly. In this context, I consider a configuration option relevant only if it affects the layout of a data structure defined in the kernel and used by a module. In total the $\mathbb{D}_{KM}$ modules access 4,716 data structures defined in the OpenWRT kernel (4,675 of these also exist in the upstream kernel). However, the vast majority thereof have fixed layouts that are not affected by any configuration options. Thus, I am only interested in the subset of 335 data structures in the upstream kernel whose layout depends on the kernel configuration. The layout of these data structures is affected by 240 configuration options, and the ground-truth `.config` used by the OpenWRT image builder has 75 of these options set. `FirmSolo`, correctly infers and selects 48 of these options. The data structure members guarded by the remaining options are either not accessed during module loading or accessing them does not lead to a crash during the emulation in step (2b).

4.4.4 Broader Applicability

To demonstrate the utility of `FirmSolo`, this section discusses my experience with two representative downstream analysis systems enhanced by `FirmSolo`: the TriforceAFL kernel fuzzer (NCC Group Plc, 2017) and the Firmadyne dynamic firmware analysis system (Chen et al., 2016). Note that `FirmSolo`'s novel capability is widely applicable to a variety of other analysis systems beyond these examples (e.g., PANDA(Dolan-Gavitt et al., 2015) or FirmAE(Kim et al., 2020)).

1) TriforceAFL. In this section I evaluate the TriforceAFL kernel fuzzer's ability to discover previously unknown bugs in IoT kernel modules. Specifically, I use TriforceAFL on 75 $\mathbb{D}_{KM}$ open and closed-source modules from my set, which either expose a character device interface or a network interface, that I can open and write to and contain at least one IOCTL interface.

I fuzz my modules for 24 hours on an Intel Xeon machine with minimum of 16GB of RAM. The fuzzer triggered 19 memory corruption bugs in 11 proprietary modules, 7 in MIPS, and 4 in ARM images (see Table 4.5). In contrast to the open-source kernel modules, which are regularly maintained, closed-source modules probably do not receive similar code review or testing, and thus contain various bugs.

These bugs belong to different categories such as reads and writes to arbitrary locations in memory (10), NULL pointer dereferences (2), out-of-bounds memory accesses (4), slab (heap) corruption (2) and large virtual memory allocation (1). I manually verify each bug by detecting the source of each crash in Ghidra. I further verify the crash cases for the `acos_nat.ko` and `ipv6_spi.ko` modules on a physical Netgear WNDR3400v2 device, to confirm that TriforceAFL's findings are true positives. I test all 5 bugs related to these two modules and confirm the existence of the bugs, either by crashing the modules or causing a kernel panic, which forces the device to reboot.

The fuzzer also produced a total of up to eleven false positives. I could confirm that

Module	Paths	Vendor	Kernel	Bugs(FP)
MIPS				
acos_nat.ko	421	Netgear	2.6.22	3
art.ko	110	DLink	2.6.31	1
art-wasp.ko	56	ZyXEL	2.6.31	1
edinvram2.ko	98	ZyXEL	2.6.36	1(1)
gpio.ko	53	DLink	2.6.31	1(2*)
i2c_drv.ko	41	Linksys	2.6.36	0(1)
ipv6_spi.ko	32	Netgear	2.6.22	2
ppp_generic.ko	75	TRENDnet	2.6.31	0(1)
ralink_i2s.ko	49	Linksys	2.6.36	0(1)
rt_rdm.ko	54	TP-Link	2.6.36	1
tun.ko	51	Belkin	2.6.31	0(1)
ARM				
gpio.ko	140	Supermicro	2.6.24	1(1*)
IDP.ko	68	Asus	2.6.36.4	3(1)
ppp_generic.ko	389	Synology	2.6.32.12	0(1)
smcdrv.ko	35	Supermicro	2.6.24	1
u_filter.ko	184	Tenda	2.6.36.4	4
orion_wdt.ko	91	Linksys	2.6.35.8	0(1)
			Total(FP)	19(11)

Table 4.5: Fuzzer statistics and results for the vulnerable modules in our set. The * indicates the False Positive might be an actual bug but requires hardware access to confirm.

the crashes in the open source `tun.ko` and `ppp_generic.ko` modules are false positives. A manual analysis with Ghidra quickly revealed that `FirmSolo` did not correctly align `structs net` and `file` which lead to a kernel crash during the modules' analysis by the fuzzer. Furthermore, the fuzzer produced eight likely false positive cases for modules without a $\mathbb{U}_{KM}$ counterpart. My manual analysis via Ghidra revealed that all of these cases are related to accesses in memory regions that might be mapped as MMIO to a physical device not modeled by QEMU. For the three crashes in the `gpio` modules in Table 5.6 I were unable to confirm whether the accessed memory locations are MMIO mapped, and whether these cases are true false positives. Due to the lack of access to a corresponding physical device I conservatively label all these crashes as false positives. Note that I do not con-

sider MMIO related errors as a limitation of `FirmSolo` since providing models for the peripherals accessed by the $\mathbb{D}_{KM}$ modules is out of scope for this work.

Of course, I responsibly disclosed all previously unknown bugs I detected to the affected vendors. During the disclosure 7 vendors, (all except Tenda) corresponded directly with me requesting a PoC and, so far, 2 (Supermicro and Asus) confirmed my findings.

2) Firmadyne. To assess the utility of `FirmSolo` for Firmadyne, I run the 1,470 images from my evaluation data set in Firmadyne. Recall that in the evaluation in Section 4.4.2 `FirmSolo` attempted to load all the modules in a firmware image. When ran in Firmadyne, the subset of modules that will be loaded is determined by the image's startup scripts. Specifically, when running in Firmadyne, the 1,470 images attempt to load 18,018 kernel modules at boot time (i.e., 32 % of all 56,688 modules in my dataset). While the original Firmadyne system cannot load a single kernel module, the improvements made through `FirmSolo` allow the system to successfully load 12,352 (69%) of these modules. Figure 4·3b shows the cumulative distribution function over the number of $\mathbb{D}_{KM}$ modules loaded in the Firmadyne experiments.

I also encountered 141 additional cases during my experiments where modules in $\mathbb{D}_{KM}$ crashed during their emulation. Step (2c), which is not automated for Firmadyne, addresses 57 of these cases, where the offending data structure is `struct sk_buff`. For the rest of the modules not addressed by step (2c), my manual analysis revealed that their majority (43) originate from kernel functions outside the offending modules and their dependencies. Currently, `FirmSolo` is unable to address these cases.

A particularly useful capability of Firmadyne is that it allows an analyst to quickly confirm the presence of known and previously-unknown vulnerabilities in a large set of firmware images. I leverage this capability and launch 10 publicly available proof of concept exploits from ExploitDB (Offensive Security, 2021), targeting the modules contained in my dataset (full list in Table 4.6). Additionally, I use Firmadyne to assess which

CVE #	Description	Confirmed
CVE-2015-3036	Stack based buffer overflow	Y
CVE-2009-1897	NULL pointer dereference	N
CVE-2016-3135	Integer overflow	N
CVE-2017-14489	Memory Corruption	N
CVE-2013-1828	Memory Corruption	N
CVE-2016-3134	Memory corruption	N
CVE-2016-4997	Memory corruption	N
CVE-2014-4943	Memory corruption	N
CVE-2007-2878	Kernel object corruption	N
CVE-2017-16939	Use-after-free	N

Table 4.6: The vulnerabilities attempted to confirm within Firmadyne and proof-of-concept exploits from ExploitDB. Seven unconfirmed vulnerabilities rely on configuration options not used by the firmware images in my dataset

firmware images are prone to the 19 IOCTL related bugs found by TriforceAFL as discussed above. Of the 10 exploits from ExploitDB only one (`CVE-2015-3036`) successfully triggers the corresponding bug. This bug exists in five different versions of the closed-source `NetUSB.ko` across 15 firmware images. Seven of the remaining PoCs only work against modules built with configuration options not commonly found on IoT firmware, such as user namespaces (i.e., `CONFIG_USER_NS`). Given the firmware images in my dataset did not use this functionality, the vulnerable code is not included in the $\mathbb{D}_{KM}$ modules. `CVE-2009-1897` and `CVE-2014-4943` target older module versions than the ones present in my module set, thus were also unsuccessful. From the 19 IOCTL bugs 6 (gpio-mips, gpio-arm, acos_nat (2) and ipv6_spi (2)) successfully crashed their corresponding modules in 84 images. The other bugs were unsuccessful since the corresponding modules re not loaded by the startup scripts in their firmware images.

In summary, my experiments with two representative downstream analysis systems illustrate how `FirmSolo`'s novel capability to build IoT firmware-image specific kernel binaries enables the analysis of proprietary binary-only kernel modules for bugs and vulner-

abilities.

4.5 Limitations

As any automated solution, `FirmSolo` is subject to a set of limitations. First, `FirmSolo` currently supports kernels newer than `2.6.18` and works for the MIPS, ARMv[5-7] architectures. Supporting earlier kernel versions can be achieved by working with older cross-compiler toolchains and additional architectures can be added with minor engineering effort. Second, the K.C.R.E. process can introduce false positives in the downstream analysis if the K_{FS} kernel and $\mathbb{U}_{KM}$ modules it produces do not agree with the $\mathbb{D}_{KM}$ modules on the memory layout of data structures these modules use. Third, the layout recovery process of step (2c) does not address cases where the errors in $\mathbb{C}_{KM}$ modules and their dependencies originate outside their crashing functions. Introducing a dynamic analysis approach in step (2c) could potentially address these cases. I leave this as future work. Finally, `FirmSolo` cannot analyze kernel modules that require the presence of peripherals (e.g., a USB device) not supported by QEMU.

Chapter 5

Pandawan: Quantifying Progress in Linux-based Firmware Rehosting

In this chapter, I detail my second contribution to firmware re-hosting. First, I present my novel comparison technique for full-system re-hosting frameworks and the high level overview of my holistic re-hosting and analysis approach in Section 5.1. Next, I discuss in detail `Pandawan`, the prototype implementation of both my comparison method and holistic re-hosting and analysis technique in Section 5.2. Finally, I provide information about the evaluation of both the `FICD` technique on state-of-the-art full-system re-hosting frameworks, as well as the holistic re-hosting and analysis capabilities of `Pandawan`.

5.1 Overview

In this section, I detail the `FICD` technique and the system overview of `Pandawan`. The goal of `FICD` is to detect the (reference) point in emulation where the firmware's initialization is complete. Since this point is conceptually common across re-hosting systems, it is possible to compare these systems along various metrics (e.g., user and kernel code coverage).

In Section 5.2.1, I discuss in detail how to apply `FICD` on different full-system re-hosting approaches and compare them based on their emulation capabilities.

To fill the void in holistic firmware re-hosting I present `Pandawan`, a framework capable of holistic (both user and privileged) re-hosting and analysis of IoT firmware code. As illustrated in Figure 5·1, `Pandawan` constitutes a two-component framework: ① The `Kernel Augmentation` (`KA`) component supplements FirmSolo's kernels (FS_k) with ker-

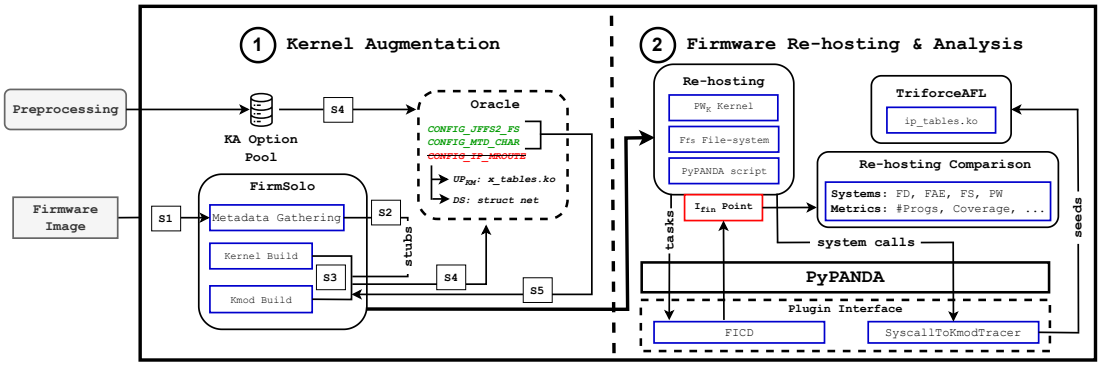


Figure 5·1: System overview of `Pandawan`: The figure features the two components of `Pandawan`. The first component, `Kernel Augmentation` is a 5-step process that produces `Pandawan`'s PW_k kernels. The second component, `Firmware Re-hosting & Analysis` uses a PyPANDA script to emulate the PW_k kernel and the F_{fs} file-system under PyPANDA. It also uses PyPANDA's plugins to holistically analyze the target firmware.

nel functionality frequently included by vendors in their IoT firmware kernels (IoT_k) and used by user level firmware code. I refer to this functionality as `user-code-required` functionality. The outcome of component ① are `Pandawan`'s "augmented" kernels (PW_k), which support both user level and kernel module code (i.e., holistic) re-hosting. ② The `Firmware Re-hosting & Analysis` component is responsible for the holistic re-hosting of the firmware code while emulating the PW_k kernels under PyPANDA. This component also employs PyPANDA's plugins to conduct a series of OS introspection and holistic analyses (e.g., tracing system calls that lead to kernel module code execution) during the firmware emulation run. Then, the results of the analyses can be transformed into seeds to fuzz the firmware kernel modules with a kernel fuzzer (e.g., TriforceAFL).

Throughout the rest of the chapter I use the notations *user* code to refer to user level firmware code and *kernel* code to refer to privileged/kernel level code. The privileged level component can be further classified into the core kernel and kernel module code. I also reference the terms *program*, *process*, and *task*. Intuitively, a program corresponds to an executable in the firmware's file-system. I define a process as the running instantiation of a program during emulation. Finally, I refer to a task as the concatenized `argv` vector,

representing (as a string) a program (name) along with its arguments.

5.1.1 Firmware Initialization Completion Detection

As mentioned previously, quantifying the progress of full-system re-hosting approaches is challenging, due to their unique implementation features that impact the emulation of a firmware image (F_{image}) for each re-hosting framework. For example, the combination of both *user* and *kernel* code being re-hosted may result in different emulation speeds and thus in a different amount of overall code executed at the same point in time compared to only *user* code emulation. Thus, simply using ad hoc comparative methods such as the wall-clock time as the common denominator to assess the progress of re-hosting systems does not produce objective results. Instead, `FICD` aims to detect the point (I_{fin}) during the emulation of F_{image} where its initialization phase has finished. At I_{fin}, full-system re-hosting approaches can be compared using different metrics, such as the number of executed user level programs, *user* and *kernel* code coverage, or number of kernel modules loaded, which I call `emulation-based` metrics. In Section 5.3.5, I use these `emulation-based` metrics to compare the emulation progress of `Pandawan`, Firmadyne, FirmAE and FirmSolo. However, these metrics can of course be supplemented with additional ones depending on the analysis' requirements (see Section 5.4). I note that I_{fin} is not fixed (in time) for any two re-hosting systems, especially if their emulation speed varies.

The rationale behind `FICD` is that all the firmware activity up to the I_{fin} point is automated via the configuration and startup scripts within F_{image}'s file-system. Beyond I_{fin}, there is none to little automated activity occurring, since IoT firmware primarily invokes new functionality (i.e., tasks), such as setting new firewall rules or recording a video, based on its interactions with external actors (i.e., human interaction (Junnila, 2023)). I consider detecting events that lead to new activity after I_{fin} as a problem orthogonal to this research and I leave it as future work.

Since the activity after the firmware initialization phase is limited, I_{fin} can be consid-

ered as any point during the emulation of F_{image} after it has stopped executing new tasks. Specifically, FICD defines a grace period (or *time frame*) tf during which the emulation of F_{image} can stay alive while F_{image} does not execute any new tasks (to see how I choose tf refer to Section 5.3.5). If during the emulation of F_{image}, time greater than tf elapses without executing a new task, FICD marks that point as I_{fin}. Here, metrics can be collected and compared. Even though I_{fin} might not be fixed (timewise) between different full-system re-hosting approaches that emulate F_{image}, it will always represent F_{image}'s end of initialization phase (regardless of the re-hosting framework).

5.1.2 Holistic Re-hosting & Analysis

```
[ncc_runtimecfg.c:  539
        initRunTimeCfg()] :::  Total 4470
        nodes, each node 24 bytes.
[ncc_runtimecfg.c:  540
        initRunTimeCfg()] :::  Total
        107280 byte for all nodes.
[ncc_runtimecfg.c:  207 loadCfg()] :::
        Start loadCfg
[ncc_lz77.c:  509 flash2rootfs()] :::
        BUG ON!!
[ncc_lz77.c:  538 flash2rootfs()] :::
        Load Fail(/var/tmp/cfg.txt)
[ncc_runtimecfg.c:  213 loadCfg()] :::
        flash2rootfs() fail!!  restore to
        default!!
[ncc_lz77.c:  621 rootfs2flash()] :::
        BUG ON!!
[ncc_runtimecfg.c:  215 loadCfg()] :::
        Del TAG default file!!
[ncc_lz77.c:  509 flash2rootfs()] :::
        BUG ON!!
...
```

(a) Example firmware image serial log when re-hosting with FirmSolo.

```
undefined4 loadCfg(void)
{
  bool bVar1;
  char *__stream;
...
  while( true ) {
    (*pcVar7)(__stream);
LAB_004a73cc:
    iVar3 = \
    flash2rootfs("/tmp/cfg.txt");
    if (iVar3 != 0) break;
    bVar1 = true;
    pFVar4 = \
     fopen64("/dev/console","a");
    if (pFVar4 != (FILE *)0x0) {
      uVar2 = getpid();
...
      fclose(pFVar4);
    }
...
  }
```

(b) Ghidra snippet of ncc's loadCfg function

```
undefined4 flash2rootfs(undefined4
        param_1)
{
  FILE *pFVar1;
...
  pFVar1 = \
   fopen64("/dev/mtdblock4","rb");
  if (pFVar1 == (FILE *)0x0) {
...
    fputs("BUG ON!!\n",pFVar1);
  }
  else {
...
    if (iVar3 == 0) {
...
    }
    return 1;
  }
  ...
  return 0;
}
```

(c) Ghidra snippet of ncc's flash2rootfs function

Figure 5·2: An example of a firmware re-hosting with FirmSolo. The firmware targets the D-Link DIR-826L Wi-Fi router. The figure provides the serial log output of the firmware under test as well as the Ghidra snippets of the loadCfg and flash2rootfs functions called by the ncc user program during the firmware's initialization.

As discussed previously, there is currently no prior work that supports holistic firmware re-hosting and analysis. At first glance, FirmSolo (refer to Chapter 4 for detailed information) seems to be readily capable of supporting holistic firmware re-hosting and

analysis by emulating both firmware *user* and *kernel* (i.e., kernel module) code. On the contrary though, my investigations reveal that FirmSolo's FS_k kernels lack the important `user-code-required` functionality required by *user* code to execute.

Specifically, the absence of the `user-code-required` functionality in the FS_k kernels might result in firmware *user* code to either prematurely terminate its execution or exhibit unwanted behavior, such as getting stuck in an infinite loop. I provide a motivating example of such a case for `D-Link DIR-826L` Wi-Fi router firmware in Figure 5·2a. The example illustrates the serial console output of F_{image}'s emulation run, where the capability to access a `Memory Technology Device` (MTD) through a character device is missing from the FS_k kernel. As a result, the `ncc` program which handles the configuration of the IoT device and uses character devices to interact with the MTD peripheral, gets stuck in an infinite loop when executed in FirmSolo (lines 7-9 in Figure 5·2a). Specifically, `ncc` executes the `loadCfg` function which in turn calls the `flash2rootfs` function. I provide the Ghidra decompilation snippets for these functions in Figures 5·2b and 5·2c. The `loadCfg` function calls `flash2rootfs` within an infinite loop (lines 6-21 in Figure 5·2b) and exits the loop only when `flash2rootfs` returns a non zero value (line 16 in Figure 5·2b). The return value of `flash2rootfs` depends on the successful opening of the `/dev/mtdblock4` character device (lines 5-17 and 19 in Figure 5·2c). The ability to access a MTD peripheral through a character device is included in the kernel via the `CONFIG_MTD_CHAR` configuration option. As none of the firmware kernel modules use any functionality guarded by `CONFIG_MTD_CHAR`, FirmSolo did not enable this configuration option in its FS_k kernel. In Section 5.1.2, I explain how `Kernel Augmentation` adds `user-code-required` functionality (e.g., `CONFIG_MTD_CHAR`) required by user level firmware programs to execute successfully, into `Pandawan`'s kernels. Next, I detail `Pandawan`'s components.

Preprocessing

Based on my observations (see Section 5.3.3) vendors use similar configuration parameters (options) to configure their IoT firmware kernels (IoT_k). In particular, it is common for privileged firmware code of different IoT devices to share the same capabilities, such as supporting MTD peripherals or networking subsystems (i.e., `netfilter`), which are accessed by *user* code. Thus, by knowing the configuration of the IoT_k kernels of firmware images I also acquire the `user-code-required` functionality required by *user* code to execute. This `user-code-required` functionality is crucial for firmware images without a `KALLSYMS` entry. The `KALLSYMS` entry is a table containing information about the kernel symbols exported by the kernel (and thus about its configuration). The entry is optionally embedded in the kernel binary if the kernel is compiled with the `CONFIG_KALLSYMS` configuration option. I consider the configuration of the IoT_k kernels of firmware images without a `KALLSYMS` entry partially known since the only source about these images' kernel configuration originates from their kernel modules. Thus, to account for the missing kernel configuration (and in turn `user-code-required` functionality) from these firmware images, `Pandawan` collects popular configuration options from the IoT_k kernels of firmware images with a `KALLSYMS` entry. Then, `Pandawan` enables these options in its PW_k kernels for the firmware images without a `KALLSYMS` entry, to provide the `user-code-required` functionality needed by *user* code to execute.

Kernel Augmentation

Component ① is responsible for supplementing the FS_k kernel of F_{image} with the `user-code-required` functionality generally used by *user* code. The product of component ① is the PW_k kernel for F_{image}, which is conducive to holistic firmware code emulation and analysis.

To produce the PW_k kernel for F_{image}, `Pandawan` follows these five steps: S1: Extract

the file-system of F_{image} and gather metadata information, such as the `KALLSYMS` entry from the IoT_k kernel (if both the kernel and entry exist in F_{image}). S2: Furthermore, to aid the loading process of additional kernel modules, supplement PW_k kernels (their code) with "stubs" of symbols that are not present in the upstream kernel source code, but required by F_{image}'s kernel modules. S3: Build the upstream counterparts (UP_{kos}) of open-source kernel modules within F_{image}'s file-system. The UP_{kos} modules are compiled with debugging information available (i.e., `DWARF`). I note here that S3 and S4 will only executed for firmware images that do not have an available `KALLSYMS` entry. I consider the kernel configuration of firmware images with a `KALLSYMS` entry to be "fully" known. Particularly, the entry reveals information about the specific symbols needed by both the *user* and *kernel* code in these images. In these cases, the respective configuration options for these symbols (also used in the IoT_k kernels) will be added (by default) in S5 in the PW_k kernels. S4: Invoke the *Oracle* process with the UP_{kos} modules and either the MIPS or ARM configuration option pool (depending on F_{image}'s architecture). The *Oracle* is a filtering process which checks and removes configuration options from the MIPS or ARM pools that affect the layout of kernel data structures used by the UP_{kos} modules (produced in S2). S5: Finally, feed the "safe" configuration options that remain from S4 to its kernel configuration and build process to produce the PW_k kernels.

Firmware Re-hosting & Analysis

In component ②, `Pandawan` relies on the custom file-system creation and network configuration logic of FirmAE (Kim et al., 2020) to produce a dedicated PyPANDA script for F_{image} (see Section 5.2.2). I note here that the custom file-system also supports `Pandawan`'s (and FirmSolo's) kernel module loading logic. Next, `Pandawan` uses the modified file-system, the PW_k kernel and the PyPANDA script to emulate F_{image} under PyPANDA and conduct a series of analyses on the target firmware using PyPANDA's plugins. Examples of the analyses implemented are the collection of coverage about the *user* code executed dur-

ing the firmware emulation, such as the executed program names and their executed QEMU Translation Block (TB) addresses, and the tracing of system calls (i.e., ID and arguments) invoked by *user* code during the emulation.

Furthermore, through its holistic re-hosting, `Pandawan` captures the interactions between the *user* code and *kernel* code. This introspection provides information about the inner workings of both the user and privileged code of F_{image}, which can then be used by dynamic analysis systems to thoroughly analyze F_{image}. Since prior works only focus on one aspect of the firmware; either the user or the privileged level, they lack insights about how both these firmware aspects actually work together. Thus, these approaches have to resort to heuristics or impose limitations during their firmware analysis. For example, FirmSolo does not have the ability to extract information about system calls that interact with the kernel modules. It is limited only to fuzzing these modules through their `IOCTL` interface.

On the contrary, `Pandawan` does not suffer from these limitations. Instead, it makes use of its OS introspection capabilities to trace the system calls of processes that lead to the execution of code in kernel modules. These traces expose entry points (i.e., through the system calls and their arguments) from user space to kernel space. `Pandawan` relies on these entry points to detect kernel modules in F_{image} that communicate with *user* code through a *socket* or a *file descriptor* such as `iptables.ko` and `gpio.ko`, respectively. Next, `Pandawan` uses this information to create seeds for a modified version of the TriforceAFL kernel fuzzer that supports fuzzing MIPS and ARM IoT kernel modules, to analyze the kernel modules in F_{image}.

5.2 Implementation

In this Section, I discuss the implementation details behind `FICD` and `Pandawan`.

5.2.1 Firmware Initialization Completion Detection

FICD seeks to identify the reference point I_{fin} in the emulation of a firmware image where the firmware completed its initialization phase. To this end, FICD considers that a firmware image reached I_{fin} if no previously unseen (i.e., unique) tasks are launched within tf seconds. I refer to tf as the *time frame* parameter. To assess whether a task is unique, FICD uses the *Levenshtein*[1] edit distance (ed) to check the similarity of a task executed at a given point in time with all the tasks that have been executed prior to that point. I define a task (t) as previously unseen if it has a similarity below a certain threshold h (see Section 5.3.5) with all the previously executed tasks. That is, t is previously unseen iff $\forall_{t_i} ed(t,t_i) < h$ where t_i refers to all tasks started before t. The use of ed rather than strict equality is crucial to mitigate cases where nearly identical tasks (e.g., `/bin/iptables -t filter -F` and `/bin/iptables -t nat -F`) impact the discovery of the I_{fin} point. My implementation of FICD uses a thread that runs alongside the main emulation and evaluates the above expression every five seconds. Once tf expired without a previously unseen task being launched, FICD declares I_{fin} reached and signals that measurements can be taken.

5.2.2 Holistic Re-hosting & Analysis

Next I provide the implementation details behind Pandawan. Since Pandawan uses FirmSolo as its foundation to configure and build its PW_k kernels, Pandawan reuses FirmSolo's reverse engineering and hybrid kernel configuration and build process.

Preprocessing

Before proceeding to the execution of Pandawan's components I first execute the *Preprocessing* step. In this step I pre-compute the user-code-required functionality that needs to be included in the PW_k kernels in component ① to support the successful re-hosting of *user* code. Specifically, I extract the KALLSYMS kernel symbols (where available) from all

[1] https://www.cuelogic.com/blog/the-levenshtein-algorithm

the available IoT_k kernels in my dataset (950 kernels). Then, I group the symbols into two sets S_{MIPS} and S_{ARM} matching the IoT_k kernel architecture respectively. Next, I use FirmSolo's `symbol-to-configuration-option` mapping mechanism to map each symbol s in S_{MIPS} and S_{ARM} to the corresponding configuration option that guards s's implementation in the kernel source code. This yields two pools (sets) of configuration options, one for MIPS (O_{MIPS}) and one for ARM (O_{ARM}). These pools represent the `user-code-required` functionality that is added to the PW_k kernels in component ①.

Kernel Augmentation

In component ①, `Pandawan` consumes a binary firmware image F_{image} as input and invokes the `Kernel Augmentation` (`KA`) technique. The goal of `KA` is to produce the PW_k kernels that are conducive to holistic firmware code emulation and analysis. The `KA` phase consists of five steps:

Step S1 In this step, `Pandawan` executes its metadata gathering process for F_{image}. This process extracts the file-system of F_{image}, metadata information such as IoT_k's `KALLSYMS` symbol entry (if both IoT_k and the entry are available) and also maps these symbols to their corresponding configuration options in the kernel source code.

Step S2 To improve its kernel module re-hosting capabilities, `Pandawan` implements the technique used by EASIER (Pustogarov et al., 2020) for loading out-of-tree kernel modules. Specifically, `Pandawan` supplements the PW_k kernels with "stubs" of kernel symbols (not present in the upstream kernel source code) required by F_{image}'s kernel modules. `Pandawan` detects these symbols in step S1. The symbol stubs simply return a `NULL` value and make the corresponding symbol available to the kernel modules through the kernel's `EXPORT_SYMBOL` macro. Without these symbol stubs the kernel modules would immediately fail to load during the kernel module loading process. Next, if F_{image} contains a `KALLSYMS` entry, `Pandawan` immediately skips to step S5.

Step S3 `Pandawan` executes its kernel build process to build and acquire the upstream

versions (UP_{kos}) of open-source kernel modules within F_{image} (since `Pandawan`, like FirmSolo, uses the upstream version of the IoT_k kernel). The UP_{kos} kernel modules are compiled with debugging information enabled (DWARF), required in step S4.

Step S4 In this step, `Pandawan` invokes the *Oracle* process whose goal is to provide the necessary `user-code-required` functionality to each PW_k kernel. First though, *Oracle* makes sure to filter out any option $o \in O_{MIPS}$ (or O_{ARM}) that affects the layout of kernel data structures used by the kernel modules within F_{image}. If the PW_k kernel and F_{image}'s kernel modules do not agree about the layout of their common data structures, then misaligned data structure member accesses and in turn kernel module crashes might occur during emulation.

To this end, *Oracle* uses the DWARF information from the UP_{kos} kernel modules compiled in S3 and extracts the data structures used by F_{image}'s kernel modules. The intuition behind this process is that both the UP_{kos} modules and their open-source counterparts within F_{image} (if they are unmodified) make use of the same data structures since they share the same source code. Afterwards, *Oracle* parses the upstream kernel modules' source code and detects all the configuration options (O_{ups}) that modify the layout of the data structures (*DS*) used by these modules (also used by their counterpart modules in F_{image}). These options are considered "unsafe" since, if they were included in `Pandawan`'s kernel build process, they would misalign the layout of these data structures. Next, *Oracle* proceeds to filter out any configuration options from $O_{MIPS/ARM}$ that belong to the set O_{ups}. I consider the remaining configuration options ($O_{safe} = O_{MIPS/ARM} - O_{UP}$) as "safe" options that can be included in `Pandawan`'s kernel build process to supplement its kernels with the `user-code-required` functionality. I note that *Oracle* cannot detect the data structures exclusively used by the proprietary kernel modules in F_{image} since these modules do not have a counterpart in UP_{kos}. However, these modules can indirectly benefit from the *Oracle* process if they use any data structures in *DS*, since *Oracle* prevents these data structures

from becoming misaligned.

Step S5 Finally, Pandawan includes the O_{safe} options in its kernel build process. Next Pandawan produces the PW_k kernels which are conducive to both *user* and *kernel* (in the form of binary kernel modules) code re-hosting. For images that contain KALLSYMS in their kernels, *Oracle* is redundant as Pandawan simply obtains all the necessary configuration options corresponding to the symbols in KALLSYMS during S1.

Firmware Re-hosting & Analysis

The goal of component ② of Pandawan is to initiate an emulation run using a modified version of F_{image}'s file-system (F_{fs}), the PW_k kernel and a dedicated PyPANDA script and analyze F_{image} using PyPANDA's plugins. Component ② is a two-pronged process; 1) Pandawan creates the F_{fs} file-system and PyPANDA script for F_{image}, and 2) it holistically emulates and analyzes F_{image}'s code with PyPANDA and TriforceAFL.

File-system and PyPANDA script creation. At first, Pandawan uses the custom file-system and network configuration process of FirmAE to create the F_{fs} file-system and set the network connectivity of F_{image}. Specifically, the custom file-system creation process proceeds as follows. Similar to FirmAE and FirmSolo, Pandawan uses binwalk[2] to extract F_{image}'s file-system, but ensures that it supports Pandawan's kernel module loading logic. The F_{fs} file-system also contains a custom serial console binary, which Pandawan uses to issue commands to the emulated firmware (e.g., run TriforceAFL's fuzzing harness). Since the original startup scripts within the F_{fs} file-system are responsible for loading any of the kernel modules, Pandawan has to ensure that these scripts do not load the kernel modules that crashed during the hybrid kernel build process in S5. Pandawan applies any kernel module substitutions in-place, meaning that if a kernel module c that crashed during S5 has a substitution s (i.e., a counterpart in UP_{kos}), then c is replaced by s within F_{fs}, else c is deleted from F_{fs} altogether. Thus, when a startup script attempts to load c during

[2] https://github.com/ReFirmLabs/binwalk

emulation, it will either load s instead or no module at all.

To infer the networking configuration of F_{image}, `Pandawan` initiates an emulation run with the F_{fs} file-system, the PW_k kernel and QEMU. During emulation, `Pandawan` gathers information about the networking interfaces (e.g., bridge mode) and IP configuration of F_{image}. Finally, `Pandawan` embeds F_{image}'s network configuration within a PyPANDA script, used to emulate and analyze F_{image} with PyPANDA and its plugins.

`Pandawan` also addresses kernel module crashes that occur during its re-hosting experiments (see Section 5.3.3). In these cases, `Pandawan` first uses FirmSolo's data structure layout recovery mechanism to address the errors. If the crashes persist then `Pandawan`, during the creation of F_{fs}, substitutes the crashing modules with their counterparts in UP_{kos} (if they exist) else it removes the crashing modules from F_{fs} entirely.

Holistic Re-hosting & Analysis. Next, `Pandawan` initiates a firmware emulation run using the PW_k kernel, the F_{fs} file-system and PyPANDA. During emulation, `Pandawan` enables the OS introspection (Py)PANDA plugins `osi_linux` (enabled by default), `coverage`, and `syscalls_logger` to analyze F_{image}. Primarily, the information collected are the executed program names, their executed QEMU Translation Blocks (TBs), and the system calls (ID and arguments) that were invoked by *user* code during the emulation. I note here that I modified PyPANDA's `coverage` plugin, in its *summary* mode, to not only provide information about the programs and the number of unique TBs executed, but also the addresses of the TBs along with their origin (e.g., the program itself or the symlink target if the program is a link). `Pandawan` also introduces two additional custom plugins; the `FICD` and the `SyscallToKmodTracer`.

As the name suggests the former plugin implements the `FICD` technique in `Pandawan`, while the goal of the `SyscallToKmodTracer` is to showcase `Pandawan`'s utility in holistic firmware analysis. Specifically, `SyscallToKmodTracer` traces system calls invoked by *user* code that lead to the execution of code in kernel modules. These traces expose en-

try points from user level to kernel level, which `Pandawan` transforms into seeds for the TriforceAFL kernel fuzzer to analyze the binary kernel modules. Essentially, `Pandawan` leverages the hooking infrastructure of PyPANDA to hook all system calls. Upon entering a system call, the corresponding hook triggers and `Pandawan` stores information about the execution context at the time (i.e., the current process name, its `pid` and creation time) and correlates the running system call (ID) with that context. Keeping track of the different execution contexts and their running system calls is mandatory, since the kernel can alternate between different contexts via *context switches*. `Pandawan` uses this information to detect the context in which the kernel module code is executed in and also the system call that led to the execution of that kernel module code.

To detect when kernel module code is executed, `Pandawan` also places a hook on memory regions occupied by kernel modules depending on the underlying architecture of F_{image} (e.g., `0xc0000000 - 0xc2000000` for MIPS). Specifically, if a system call leads to code residing in these regions then the aforementioned hook will trigger. Within the hook, `Pandawan` checks the current execution context and iterates all the previously traced execution contexts (captured by the system call hooks) until it finds a match. When it does, `Pandawan` correlates the current kernel module code address (i.e., the start of the TB currently executed) with the current execution context and the (running) system call associated with that context. The aforementioned system call is considered the one that led to the kernel module code execution.

Seed Creation. However, to create useful seeds for the TriforceAFL fuzzer, `Pandawan` also requires information about the arguments (i.e., values) of the system calls that lead to the execution of kernel module code. Another PyPANDA plugin, the `syscalls_logger`, is responsible for collecting the arguments of all the system calls executed during the emulation. `Pandawan` combines the data collected by the `syscalls_logger` and `SyscallToKmodTracer` plugins to create seeds and fuzz the firmware kernel modules.

Seed generation for fuzzers is not a new research topic. Prior works (Chen et al., 2021; Bars et al., 2023; Pailoor et al., 2018; Shen et al., 2022; Zhao et al., 2023) leverage either system call traces, concolic execution or static analysis to extract information about the program execution and use it to create seeds for popular fuzzers such as syzkaller (Google, 2023) or AFL/AFL++ (Zalewski, 2017; Fioraldi et al., 2020). My approach is similar to (Pailoor et al., 2018) in the sense that it relies on system call traces to create seeds for the TriforceAFL kernel fuzzer. However, I target only kernel modules that are accessed by system calls either through a `socket` or a `file descriptor` since TriforceAFL's fuzzing agent (Newsham, 2017) is inherently compatible with these types of system calls.

As I show in Section 5.3, the majority of TriforceAFL compatible system calls (*sc*) that lead to kernel module code execution are networking (e.g., `sys_setsockopt`) or file-based system calls (e.g., `sys_write`). Thus, to create valid seeds for the fuzzer, `Pandawan` first has to detect which `sys_socket` and `sys_open` system call creates the `file descriptor` accessed by each *sc*, respectively. To gain this information, `Pandawan` leverages the first argument of a *sc* which corresponds to the file descriptor number fd. Then for each *sc*, `Pandawan` parses the information of `syscalls_logger` and detects the `sys_socket` or `sys_open` system call that was invoked by the same process as *sc* and returned a value equal to fd. System calls that operate on the same fd are grouped together in the same seed.

Fuzzing. To start fuzzing the kernel modules in F_{image}, `Pandawan` runs the PW_k kernel along with the F_{fs} file-system and the seeds found previously under TriforceAFL. The I_{fin} point for F_{image} plays a critical role also in this scenario. In particular, `Pandawan` spawns a thread that runs alongside the TriforceAFL process and waits until F_{image} reaches its I_{fin} point, since all the target kernel modules would be loaded by that point. Then, the thread connects through a `UNIX` socket to the custom serial console within the F_{fs} file-system and initiates the fuzzing harness to begin the kernel module fuzzing. I note here that `Pandawan`

System Call	# of Invocations
sys_setsockopt	1,769,439
sys_ioctl	671,531
sys_getsockopt	648,336
sys_sendto	424,880
sys_close	123,338
sys_read	43,126
sys_open	42,847
sys_write	32,187
sys_send	10,564
sys_fcntl	4,611
sys_socket	3,584
sys_connect	1,115

Table 5.1: The most popular system calls used in `Pandawan`'s fuzzing experiments.

System Call	# of Invocations
sys_recvmsg	301,866
sys_select	293,624
sys_nanosleep_time32	291,664
sys_nanosleep	211,628
sys_gettimeofday	85,761
sys_brk	49,077
sys_fork	23,625
sys_execve	19,625
sys_rt_sigprocmask	14,718
sys_wait4	13,704

Table 5.2: The top 10 traced system calls that are not "compatible" with TriforceAFL.

limits the range of system calls that the fuzzing harness can execute to the twelve system calls illustrated in Table 5.1. I opt to fuzz only these system calls due to their inherent compatibility with TriforceAFL's fuzzing agent (harness), TriforceLinuxSyscallFuzzer (Newsham, 2017). Even though the fuzzing agent is capable of invoking any system call, it primarily supports `socket` or `file-descriptor` based system calls (see Table 5.1).

Table 5.2 illustrates the top ten traced system calls, which lead to kernel module code execution, but not fuzzed during my fuzzing experiments. I opt out from fuzzing these system calls for two reasons. First these system calls do not operate on a `file descriptor` or `socket` (i.e., the most compatible system calls with TriforceAFL's fuzzing agent). Second, invoking these system calls does not lead to kernel module code being executed until certain conditions are met. For example, `sys_select` is used to monitor `file descriptors` until they are ready for I/O, which the fuzzer is not sophisticated enough to generate. Also fuzzing this specific system call would not produce any interesting results (i.e., crashes). The same premise is true for the rest of the system calls I did not fuzz.

5.3 Evaluation

In this Section, I first evaluate Pandawan's effectiveness on holistic IoT firmware re-hosting and analysis and then FICD's capability to meaningfully compare full-system re-hosting approaches. Specifically, I answer the following three research questions:

RQ1 Is Pandawan capable of holistically re-hosting firmware code (§ 5.3.3)?

RQ2 How effective is Pandawan on enabling holistic firmware analysis (§ 5.3.4)?

RQ3 How efficient is FICD when it comes to quantifying the forward progress in full-system IoT firmware re-hosting (§ 5.3.5)?

First, I describe my dataset and my experimental setup, then I detail the experiments I use to evaluate Pandawan and finally discuss the comparative experiments with FICD.

5.3.1 Dataset

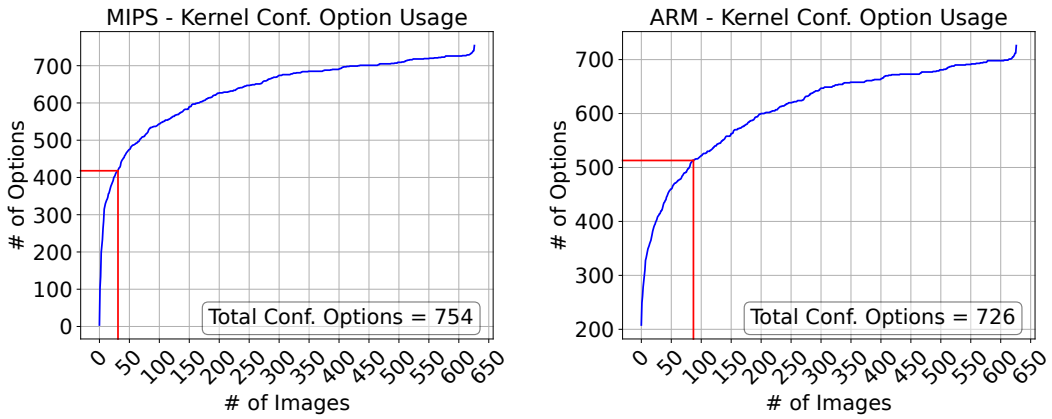
For my evaluation, I use the dataset of FirmSolo and a subset of firmware images used in Greenhouse (Tay et al., 2023). The Greenhouse dataset was shared with me upon request to the authors. I use the FirmSolo dataset since it is supported by FirmSolo, Firmadyne, and FirmAE. I do not use the entire Greenhouse dataset (consisting of 7,347 firmware images) since the majority of the images use the same kernel version as the images in the FirmSolo dataset. Instead, I pick at random 50 images from the 97 images that use a kernel version between 4.4.198 (latest version used by FirmSolo) and 4.9.206 (the latest kernel version used by the ARM/MIPS 32bit images in Greenhouse). My dataset consists of 1,520 firmware images containing a total of 61,319 binary kernel modules. The firmware images in the dataset target the MIPS (984 images) and the ARM (536 images) platforms. Finally, the firmware images span a total of 95 unique Linux kernel versions, ranging from version 2.6.18 to version 4.9.206.

5.3.2 Experimental Setup

I run all my experiments Intel Xeon machines using 2 cores with minimum of 16GB of RAM. All of my re-hosting experiments consist of three runs and the results correspond to the averages across these runs.

5.3.3 Holistic Re-hosting

In this section I evaluate the ① `Kernel Augmentation` and ② Firmware Re-hosting & Analysis components of `Pandawan` to answer RQ1.



(a) CDF of the configuration option usage in O_{MIPS}.

(b) CDF of the configuration option usage in O_{ARM}.

Figure 5·3: CDFs of the configuration option popularity/usage in O_{MIPS} and O_{ARM}, respectively. I use the Kneedle algorithm to detect the "knee" points in both figures. The options used by the number of images below the "knee" points are filtered out.

Preprocessing. I execute this step only once in my experiments, to generate the configuration option pools O_{MIPS} and O_{ARM}, which combined constitute the `user-code-required` functionality that is added in the PW_k kernels in component ①. I include these configuration option pools as part of `Pandawan`'s repository. During this step, I extract 754 options in the O_{MIPS} and 726 options in the O_{ARM} pools, respectively. I opt to only keep the most popular options in both sets. In Figures 5·3a and 5·3b, I pro-

vide the Cumulative Distribution Function (CDF) of the configuration options in O_{MIPS} and O_{ARM} pools, respectively, over the images that have `KALLSYMS` available. To find the most popular options in each pool I use the Kneedle algorithm (Satopaa et al., 2011) to deduce the "knee" points in both figures (marked with the red lines). I filter out the options that reside beneath these knee points. Specifically, the knee point in Figure 5·3a is 30 (images), 418 (options), while the knee point in Figure 5·3b is 85 (images), 513 (options). In the end, 336 (754 - 418) and 213 (726 - 513) options remain in the O_{MIPS} and O_{ARM} pools, respectively.

Kernel Augmentation. Regarding the unresolved symbols required by kernel modules to load into the PW_k kernels, `Pandawan` successfully adds the stubs for all these 4,254 unique symbols into the PW_k kernels' source code. Next, I evaluate the *Oracle* only on the subset of 570 (38%) firmware images that do not contain a `KALLSYMS` entry, since the *Oracle* only affects the PW_k kernels produced for these images. After S4, 200 configuration options (out of O_{MIPS} and O_{ARM}) on average remain as the "safe" options that can be included in S5 to produce the PW_k kernels for the images without a `KALLSYMS` entry. Regarding the 950 firmware images that contain a `KALLSYMS` entry, there are no additional options included in S5, thus the PW_k kernels are identical to the FS_k kernels produced (by default) by FirmSolo.

Network configuration. Since `Pandawan` leverages FirmAE's network configuration logic, every PyPANDA script by default sets at least a networking interface in PANDA for each firmware image. Unfortunately, I notice that the PyPANDA/PANDA emulation hangs for the ARM images in my dataset that target the `Versatile` and `Realview` platforms (498/536 images) when the networking interfaces are enabled. The remaining 38 images target the "dummy" `virt` QEMU platform and are unaffected. Since FirmAE's pre-built ARM kernel targets the `virt` platform, FirmAE does not suffer from the same issue. Thus, I disable these interfaces for the affected images to successfully re-host them with `Pandawan`.

I have notified the PyPANDA/PANDA developers about this issue.

Re-Hosting Success. Component ② of `Pandawan` is responsible for initiating an emulation run, using the F_{fs} file-system and the PW_k kernels while concurrently enabling PyPANDA's OS level introspection and analysis plugins. `Pandawan` successfully re-hosts 1,389 (91%) of all the images in my dataset. I define as successfully re-hosted all the cases where the emulation successfully executed `init` without panicking or freezing. Unfortunately, `Pandawan` cannot re-host the remaining 123 firmware images since for 38 images there is no working `init` script available in the file-system, `init` immediately crashes for 12 images or important libraries are missing in the file-system for 11 images. Additionally, 29 images use a kernel with `MIPS Thread Context`, a feature that improves parallelism in MIPS systems, but not supported by QEMU. Furthermore, 2 images use a very old kernel version (`2.6.18`), resulting in the emulation freezing while the kernel boots up. I also notice 13 cases where the kernel cannot not mount the file-system because the latter is corrupted and 8 cases where the kernel hangs while mounting the file-system. Finally, PyPANDA crashed for 18 images during emulation, while triggering a bug in its `osi_linux` plugin (used by default by PANDA). I have notified the (Py)PANDA developers about the issue. Since these failures are not specific to `Pandawan`'s implementation, I do not consider them as a limitation of my work.

Issue	**# of Images**
Console is unresponsive	82
The console binary crashes	59
The kernel panics	45
Firmware reboots	3
Total	**189**

Table 5.3: Console connectivity issues during the `Pandawan` emulation experiments. Column one illustrates the reasons behind the console connectivity issues.

Serial Console Connectivity. As discussed in Section 5.2.2, to run the fuzzing harness within a target firmware image, `Pandawan` has to first connect through a `UNIX` socket to a

custom serial console within the image's file-system. To discover the number of firmware images that are accessible through their serial console, I conduct an experiment where I emulate each firmware image with `Pandawan` and attempt to connect to its serial console through a `UNIX` socket after 30 seconds of emulation time and run a dummy command (e.g., `/bin/ls`). A successful connection with a serial console means that I get the output of the dummy command in the serial log output of the firmware image. I am able connect to the serial console of 1,200 (86%) out of the 1,389 firmware images that `Pandawan` successfully re-hosts. I "break down" the reasons behind the console connectivity issues for the 189 failed cases in Table 5.3.

Kernel Module Re-hosting. During the firmware emulation runs in component ②, `Pandawan` loads 14,413 (24%) kernel modules out of the 61,319 IoT kernel modules in my dataset. As illustrated in (Angelakopoulos et al., 2023), the configuration and bootup scripts dictate which kernel modules will be loaded into the kernel during the firmware initialization, which is merely a fraction of all the kernel modules in the images' file-systems. Finally, `Pandawan` executes 336 kernel module TBs on average.

In cases where kernel module crashes occur during emulation, `Pandawan` uses its custom crash solving method (see Section 5.2.2) to address the issue. In total, `Pandawan` addreses 249 kernel module errors.

User Code Re-hosting. When considering its *user* code re-hosting capabilities, `Pandawan` executes 30 user level programs and 15,671 unique QEMU TBs on average. The startup scripts within the firmware images dictate which programs are executed during bootup. I make sure to substitute symlinks with their actual targets in my results so that the TBs executed get attributed to their actual origin. For example, if the firmware executes `/bin/ls` which is a symlink to `/bin/busybox`, I only include and count the TBs executed towards the `busybox` executable in my results. Note that each TB is attributed only once (counted one time only) to their origin. For instance, a TB that belongs to a shared library

Dataset	All Images (/w cov) (1328)				No KALLSYMS (/w cov) (456)				KA Ablation Dataset (/w cov) (135)			
Framework	**FD**	**FAE**	**FS**	**P**	**FD**	**FAE**	**FS**	**P**	**No KA**	**KA w/o oracle**	**KA w/o stubs**	**KA**
Avg. Progs	**30**	**31**	**31**	31	**35**	**36**	**34**	36	**32**	**32**	33	34
Avg. TBs	**15,483**	**16,767**	**15,523**	16,360	**15,715**	16,552	**13,835**	16,740	**15,099**	**14,961**	**16,611**	16,790
KOs Loaded	0	0	**13,598**	14,089	0	0	**4,936**	5,146	**1,481**	**1,470**	1,535	1,521
Avg. KOs TBs	0	0	**323**	336	0	0	**200**	251	**266**	**322**	**280**	310

Table 5.4: Comparison results between the Pandawan (P), Firmadyne (FD), FirmAE (FAE) and FirmSolo (FS) re-hosting frameworks. The table depicts the average number of user programs and QEMU TBs executed, the number of kernel modules loaded and kernel module TBs executed, for each system respectively. The green cells represent the best results for each metric. The bold values represent the cases with statistical significance, where p-value $p < 0.05$ (using the Wilcoxon signed-rank test).

and executed by multiple programs will only be counted once towards the shared library it belongs to. In Section 5.3.5, I discuss how these results compare against Firmadyne, FirmAE and FirmSolo.

Kernel Augmentation Ablation Study. To demonstrate the contribution of `KA` in firmware re-hosting, I also conduct an ablation study where I measure the effectiveness of the two key features of `KA`: 1) The addition of the `user-code-required` functionality by the `Oracle` and 2) the addition of the kernel symbol stubs (required by kernel modules to load into the kernel). For this experiment I use a smaller dataset of 150 firmware images (100 images without `KALLSYMS` and 50 images with `KALLSYMS`). I bias my selection towards images without `KALLSYMS` since `KA` benefits these images the most. I present the results of this study in Table 5.4 in the third group column. Based on my results the contribution of `KA` is apparent when taking into account all of my metrics. Specifically, `Pandawan` with `KA` executes 6% more user level programs and executes 11% more TBs than `Pandawan` without `KA` (i.e., FirmSolo), while also loading 3% more kernel modules and executing 17% more kernel module TBs. Finally, my experiments show that the addition of only the `user-code-required` functionality by the `Oracle` is more beneficial than simply adding the kernel symbol stubs, since the majority of the metrics are the closest to `Pandawan` with `KA`. Specifically, `Pandawan` with only the `Oracle` enabled manages to surpass `Pandawan`

Plugins	I_{fin} Avg. (sec)	Overhead (%)
All	496	22
No `syscalls_logger`	496	22
No `coverage`	460	13
No `SyscallToKmodTracer`	440	8
Only `coverage`	462	14
Only `SyscallToKmodTracer`	456	12
Only `syscalls_logger`	413	2
Only `FICD`	407	0
Frameworks		
FirmSolo	476	17
Firmadyne	441	9
FirmAE	477	17

Table 5.5: `Pandawan`'s plugin performance ablation study. The experiment includes every plugin combination used by `Pandawan`. The second column provides the average I_{fin} points marked by `FICD` for every plugin combination. The third column provides the performance overhead incurred by each plugin combination. The table also includes the average I_{fin} measured for FirmSolo, Firmadyne and FirmAE on the same dataset with all the plugins enabled.

with `KA` in kernel module loading by 1%. Upon further inspection, two images whose the kernel modules crashed and deleted from the F_{fs} file-system (see Section 5.2.2) during the emulation with `Pandawan` with `KA`, skew the results in favor of `Pandawan` with only the `Oracle` enabled.

Plugin Overhead. To measure the performance overhead of each PyPANDA plugin, I pinpoint the I_{fin} points using every possible combination of the plugins enabled in `Pandawan`'s analysis (i.e., `SyscallToKmodTracer`, `syscalls_logger`, and `coverage`) excluding `FICD` which is responsible for determining the I_{fin} points and thus is always enabled. For this study I use again a smaller dataset of 150 images (95 with `KALLSYMS` and 55 without `KALLSYMS`) that better represents the distribution of the images in my dataset. I provide the results in Table 5.5. When all the plugins are enabled I notice a 22% slowdown compared to enabling only `FICD`. This is expected since each plugin contributes an additional level of analysis which impacts the overall emulation speed. Specifically in my case, the most computational heavy plugins are `coverage` and `SyscallToKmodTracer`

since they constantly track the execution of QEMU TBs during the emulation, incurring a slowdown of 14% and 12%, when enabled individually and a slowdown of 22% when both are enabled.

5.3.4 Holistic Analysis

To demonstrate `Pandawan`'s contribution to holistic firmware analysis and answer RQ2, I evaluate how the information collected by the `SyscallToKmodTracer` plugin is used to further analyze the binary kernel modules with TriforceAFL.

Seed Generation. The `SyscallToKmodTracer` plugin monitors the interactions between the *user* and *kernel* code during the firmware emulation run in component ②. `Pandawan` processes the information collected by the `SyscallToKmodTracer` plugin (i.e., system calls that lead to kernel module code execution) and generates seeds that are used by TriforceAFL, as explained in Section 5.2.2.

During my experiments, the plugin identified 927 out of the 1,389 successfully rehosted firmware images that load at least a kernel module. In addition, within these images, 353 processes invoke 6,954 system calls on average that lead to kernel module code being executed. The remaining 462 firmware images did not successfully load kernel modules.

Unlike FirmSolo which targets only the IOCTL system call, `Pandawan` can trace and generate seeds for all types of system calls that interact with kernel modules. However, I opt to create seeds only for cases involving popular system calls that interact with kernel modules through a file descriptor created either by a `sys_socket` or `sys_open` system call, due to their compatibility with TriforceAFL's fuzzing agent. I also provide details about the system calls I did not fuzz in Table 5.2. In the end, `Pandawan` creates seeds for 479 images whose *user* code invokes system calls (see Table 5.1) which lead to the execution of kernel module code. Out of these images, 466 (97%) have serial console connectivity, thus can be fuzzed by TriforceAFL.

Kernel Module Fuzzing. After the seed creation, `Pandawan` uses TriforceAFL to fuzz

Module	Type	Vendor	Kernel	Paths	Path var (std)	Bugs
MIPS						
arp_tables	O	AT&T	2.6.31	121	43 (7)	2
led	P	Linksys	2.6.31	26	39 (6)	1
ipt_STAT	P	TP-Link	2.6.36	59	28 (5)	1
x_tables	O	TP-Link	2.6.31	153	134 (12)	1
statistics	P	TP-Link	2.6.31	166	81 (9)	1
ip6_tables	O	AT&T	2.6.30.10	252	129 (11)	2
ipv6_spi	P	Netgear	2.6.30	43	245 (16)	2
ip_tables	O	TP-Link	2.6.31	202	19 (4)	2
gpio	P	DLink	2.6.31	31	18 (4)	1
gpio_module	P	DLink	2.6.31	5	0 (1)	1
ARM						
ipt_STAT	P	TP-Link	2.6.32.11	56	47 (7)	1
statistics	P	TP-Link	2.6.36.4	25	121 (11)	1
					Total	16

Table 5.6: Statistics about `Pandawan`'s fuzzing experiments with TriforceAFL. The `O` and `P` in column two represent kernel modules that are *open-source* and *proprietary*, respectively. Column six provides the variance and standard deviation for the paths found by the fuzzer over the ten runs.

the IoT modules within the firmware images. I run all experiments ten times for 12 hours.

While `Pandawan` creates seeds for 479 firmware images, I chose to fuzz only the kernel modules in a subset of 20 randomly selected firmware images (with serial console connectivity to invoke the fuzzing harness) due to computational resource constraints. I provide the information about my fuzzing campaigns in Table 5.6. The table provides the average number of paths found by TriforceAFL, the path variance and standard deviation over the ten runs and the number of bugs I confirmed, for each image. Based on the path variance and standard deviation measured (see column six in Table 5.6), the coverage found in all the fuzzing campaigns is consistent (the variances and standard deviations are insignificant).

Specifically, TriforceAFL triggers 16 bugs in 12 kernel modules (8 proprietary and 4 open-source – see Table 5.6). The bugs fall into the stack corruption (6), arbitrary memory reads and writes (7), and large virtual memory allocation (3) categories. Three of these bugs (`gpio` (1) and `ipv6_spi` (2)) were also detected by FirmSolo. In addition, the

Pandawan			FirmSolo		
Module	**TTC (sec)**	**TTC var (std)**	**Module**	**TTC (sec)**	**TTC var (std)**
MIPS					
ipv6_spi	1,644	14e+5 (1e+3)	ipv6_spi	16,262	126e+6 (11e+3)
gpio	751	114e+3 (337)	gpio	51	678 (26)
arp_tables	1,488	24e+4 (490)	acos_nat	70	570 (24)
led	n/a	n/a	art	n/a	n/a
ipt_STAT	3,856	25e+6 (5e+3)	art-wasp	151	39e+2 (62)
x_tables	880	200e+3 (448)	edinvram2	25	379 (20)
statistics	152	5,071 (71)			
ip6_tables	811	536e+3 (732)			
ip_tables	1,811	72e+3 (270)			
gpio_module	124	3 (2)			
ARM					
ipt_STAT	2,776	18e+5 (1.4e+3)	gpio	189	91e+3 (302)
statistics	169	6,878 (83)	IDP	28,919	74e+6 (86e+2)
			smcdrv	7	1 (1)
			u_filter	1,029	440e+3 (663)
Avg.	**1,315**			**5,189**	

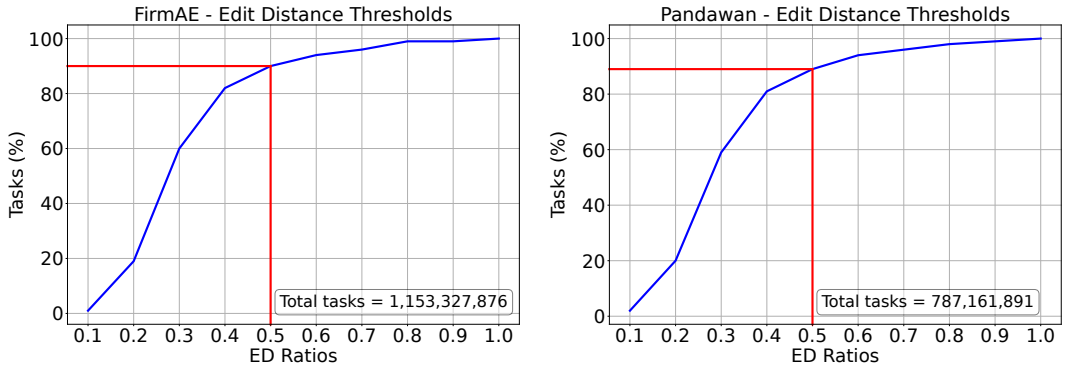
Table 5.7: `Pandawan`'s and FirmSolo's bug finding speed measurements. I run the experiments 10 times for 12 hours and the results are the averages over these runs. `TTC` in column two and four stands for `Time-To-Crash` (i.e, the time it takes the fuzzer to reach the first crash). Columns three and six provide the measured variance and standard deviation for `TTC`. Regarding the `led.ko` kernel module, in line four, TriforceAFL registers the bug as a *hang* and not a *crash*, hence there is no `TTC` in this case. The `art.ko` kernel module in line two crashed while being loaded into FirmSolo's kernels and thus the measurement is invalid.

seven bugs triggered for the open source kernel modules (`arp_tables` (2), `ip6_tables` (2), `ip_tables` (2), and `x_tables` (1)) are related to known bugs (`CVE-2016-4998` and `CVE-2016-3135`), which is why I did not report these bugs to the Linux kernel developers. Finally, the six remaining bugs on the proprietary kernel modules `led` (1) `ipt_STAT` (2), `gpio_module` (1), `statistics` (2) are previously unknown bugs. I have disclosed these bugs to the respective vendors, and two vendors acknowledged my findings; `TP-Link` and `DLink`.

5.3.5 Comparative Results

The previous evaluation based on the coarse-grained "number-of-bugs-found" metric

shows that Pandawan finds more diverse bugs than FirmSolo. Even when it comes to bug finding speed, Pandawan is four times faster on average than FirmSolo (see Table 5.7). However, FICD helps to assess re-hosting process on a finer granularity than just bugs found. Thus, in this Section, I evaluate FICD's utility in comparing full-system re-hosting approaches based on the emulation-based metrics. Specifically, I compare the Pandawan , FirmAE, Firmadyne and FirmSolo re-hosting frameworks to answer RQ3.



(a) FirmAE edit distance threshold experiment.

(b) Pandawan edit distance threshold experiment.

Figure 5·4: Edit distance threshold experiments for FirmAE and Pandawan. For both frameworks, I measure the similarity between a task and all the tasks created at an earlier point in time using the *Levenshtein* edit distance algorithm.

As mentioned previously, the FICD technique requires two parameters; 1) the task similarity threshold h for the *Levenshtein* edit distance (ed) which Pandawan uses to identify previously unseen tasks, and 2) the time frame tf which indicates how long can the emulation continue without the firmware executing a previously unseen task. For my experiments, I set the threshold h equal to 0.5 (for all re-hosting frameworks) and tf equal to 300 seconds for Pandawan and FirmSolo and 220 seconds for FirmAE and Firmadyne, respectively. To obtain these values I initiate a ten minute experimental run with FirmAE and Pandawan for all the images in the dataset, without the FICD plugin enabled. I set the h and tf of Firmadyne and FirmSolo the same as FirmAE's and Pandawan's due to the

re-hosting similarities shared between the frameworks (i.e., Firmadyne with FirmAE and FirmSolo with `Pandawan`). To calculate the threshold h, I first sort all tasks based on their creation time. Then, for each task in the sorted list I use `ed` to calculate its similarity against all the tasks with an earlier creation time, as a ratio between 0 and 1. According to Figures 5·4a and 5·4b, almost 90% of the tasks have a similarity less than 0.5 with any other task created earlier, for both FirmAE and `Pandawan`, thus I choose $h = 0.5$. Then, to get the time frame tf, I measure the average $\bar{t}$ and standard deviation (σ) of the time that passes between the execution of previously unseen tasks. Based on the ten minute experiment runs, FirmAE's $\bar{t} = 52sec$ and $\sigma = 81sec$, while `Pandawan`'s $\bar{t} = 69sec$ and $\sigma = 115sec$. To ensure statistical relevance I choose a time frame $tf > \bar{t} + 2\sigma$, thus I set $tf = 220sec$ as my optimal time frame for FirmAE and Firmadyne and $tf = 300sec$ for `Pandawan` and FirmSolo, respectively.

Re-hosting Fidelity Comparison. To showcase `Pandawan`'s progress on holistic firmware re-hosting, I use the`FICD` technique to compare it with the Firmadyne, FirmAE, and FirmSolo re-hosting frameworks, based on the number of executed user level programs (non symlinks), the QEMU TBs executed in these programs, the number of kernel modules loaded, and the kernel module TBs executed during emulation. To make my comparison fair among all compared re-hosting frameworks I only include images where I collected coverage in all three emulation runs with a maximum total emulation time of $45min$ for all frameworks. Thus, I compare `Pandawan` , Firmadyne, FirmAE, and FirmSolo on 1,328 out of the 1,520 images in my dataset. I note here that throughout all my experiments `FICD` was successful in detecting the I_{fin} point for each framework and each image and terminating the emulation way before the $45min$ global timeout (see Table 5.5).

`Pandawan` executes up to 3% more user level programs on average than Firmadyne, FirmAE and FirmSolo (see column group 1 in Table 5.4). Similarly, `Pandawan` outperforms both Firmadyne and FirmSolo on the average number of TBs executed by up to 5%.

However, FirmAE executes 3% more TBs on average than Pandawan. The predominant reasons behind this outcome are twofold; 1) the emulation speed of Pandawan is slower than FirmAE's due to the SyscallToKmodTracer affecting only Pandawan (and FirmSolo) which loads and analyzes the IoT kernel modules, and 2) KA impacts negatively the re-hosting progress. Since FirmAE and Firmadyne focus solely on user level re-hosting and do not load any firmware kernel modules, the plugin's module TB tracing functionality is ineffective for these frameworks. As illustrated in Table 5.5 both Firmadyne and FirmAE are 13% and 4% faster than Pandawan on average. In addition, for 36 out of the 1,328 images used in the comparison experiments, the PW_k kernels produce a crash (i.e., Oops), while the FS_k kernels do not suffer from the same issue. The addition of the user-code-required functionality and the kernel symbol "stubs" by KA hinders the re-hosting progress of the images in these cases. This small regression can be attributed to the heuristic nature of KA.

When it comes to the kernel module loading I compare only Pandawan and FirmSolo, since neither Firmadyne nor FirmAE load IoT kernel modules. Specifically, Pandawan loads 6% more kernel modules and executes 5% more kernel module TB's on average than FirmSolo. To be fair towards FirmSolo I addressed the crashes during its emulation runs using Pandawan's crash solving methods (see Section 5.2.2). In this case, I address 225 crashes in total.

Since Pandawan only includes the user-code-required functionality in images without a KALLSYMS entry (see Section 5.1.2), the improvements of Pandawan over FirmSolo can be better observed on these images (see column group 2 in Table 5.4). In particular, Pandawan executes 6% more user programs and 21% more TBs on average than FirmSolo. Pandawan also outperforms FirmSolo on the number of kernel modules loaded and kernel module TB's executed by 9% and 26% on average, respectively. Both the user-code-required functionality and the symbol "stubs" added by Pandawan in the

Data	Avg. TBs			
Framework	**FD**	**FAE**	**FS**	**P**
httpd	1,784	1,886	1,803	1,790
uhttpd	268	279	249	252
mini_httpd	751	782	743	744
lighttpd	2,281	2,448	2,267	2,270
goahead	194	198	188	185
httpd	699	637	596	595

Table 5.8: Coverage information of popular IoT webservers.

PW_k kernels for these images, greatly benefit the progress of holistic (*user* and *kernel* code) re-hosting as well as the loading of additional kernel modules.

I also test the statistical significance of my results using the Wilcoxon signed-rank test (Solutions, 2024). Specifically, for each metric I compare `Pandawan`'s data with Firmadyne's, FirmAE's and FirmSolo's data (28 comparisons). I exclude the kernel module related metrics (loaded and TBs executed) for Firmadyne and FAE since they do not load the IoT kernel modules. I note that the majority of my measurements (24/28) are statistically significant with a p-value $p < 0.05$ (see Table 5.5).

Webserver Re-hosting. To further showcase the efficiency of `Pandawan` in *user* code re-hosting, I conduct a study about the re-hosting of webservers used by firmware images. I provide the results in Table 5.8. Specifically, I collect coverage information for 6 popular webservers used in IoT (Kim et al., 2020). My findings show that FirmAE and Firmadyne which are the state-of-the-art in *user* code re-hosting marginally outperform `Pandawan` in re-hosting webservers by executing up to 8% more QEMU TBs across all the servers in my study. Given the fact that `Pandawan` aims to holistically re-host and analyze Linux-based IoT firmware, `Pandawan`'s slight deficiency in webserver re-hosting is an acceptable tradeoff. I leave the qualitative analysis of the differences in webserver re-hosting between the compared frameworks as future work.

5.4 Discussion

In this Section I discuss future applications of FICD and Pandawan. With the introduction of FICD, developers can efficiently improve upon the existing works by comparing their implementations with the state-of-the-art. Furthermore, Pandawan could be leveraged to develop future frameworks whose re-hosting capabilities closely approximate physical IoT devices. For instance, the re-hosting techniques implemented in Honware (Vetterl and Clayton, 2019) could be integrated into Pandawan to further improve its holistic re-hosting capabilities. In addition, the FICD and setup of Pandawan (i.e., using PyPANDA for firmware emulation) are generic and OS agnostic. Thus, any firmware that can be emulated by QEMU and uses a concept of process/task (e.g., FreeRTOS (FreeRTOS, 2023)) can be adapted to leverage FICD. Next, I explore the categories of metrics that could be used to quantify re-hosting progress and lastly specify certain limitations of both FICD and Pandawan.

Alternate Metrics. In general, I can distinguish the metrics that quantify the emulation progress in two categories; system- and process-oriented metrics. My existing metrics; number of programs executed and number of kernel modules loaded fall into the system-oriented metrics, since they showcase emulation progress from a higher (system) level of abstraction. Other metrics in this category could be the number of networking interfaces successfully initialized and/or the successful firewall configuration (e.g., the number of successfully applied rules).

The average number of QEMU TBs traced belong in the process-oriented (lower-level) metrics, since (Py)PANDA correlates each TB executed to a running program or kernel module. Another metric that could fit in this category is the number and identity of successful and unsuccessful system calls executed by each process. In summary, the metrics used in this chapter work well and other metrics can be easily integrated.

Regarding techniques to assess a successful firmware initialization, generally there is

no reliable indicator to establish the end of a firmware's initialization phase. However, 33 of my images print a specific message about the end of bootup and timewise this message coincides with I_{fin}- tf, thus I_{fin} accurately corresponds to these images' end of initialization phase.

Limitations. As far as `FICD`'s limitations are concerned, the time frame tf is directly dependent on which (Py)PANDA plugins are enabled during a firmware emulation run. When many computationally heavy plugins are enabled, the emulation speed will decrease, thus tf requires an adjustment. In my re-hosting experiments I use three (Py)PANDA plugins (`SyscallToKmodTracer`, `syscalls_logger`, and `coverage`) along with the `FICD` plugin, which results in the optimal $tf = 220sec$ and $tf = 300sec$ for FirmAE and `Pandawan`, respectively. Furthermore, `Pandawan`, like other re-hosting frameworks, faces specific limitations. Since the *Oracle* targets only the data structures used by open-source kernel modules with a counterpart in UP_{kos}, it does not guarantee that the layout of data structures used by proprietary kernel modules will be unaffected. The options produced in S4 can potentially misalign the layout of data structures used by proprietary kernel modules causing the modules to crash during emulation. Furthermore, as shown in Section 5.3.3 only a fraction of the total modules in the F_{fs} file-systems are loaded during emulation by the firmwares' bootup scripts. Consequently, the `SyscallToKmodTracer` plugin only collects system call information for a subset of user level programs that interact with the loaded kernel modules during emulation. The kernel modules that are not loaded and user level programs that are not executed during the emulation do not contribute to the holistic firmware analysis.

Chapter 6

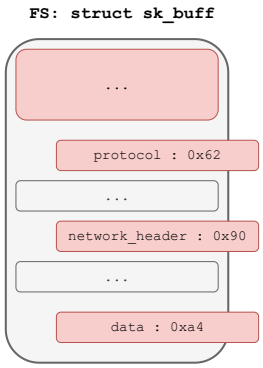
FirmDiff: Improving the Configuration of Linux Kernels Geared Towards Firmware Re-hosting

I this chapter, I present my third contribution to advancing firmware re-hosting techniques. Initially, I elaborate on the motivation behind this work. Next, I discuss, at a high level, my technique for improving current full-system re-hosting approaches that target privileged Linux-based firmware code (in the form of kernel modules and drivers). In addition, I provide in-depth information about `FirmDiff`, the prototype implementation of this technique. Finally, I detail the evaluation of `FirmDiff` on modern firmware images (that use kernels above the version v4.x) to showcase the effectiveness of `FirmDiff` in improving the fidelity of existing full-system re-hosting approaches.

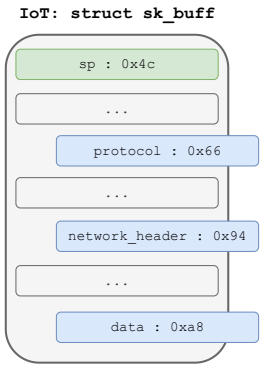
6.1 Motivation

As previously mentioned, FirmSolo produces kernels (i.e., FS_{kern}) whose configuration approximates the configuration of the kernels used by firmware images (i.e., IoT_{kern}). There are cases, though, where there are important discrepancies between the configuration of the FS_{kern} and IoT_{kern}. Specifically, these configuration discrepancies might cause the views of the memory layout of common data structures between the FS_{kern} and the IoT_{kern} and IoT kernel modules (IoT_{kos}) to differ. Consequently, there is a high possibility that misaligned data structure accesses occur when the IoT_{kos} are loaded or executed within the FS_{kern}, further leading to crashes. I provide a motivating example of such a case in Figure 6·1.

In this scenario, the misaligned data structure is `struct sk_buff`. Figures 6·1a



(a) Memory layout of `struct sk_buff` in the FS_{kern}.



(b) Memory layout of `struct sk_buff` in the IoT_{kern}.

```
1  struct sk_buff {
2     struct  sk_buff *next;
3  ...
4  #ifdef CONFIG_XFRM
5     struct  sec_path *sp;
6  #endif
7     unsigned int  len, data_len;
8  ...
9     __be16  protocol;
10 ...
11 }
```

(c) Source code implementation of `struct sk_buff` in the Linux 3.4.103 kernel.

Figure 6·1: Example of the misaligned data structure `struct sk_buff`. Figures 6·1a and 6·1b illustrate the incorrect and correct offsets of the misaligned members of `struct sk_buff` (red and blue shapes, respectively). Here the analyst has to enable the configuration option `CONFIG_XFRM` during FirmSolo's kernel build process to align the members `protocol`, `network_header`, and `data` at the correct offsets.

and 6·1b showcase the memory layout of `struct sk_buff` in the FS_{kern} and IoT_{kos} (and IoT_{kern}), respectively, while Listing 6·1c illustrates the implementation of `struct sk_buff` in the Linux source code. In my example, the members `protocol` and `data` are misaligned by 4 bytes between the FS_{kern} and the IoT_{kern}. The `struct sk_buff` data structure and specifically the two members above are accessed by the `bonding.ko` kernel module (used to combine networking interfaces). However, these members are not accessed during the execution of the kernel module's initialization function. Instead, they are accessed when function `bond_arp_rcv` is executed. As a result the example module does not crash when loaded into the FS_{kern}, only when its code (and specifically function `bond_arp_rcv`) is executed. Thus, FirmSolo cannot automatically detect and fix the misaligned data structure `struct sk_buff`. To correctly align these members, the analyst has to enable the configuration option `CONFIG_XFRM` (see lines 4-6 in Listing 6·1c) which will insert the member `sp`, with a size of 4 bytes, before the two misaligned members, thereby bringing these members to the correct offsets. Unfortunately, as is the case in this exam-

ple, FirmSolo is unable to automatically detect the issue and correct the memory layout of `struct sk_buff`, if the IoT kernel modules do not meet the following requirements:

1. The crashing IoT kernel modules are open-source and their equivalent upstream kernel modules must be compiled by FirmSolo.

2. The crashing IoT kernel modules are not stripped (i.e., the function names used by the kernel modules are available).

3. There is an available crash dump (i.e., a kernel `Oops` message).

4. The crash occurs while the IoT kernel modules are being loaded into the custom kernel (i.e., while executing their `init_module` function).

5. The crash occurs within a known function that belongs to the crashing IoT kernel modules or their dependencies and not within a function of the custom kernel.

6. The misaligned data structures are not modified at source by the vendors (FirmSolo uses the unmodified open source equivalents of the IoT kernels).

In Section 6.2.2, I detail how an analyst can rely on `FirmDiff` to pinpoint the misaligned data structure and its offending members to align the layout of the target data structure.

6.2 Overview

In this Section, I discuss the design of `FirmDiff`, whose goal is to aid analysts in improving FirmSolo's kernel configuration to closely approximate the configuration of the IoT kernels in firmware images. In turn, FirmSolo's kernels can be used to analyze either the user level programs or the kernel modules within these images by preventing errors (i.e., kernel module crashes) from occurring during the analysis. In Section 6.3, I provide detailed

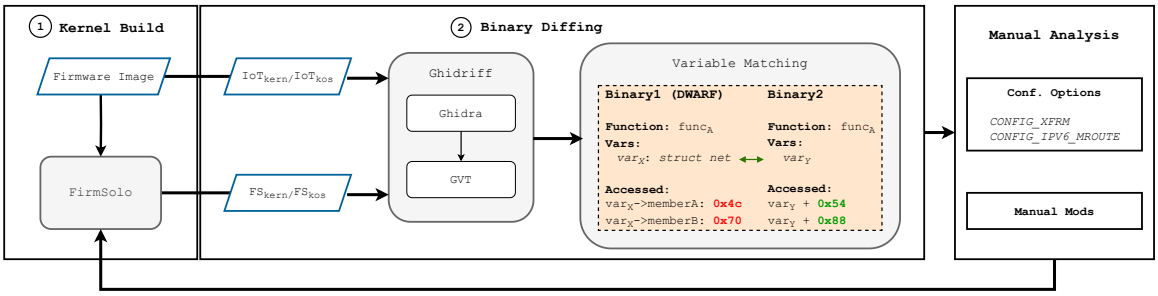


Figure 6·2: The overview of `FirmDiff`. The figure depicts the 2 stages of `FirmDiff`. In the Kernel Build stage, `FirmDiff` consumes a firmware image and analyzes it with FirmSolo. In the Binary Diffing stage, `FirmDiff` diffs the FS_{kos} and IoT_{kos} open-source modules with Ghidriff and matches the variables used within the functions of the compared binaries. The analyst analyzes the information produces by `FirmDiff` to correct the layout of misaligned data structures.

information about how analysts can use `FirmDiff`'s results to improve FirmSolo's kernel configuration and build process.

As illustrated in Figure 6·2, `FirmDiff` consists of two stages; ① The Kernel Build stage where `FirmDiff` uses FirmSolo to build the custom kernel and kernel modules for the target firmware image. ② The Binary Diffing stage, where `FirmDiff` diffs all the open-source kernel modules of a firmware image with their upstream variants built by FirmSolo in stage ①. During the diffing process, `FirmDiff` first matches the functions between the compared kernel modules. Then, for each function match, `FirmDiff` further pairs the variables (with a data structure type) and outputs these variables' memory accesses. Any discrepancies in these memory accesses indicate the existence of misaligned data structures between the IoT and FirmSolo's kernels. By fixing these data structure layout discrepancies, analysts can prevent the IoT kernel modules from crashing when analyzed within FirmSolo's kernels.

6.2.1 Kernel Build

Stage ① is responsible for building a custom kernel that can load and analyze the IoT_{kos} of a firmware image F_{image}. It is an iterative stage since the analyst can invoke the kernel build process multiple times (see Section 6.2.3).

Initially, `FirmDiff` uses FirmSolo's analysis pipeline to build the FS_{kern} and FS_{kos} for F_{image}. The FS_{kos} are kernel modules which have an open-source (or upstream) counterpart in F_{image} and are compiled by default by FirmSolo. It is important that the modules in FS_{kos} and their counterparts in IoT_{kos} are available, so that `FirmDiff` can diff them.

After the diffing process in stage ② concludes, the analyst can supplement the configuration of FirmSolo with additional options and re-invoke stage ① or modify the kernel source (i.e., the layout of data structures) and manually compile the custom kernel. Technically, FirmSolo could be substituted with any other firmware re-hosting framework that uses a custom kernel, such as Firmadyne, FirmAE, Honware or EASIER. However, these systems use pre-built kernels that are not accompanied by open-source kernel modules that have a counterpart in F_{image}. Thus, for `FirmDiff` to function, these systems would have to be modified to also produce open-source kernel modules with a counterpart in IoT_{kern}. FirmSolo is the only system which for every firmware image automatically produces the kernel modules (FS_{kos}) with IoT_{kos} counterparts. This justifies my decision of using FirmSolo as one of the foundation blocks of `FirmDiff`.

6.2.2 Binary Diffing

In stage ②, `FirmDiff` compares the modules in FS_{kos} with their counterparts in IoT_{kos}. This stage aims to reveal discrepancies in the layouts of data structures between the modules in FS_{kos} (and FS_{kern}) and the modules in IoT_{kos}.

The kernel data structures represent shared data, with specific semantics (i.e., the memory layout), between the kernel and the kernel modules. Both parties that use these data

(i.e., the kernel and kernel modules) must agree on these semantics, so that all data accesses are correct. Data structures that are defined in and used only by the kernel, of course, do not conform to this rule. The memory layouts of data structures can be affected by two aspects; 1) The configuration options in the kernel configuration, and 2) Any modifications implemented by the vendors in their kernels' source code. To make sure that both the FS_{kern} and IoT_{kos} agree about the layout of their common data structures, it is important to first identify these common data structures and then detect which configuration options or vendor modifications affect their memory layout. As stated in Section 6.1, FirmSolo's memory layout alignment algorithm needs to meet certain conditions and thus cannot always identify these data structures and find the options or modifications that affect their layout.

In contrast, `FirmDiff` only requires that either the crashing modules in IoT_{kos} are open source or there exist other open source modules in IoT_{kos} that use the same (or a subset) of data structures that the crashing modules use. By default though, `FirmDiff` uses Ghidriff to diff all the open-source modules in IoT_{kos} with their variants in FS_{kos} produced in stage ①. Furthermore, unlike FirmSolo's memory layout alignment algorithm which is limited by FirmSolo's dynamic nature (see requirement 4 in Section 6.1), `FirmDiff`'s static analysis can detect data structure misalignments throughout the analyzed kernel modules. I discuss how `FirmDiff`'s analysis works below.

For each compared pair of kernel module binaries, `FirmDiff` follows these steps: 1) It analyzes both binaries with Ghidra. 2) It runs GVT on the two binaries and matches their functions. 3) Finally, for each function match `FirmDiff` invokes the variable matching algorithm. This algorithm correlates the variables with a data structure type (e.g., `struct net`) between the matched functions. It leverages the DWARF information embedded within the FS_{kos} and FS_{kern} during their compilation, to detect the data structure types of the variables. `FirmDiff` disregards variables with a primitive type (e.g., int) since their

memory layout is static and not affected by any configuration options in the kernel. Once the variables with a data structure type are paired (between the diffed binaries), `FirmDiff` proceeds to extract the memory accesses for each variable in a pair. These memory accesses correspond to data structure members accessed (i.e., their offsets from the base of the data structure). `FirmDiff` displays the member accesses (member names and their offsets) for each variable pair side by side, enabling a quick and easy comparison. If the offsets do not match, the memory layout of the target data structure (i.e., the type of the variable pair under test) is misaligned between the modules in FS_{kos} and IoT_{kos}. After the variable matching algorithm concludes, `FirmDiff` stores all the information produced (i.e., the variable pairs and their member accesses for each function) within the `JSON` and markdown files produced by Ghidriff.

6.2.3 Manual Analysis

After the information produced by `FirmDiff` is available, the analyst can parse that information to extract the misaligned data structures highlighted by `FirmDiff`. Afterwards, by using other tools available (e.g., (Carvalho, 2009; NSA Research Directorate, 2025; GNU Project, 2024; Bell Labs, 2012)) the analyst can pinpoint the root cause behind the data structure misalignments. Specifically, for each variable pair (with a data structure type) the analyst can infer if and which members are shifted and by how many bytes. Based on the number of bytes shifted, they can in turn detect which configuration options need to be toggled in FirmSolo's kernel configuration process or manually modify the data structures' source code to align these members between the compared binaries. To infer which options need to be toggled, the analyst can consult FirmSolo's built-in utility which provides a mapping for each data structure (in the FS_{kern} kernel) and the options that affect its layout. The outcome of this analysis is a refined FS_{kern} that agrees with the IoT_{kos} about the layout of the data structures fixed by the analyst. The IoT_{kos} should not crash when correctly accessing the members of the aligned data structures while being analyzed within

the refined FS_{kern}.

6.3 Implementation

In this Section, I provide the implementation details behind `FirmDiff`.

6.3.1 Kernel Build

In the first iteration of stage ①, `FirmDiff` consumes a firmware image F_{image} as input and analyzes it with FirmSolo. In turn, FirmSolo produces the FS_{kern} and FS_{kos} for F_{image}. I note that it is not necessary to execute FirmSolo's entire analysis pipeline, as `FirmDiff` only requires FS_{kern} and FS_{kos}. Thus, `FirmDiff` can be instructed to run FirmSolo until the kernel build step and omit the resource-intensive emulation and data structure layout recovery steps. Next, `FirmDiff` provides the FS_{kos} and IoT_{kos} to stage ②.

Stage ① is iterative, the analyst can re-invoke it after they analyze the information produced by stage ②. Specifically, the analyst can either modify the configuration of the FS_{kern} by adding or removing configuration options during FirmSolo's kernel configuration process or they can manually modify the kernel source, change the configuration and re-compile the FS_{kern} and its FS_{kos} (see Section 6.3.3).

6.3.2 Binary Diffing

In stage ②, `FirmDiff` diffs every open-source kernel module in the IoT_{kos} with its counterpart in FS_{kos}, using Ghidriff. By default, Ghidriff analyzes the binaries with both Ghidra's main and GVT tools to find which functions match between the diffed binaries. To realize `FirmDiff`, I modify Ghidriff to also match variables used by the same functions (found by GVT) in the diffed binaries, and detect the memory accesses for these variables (i.e., the variable matching algorithm explained below). The information about the matched variables and their memory accesses helps the analyst align the memory layouts of misaligned

data structures between the modules in FS_{kern} and IoT_{kos}. Below, I detail the analysis steps followed in this stage.

Ghidra Analysis. First, every binary pair is analyzed with Ghidra's main tool. I note that the FS_{kern} and the modules in FS_{kos} contain debugging information (i.e., DWARF) which I use later during the variable matching process. Next every pair of binaries is analyzed with Ghidra's Version Tracking (GVT) tool. GVT uses 15 different correlators (i.e. heuristics), to match the functions between the compared binaries.

Once the function matching process concludes, `FirmDiff` initiates its variable matching algorithm for each found pair of functions between the compared binaries. For simplicity, for any two binaries diffed (or compared) I refer to the binary originating from FirmSolo as B_{FS} and the binary originating from F_{image} as B_{IoT}. Similarly, I refer as F_{FS} and F_{IoT} the functions matched between the B_{FS} and the B_{IoT}.

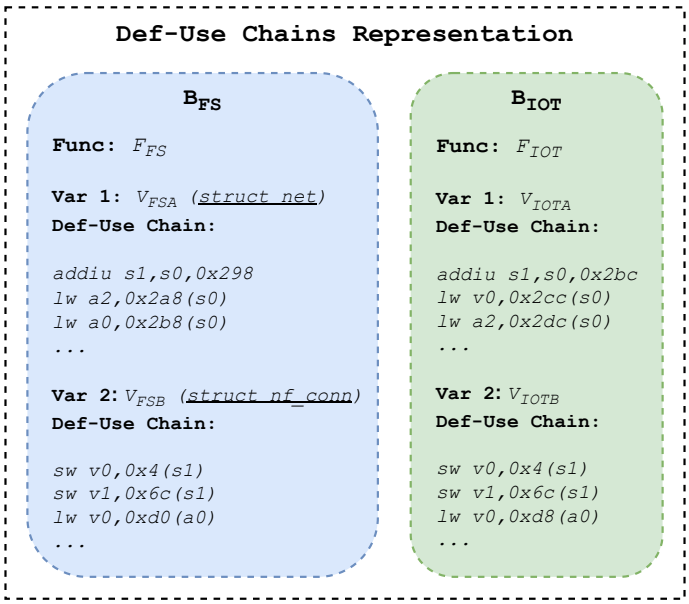


Figure 6·3: Example of the Def-Use chains inference for variables V_{FSA} and V_{FSB} (accessed by function F_{FS}) and variables V_{IOTA} and V_{IOTB} (accessed by function F_{IOT}).

Variable Matching. `FirmDiff` first lifts F_{FS} and F_{IoT} to Ghidra's PcodeOp Intermediate Representation (IR). By using the IR of the functions, `FirmDiff` extracts the Def-Use

chains of all the variables within F_{FS} and F_{IoT}. The Def-Use chains are represented as the assembly instructions (`FirmDiff` maps PcodeOp back to MIPS or ARM assembly) that access each variable. In turn, these assembly instructions reveal how each function accesses the memory of the corresponding variable (i.e., the corresponding data structures' members). I illustrate how `FirmDiff` represents the Def-Use chains in Figure 6·3. In this example, functions F_{FS} and F_{IoT} access the variables V_{FSA}, V_{FSB}, and V_{IoTA}, V_{IoTB}, respectively. Since B_{FS} has debugging information available, the types (and accessed members) of V_{FSA} and V_{FSB} are known; `struct net` and `struct nf_conn`, respectively. `FirmDiff` will propagate this information to variables V_{IoTA} and V_{IoTB}, during the variable pairing step discussed below. After the variable Def-Use chains are available, `FirmDiff` invokes the variable pairing step.

Variable Pairing. The variable pairing step borrows its logic from the layout alignment algorithm that FirmSolo uses. I present the inner workings of variable pairing in Figure 6·4. First, `FirmDiff` maps the Def-Use chains for the variables V_{FSA}, V_{FSB}, V_{IoTA} and V_{IoTB} into the vectors C_{FSA}, C_{FSB}, C_{IoTA} and C_{IoTB}, respectively. Specifically, each ISA mnemonic (e.g., `sw`) is mapped to a coordinate in the vector. The value of each coordinate is equal to the number of times the corresponding mnemonic occurs in the Def-Use chain of the target variable. In addition, I set the last element of each vector as the total number of instructions accessing the corresponding variable to make the vector similarity measurement discussed below more accurate. Specifically, to match the variables V_{FSA} and V_{FSB} with the variables V_{IoTA} and V_{IoTB}, `FirmDiff` uses the cosine similarity measurement (Wikipedia, 2024) between C_{FSA}, C_{FSB} and C_{IoTA}, C_{IoTB}. The variables with the highest similarity score are considered a pair. In my example, variable V_{FSA} is paired with V_{IoTA} and variable V_{FSB} is paired with V_{IoTB}. Furthermore there can only be one match between the variables in F_{FS} and F_{IOT}, to not confuse the analyst during the manual analysis process.

Once the variable matches are available, `FirmDiff` transfers the types from the vari-

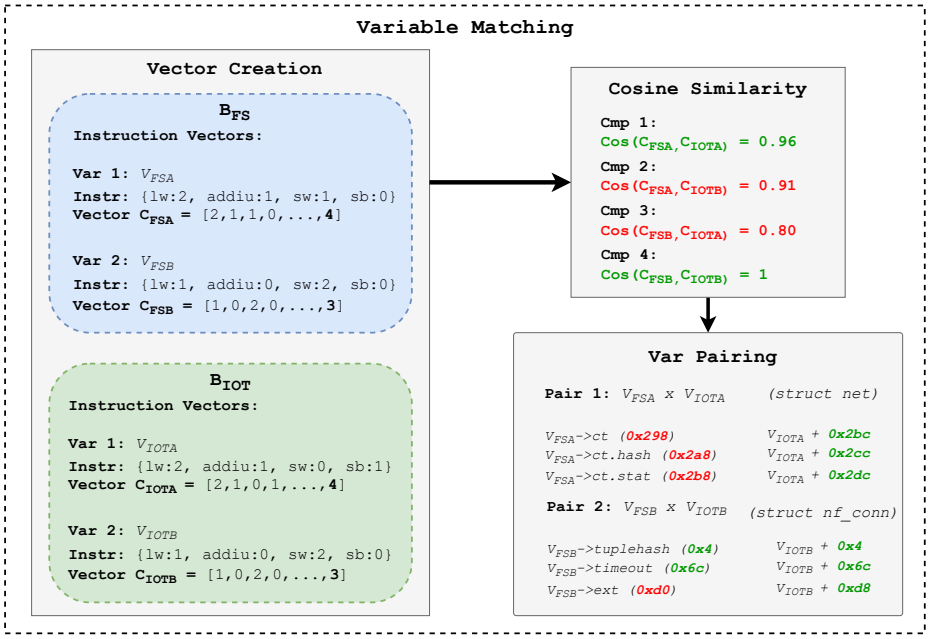


Figure 6·4: The variable matching algorithm. First, `FirmDiff` creates a vector where each coordinate corresponds to an ISA mnemonic and each value corresponds to the times the mnemonic occurs in the Def-Use chain. In this case vectors C_{FSA}, C_{FSB}, C_{IOTA} and C_{IOTB} correspond to variables V_{FSA}, V_{FSB}, V_{IOTA}, and V_{IOTB}, respectively. By using the cosine similarity measurement, `FirmDiff` pairs the variables V_{FSA} and V_{IOTA} (of type `struct net`) and the variables V_{FSB} and V_{IOTB} (of type `struct nf_conn`). Finally, `FirmDiff` displays the members and offsets of the accessed members of each variable.

ables in F_{FS} to their counterpart in F_{IOT}. If the variable has a primitive type (e.g., `int`), `FirmDiff` discards the pair since its layout is always static and cannot be modified manually or by toggling configuration options (i.e., it cannot be incorrectly accessed). Thus, `FirmDiff` keeps only the variable pairs with a data structure type, since their layout can be modified and potentially be misaligned between B_{FS} and the B_{IoT}. For these variable pairs, `FirmDiff` outputs their type and their accessed members (i.e., names and offsets) in ascending order by offset. The information about which members are accessed and their offsets is embedded within the Def-Use chains and DWARF information extracted previously. I note here that `FirmDiff` only knows the names of the data structure members

accessed for the variables in F_{FS}, due to the availability of the DWARF information in B_{FS}. For variables that belong in F_{IOT} (since B_{IOT} is usually stripped), `FirmDiff` displays only the offsets of the accessed members. Once the variable matching algorithm concludes, `FirmDiff` stores the variable pairs and their accesses for each matched function within a `JSON` and a markdown file. Next, these findings can be parsed and/or processed programatically by the analyst.

6.3.3 Manual Analysis

To identify any potential misaligned data structures between the FS_{kern} and the modules in IoT_{kos} (and the IoT_{kern}), the analyst can process the data structure type variable pairings and their memory accesses for the matched functions in B_{FS} and B_{IOT}. Specifically, misaligned data structures are recognizable based on the differences (or shifts) in the offsets of their accessed members within two matched functions. As `FirmDiff` does not provide further information about the root cause behind these shifts, the analyst leverage other complementary static and dynamic analysis tools (e.g., cscope (Bell Labs, 2012), pahole (Carvalho, 2009), Ghidra (NSA Research Directorate, 2025) and GDB (GNU Project, 2024)) to find the reason behind the misalignments. As mentioned in Section 6.2.2 the layout of data structures is affected by the configuration options in the kernel configuration and any modifications applied by vendors in their kernels' source code. FirmSolo provides a builtin utility that maps each data structure in the FS_{kern} to the configuration options that affect the layout of the data structure. The analyst can first consult this utility to explore possible alignment solutions requiring only toggling configuration options. When no effective solution is achieved, it is an indication that the IoT kernel source code has been modified by the vendors (see Section 6.4.3). Thus, the analyst can either modify FS_{kern}'s source code (i.e., pad the layout of data structures) and/or its configuration to correctly align the layout of the misaligned data structures and re-invoke stage ① to build the refined FS_{kern} kernel. If the refined FS_{kern} kernel and IoT_{kos} agree about the layout of their common data structures

then the IoT_{kos} should not crash when accessing the correctly aligned members of these data structures (while being loaded or executed within the FS_{kern}).

6.4 Evaluation

In this Section, I evaluate `FirmDiff`'s effectiveness in firmware re-hosting. In particular, I provide the answers to the following research questions:

RQ1 How efficient is `FirmDiff` in highlighting misaligned data structures in the FS_{kern} and FS_{kos} (§ 6.4.2)?

RQ2 How effective is `FirmDiff` in improving the kernel configuration of the FS_{kern} (§ 6.4.3)?

RQ3 Can `FirmDiff` improve FirmSolo's firmware re-hosting (§ 6.4.4)?

First, I discuss my dataset and next detail my experimental analysis to answer the research questions above.

6.4.1 Dataset

I evaluate `FirmDiff` on a set of 10 firmware images with 7 of the images originating from the FirmSolo dataset. Specifically, I randomly select the set from the images that have persistent kernel module crashes in FirmSolo, even after applying its memory layout alignment algorithm. I aim to illustrate that `FirmDiff` is capable of aiding the analyst in correctly aligning the memory layout of the misaligned data structures in these cases. I also evaluate `FirmDiff` on 3 randomly picked firmware images from the Greenhouse (Tay et al., 2023) dataset which was published in 2023. I choose a set of images that FirmSolo can successfully emulate and load their kernel modules, but is unable to address their kernel module crashes. The reason for this additional set of images is to showcase that `FirmDiff`'s analysis is also applicable to modern firmware images. The images in my dataset belong to 6

Image #	Vendor	Kernel	Arch	CS KOs	CK KOs	MA DS
1	TPLink	2.6.21.5	MIPS	4	2	2
2	Trendnet	2.6.31	MIPS	1	2	4
3	DLink	2.6.33.2	MIPS	2	2*	3
4	Zyxel	2.6.36	MIPS	4	2	2
5	ASUS	2.6.36	MIPS	4	2	2
6	TPLink	3.0.21	ARM	1	2	7
7	Netgear	3.4.103	ARM	6	1	3
8	TPLink	4.1.52	ARM	4	1	5
9	Netgear	4.4.60	ARM	11	1	5
10	ASUS	4.4.198	MIPS	111	4	4
			Total	**148**	**18**	**37**

Table 6.1: Statistics about the firmware images in our dataset. The table depicts the identifier of each firmware image, the vendor, the kernel version, the architecture, the number of kernel modules crashed (CS) during FirmSolo's re-hosting, the number of kernel modules I manually checked (CK) for each image and finally the number of misaligned (MA) data structures (DS) detected with `FirmDiff`. The * indicates that I also analyzed the FS_{kern} along with IoT_{kern} for the specific image. With the blue color and green colors I represent the firmware images originating from the FirmSolo and Greenhouse datasets, respectively.

vendors and their kernels range from version `2.6.21.5` to `4.4.198`. I provide the relevant information in Table 6.1.

6.4.2 Misaligned Data Structures

In this Section, I evaluate the accuracy of `FirmDiff`'s variable matching algorithm and also compare its efficiency with FirmSolo's misaligned data structure discovery capabilities, to answer RQ1.

Overall, `FirmDiff` analyzes 842 open source kernel modules (i.e., open source kernel modules in IoT_{kos} and their counterparts in FS_{kos}). During this analysis, `FirmDiff` matches 29,485 functions and pairs 28,269 variables (with a data structure type). Out of the total functions matched, 20,564 (70%) functions actually contain zero variable pairs with a data structure type. `FirmDiff`'s analysis takes 17 minutes on average. Due to the volume

of the results, I manually verify FirmDiff's findings for a subset of 18 crashing open source kernel modules and 1 kernel binary and confirm the existence of 37 misaligned data structures. I detail how I proceed to correct the layout of these data structures in Section 6.4.3.

Variable Matching Accuracy. To measure the accuracy of the variable matching algorithm, I randomly choose one kernel module per image (i.e., 10 kernel modules) out of the 18 kernel modules I analyzed. Next, I sample 10 functions per kernel module (100 functions in total) and verify if FirmDiff correctly matches (or pairs) the variables in the sampled functions. In particular, I confirm that FirmDiff pairs 392 variables in total. Out of these pairs, 233 (59%) are correct pairs while 159 (41%) are not. I observe that the incorrect pairs are most prevalent between variables accessed by a single instruction, which tend to confuse the variable matching algorithm. Nevertheless, a data structure can be referenced multiple times (through distinct variables) in different functions in a kernel module. Thus, the analyst can extract sufficient information to align the layout of the data structure by consulting only the correct variable pairs (corresponding to the data structure), while discarding the incorrect pairs.

Kernel Analysis. During my experiments, I opt to not analyze the FS_{kern} and IoT_{kern} kernels for these images with FirmDiff. Specifically, the analysis takes hours, due to the thousands of functions and variables that are matched between these binaries. In addition, having the IoT_{kern} available is not guaranteed. In this case, I am able to extract the IoT_{kern} for only 2 images in my dataset, thus I emphasize the analysis of kernel modules.

Image 3 is the only case where I proceed to analyze the FS_{kern} and IoT_{kern} kernels with FirmDiff. In particular, the image contains two crashing kernel modules, one open source (nf_conntrack_prot_gre.ko) and one closed source (jcp.ko [1]). The jcp.ko kernel module implements a proprietary protocol that exposes USB devices over the net-

[1] https://www.silextechnology.com/connectivity-solutions/device-connectivity/sx-virtual-usb-sdk-for-linux

work. Since the `nf_conntrack_prot_gre.ko` exists in both the FS_{kos} and IoT_{kos}, both versions are analyzed by `FirmDiff`. However, `jcp.ko` does not have an upstream counterpart compiled by FirmSolo. Thus, I cannot analyze `jcp.ko` with `FirmDiff` to identify potential misaligned data structures between this kernel module and the FS_{kern} kernel. Unfortunately, `FirmDiff`'s analysis on `nf_conntrack_prot_gre.ko` is ineffective and does not provide information about the misaligned data accessed by this kernel module. For these reasons, I also proceed to analyze the IoT_{kern} and FS_{kern} for image 3 since the kernels have information about all the data structures that are shared between the kernel modules (open and closed source) and the core kernel. I aim to detect the misaligned data structures accessed by both crashing kernel modules in this case. I note though, that the entire analysis takes over 5 hours, hence it is not scalable for a large number of firmware images. In addition, compared to the hundreds of functions matched within the kernel modules, in this case there are 14,066 functions matched between the IoT_{kern} and FS_{kern} kernels. Analyzing all the function matches for potential misaligned data structures requires extensive effort and time. For this reason, I inspect only the first 200 functions matched between IoT_{kern} and FS_{kern} for image 3 and identify 4 misaligned data structures.

`FirmDiff` vs. FirmSolo. In general, `FirmDiff` outperforms FirmSolo by highlighting 16 unique misaligned data structures during my experiments, while FirmSolo only detects 4 with its layout alignment mechanism. I provide a detailed breakdown of the identified data structures for both `FirmDiff` and FirmSolo in Table 6.2. Thus, `FirmDiff`'s high efficiency in highlighting misaligned data structures, also justifies `FirmDiff`'s utility as a complementary analysis to FirmSolo.

6.4.3 Kernel Configuration

Identifying misaligned data structures is only the first step. To improve firmware re-hosting and analysis, the analyst needs to address these misalignments. In this Section, I detail how `FirmDiff`'s analysis aids the analyst to improve the configuration of FirmSolo's kernels,

to answer RQ2.

Data Struct.	# Seen	Opts Only	Mods Only	Both
FirmDiff				
net	7	7	0	0
sk_buff	7	3	2	2
nf_conn	5	1	3	1
net_device	5	2	1	2
wiphy	2	0	2	0
cfg80211_registered_device	1	0	0	1
ip_tunnel	1	1	0	0
inet6_ifaddr	1	1	0	0
inet6_dev	1	1	0	0
ipv6_pinfo	1	1	0	0
cfg80211_scan_request	1	0	1	0
wireless_dev	1	1	0	0
input_dev	1	1	0	0
inode	1	1	0	0
neighbour	1	0	0	1
bonding	1	0	1	0
Total	**37 (35)**	**20**	**10**	**7**(5)
FirmSolo				
struct module	8	8	0	0
struct net	2	2	0	0
struct net_device	1	1	0	0
struct kmem_cache_alloc	1	1	0	0
Total	**12 (4)**	**12**	**0**	**0**

Table 6.2: The misaligned data structures identified using FirmDiff and FirmSolo and the types of alignments applied for each data structure (i.e., configuration options only, modifications only, or both). FirmSolo can only apply alignments that require toggling configuration options. The parentheses in the total values indicate how many of these data structures are successfully aligned when using each system.

For this experiment, I further analyze the 37 misaligned data structures highlighted by FirmDiff (see Section 6.4.2), using complementary source code and binary introspection tools (Carvalho, 2009; NSA Research Directorate, 2025; GNU Project, 2024; Bell Labs, 2012). I successfully align 35 out of these 37 data structures in the corresponding FS_{kern} kernels, by either toggling configuration options in the kernel, directly modifying the source code of the data structures or both. I note that I am able to identify the combination of configuration options and modifications that align the layout of these data structures through trial and error. Of course human expertise and knowledge also plays a crucial role

in this case. Specifically, I am able to correct the layout of 20 data structures by simply toggling configuration options in the kernels, 10 data structures by directly modifying their implementation in the kernel source code, and 5 data structures by both toggling options and modifying their source code. In contrast, FirmSolo's memory alignment algorithm is only able to correct the layout of 4 misaligned data structures (see Table 6.2). Unfortunately, I am unable to align the layout of 2 data structures (`struct net` and `net_device`) for a single image (i.e, image 10). In particular, I cannot guess the correct combination of modifications and configuration options that align the layout of these data structures. As I discuss below, extensive modifications by the vendors in their kernel source code can impede the analyst's layout alignment process.

```
1 struct net_device {
2   char  name[IFNAMSIZ];
3 ...
4   const struct header_ops *header_ops;
5   unsigned int  flags;
6 ...
7 }
```

Figure 6·5: Implementation of `struct net_device` in the open source Linux-4.4.60.

```
1  struct net_device {
2    char  name[IFNAMSIZ];
3  ...
4  #ifdef CONFIG_ETHERNET_PACKET_MANGLE
5    void (*eth_mangle_rx)(...);
6    struct sk_buff *(*eth_mangle_tx)(...);
7  #endif
8    const struct header_ops *header_ops;
9    unsigned int  flags;
10 ...
11 }
```

Figure 6·6: Implementation of `struct net_device` in the IoT Linux-4.4.60.

Figure 6·7: Code diff of the source implementation of `struct net_device` between the upstream and IoT version of Linux-4.4.60.

Vendor Modifications. The majority of the misaligned data structures I identify in the three modern images (12 of 15 data structures), are modified in their source code by the vendors. Listing 6·7 provides the code diff of the implementation of `struct net_device` between the upstream version of the kernel and the modified kernel used by Netgear (I downloaded the GPL code provided by Netgear for image 9). The `struct net_device` is the core representation of network devices in the kernel, containing information about the network adapters and the configuration of the network devices. In this example, I show-

case a small portion of the modifications implemented by Netgear in their kernel source code. Specifically, the modified kernel contains additional configuration options (e.g., `CONFIG_ETHERNET_PACKET_MANGLE`) that guard members of `struct net_device` that are not present in the upstream kernel (see line 3 in Figure 6·5 and lines 4-7 in Figure 6·6). As the name of the newly added configuration option and members (`eth_mangle_rx` and `eth_mangle_tx`) suggest, this functionality is relevant to mangling ethernet packets. I note here that without consulting the kernel source code used by the vendors, it is not possible to guarantee that my alignment solutions are "correct". In particular, I might not use the same configuration options or place the modifications in the exact places as the vendors. Unfortunately, I am only able to retrieve the GPL source code of images 8 and 9. Of course, I only use the GPL source code to verify the code modifications in the vendors' kernels and do not rely on the source code for the analysis in any form. Nevertheless, as I discuss next, the majority of my solutions are effective in improving firmware re-hosting.

6.4.4 Firmware Re-hosting

In this Section, I discuss how `FirmDiff` improves FirmSolo's firmware re-hosting, to answer RQ3. I aim to prevent IoT kernel module crashes that are related to misaligned accesses (specifically regarding the misaligned data structures I identified) during FirmSolo's emulation.

In this experiment, I re-host my firmware images with FirmSolo, using the refined custom kernels and attempt to load the 148 kernel modules that originally crash while being loaded in the FS_{kern} kernels. Next, I observe which of these kernel modules can load successfully in the refined kernels. If successfully loaded, these kernel modules can be in turn analyzed with FirmSolo's downstream analysis. In particular, during my experiments 28 kernel modules load successfully due to my layout alignment solutions implemented with the help of `FirmDiff`. In contrast, FirmSolo is only able to address the crashes for and successfully load 5 kernel modules with its automated layout alignment algorithm.

Failed Cases. The majority of the kernel modules (114 out of 120) continue to crash due to lingering misaligned data structures. Most of these kernel modules (111 out of 114) belong to image 10. As I mention in Section 6.4.3, I am unable to fix all the misaligned data structures in this case. However, upon further inspection of the crashes, I observe that they occur within the `load_module` function (responsible for loading the kernel modules into the kernel). Based on this observation, I confirm with Ghidra that the `struct module` data structure is misaligned (not highlighted by `FirmDiff`) and incorrectly accessed by the `load_module` function. Unfortunately, I cannot find a working alignment solution for `struct module`, which might significantly reduce the number of the crashing modules in image 10. For the remaining 3 kernel modules, my layout alignment solutions did not address the crashes. Note that even if all data structures are correctly aligned, other errors can still lead to module crashes. For example, unrelated (to the layouts of data structures) accesses to MMIO regions can be the root cause for such issues. Furthermore, 5 out of the 120 crashing kernel modules, crash with a *Kernel bug detected* bug type that is not related to misaligned data structure accesses (as opposed to *Unable to handle kernel paging request at*). These types of bugs indicate that the kernel module code reached a check that failed and resulted in a kernel trap. FirmSolo's authors also confirmed the existence of these bugs and observed that vendors had modified the checks in their own kernels to prevent the code execution from reaching the traps. I do not consider these types of bugs as a limitation of `FirmDiff`, as they are not related to misaligned data structures. Finally, 1 kernel module (`jcp.ko`) crashes when accessing the misaligned `kmem_cache` data structure. FirmSolo, by default, is able to detect and correct misalignments in this data structure. However, the crash has to occur within the `kmem_cache_alloc` function, which is not the case with `jcp.ko`. My analysis with Ghidra, shows that the crash occurs after the `kmem_cache_alloc` function is executed. Similar to FirmSolo, `FirmDiff`'s analysis does not detect the misalignment of `kmem_cache`, and thus I do not address this crash.

FirmSolo's re-hosting experiments only showcase if and how many kernel modules are successfully loaded in the refined FS_{kern} kernels. To further illustrate the utility of `FirmDiff` in firmware re-hosting, I check if the "fixed" kernel modules crash while their code is executed during emulation. For this purpose, I use the FirmSolo-compatible Firmadyne system to emulate the firmware images in my dataset. All the experiments run without errors except one case (i.e., image 7) where a crash occurs due to a kernel bug. Again the crash is not related to a misaligned data structure and thus it is not considered a limitation of `FirmDiff`.

6.5 Discussion

In this Section, I discuss future applications, the limitations of `FirmDiff` and remaining gaps in the research landscape to analyze Linux-based IoT firmware kernel modules. Even though its main target is IoT privileged firmware code, `FirmDiff` can also be applied to the same targets as Ghidriff, i.e., general purpose OS binaries. Since `FirmDiff` relies on Ghidra and Ghidriff, its binary analysis is OS agnostic. For example, I can use `FirmDiff` to analyze patched kernel modules targeting general purpose OS machines. As previously mentioned, `FirmDiff`'s goal is to expose misaligned data structures in FirmSolo's kernels, which can lead to kernel module crashes, if the IoT kernel modules access these data structures during the emulation. Currently, the analyst has to manually align the layout of these offending data structures, through toggling configuration options during the kernel build or by directly modifying these data structures. Similar to FirmSolo, the alignment process can be (partially) automated to alleviate some of the burden of the analysis from the analyst. I leave this as future work.

`FirmDiff` like other static analysis systems has limitations. First, `FirmDiff`'s analysis depends tightly on the accuracy of Ghidra. If Ghidra incorrectly analyzes the target binaries (e.g., incorrectly disassembles the code of functions), `FirmDiff`'s analysis will in turn be

ineffective. Second, unlike FirmSolo, `FirmDiff` cannot pinpoint the origin of the crashes within the kernel modules. `FirmDiff`, only provides information about the layout of data structures used by the crashing kernel modules. The analyst is responsible for identifying if the actual root cause behind the crash is a misaligned data structure access or another type of bug.

Chapter 7

Lock ‘n Load: Detecting Deadlocks in Linux-based Binary-only Firmware

In this chapter, I present my contribution to improving the scope of current dynamic firmware analysis techniques, to deadlock bug detection. First, I provide information about the design my deadlock detection technique on privileged binary-only IoT firmware code (i.e., kernel modules and drivers). Next, I discuss the implementation of LL, the prototype implementation of this technique. Finally, I detail the evaluation of LL's capabilities to detect potential deadlocks on binary-only IoT kernel modules.

7.1 Overview

In this section, I discuss the system overview of LL. Given a target firmware, the goal of LL is to produce a Linux kernel capable of analyzing the firmware's binary-only kernel modules with Lockdep.

To achieve this goal LL has to address three challenges related to the functionality of Lockdep. Given a specific kernel version, the first challenge that LL has to address is to decouple the lockdep metadata in the kernel from Lockdep. Specifically, LL has to find (Stage ①) and modify the code that uses the lockdep metadata to use the *SLoMs* instead (Stage ②). Second, Lockdep's deadlock detection depends inherently on code coverage. LL, relies on the syzkaller kernel fuzzer and a custom network traffic generator (NTG) to exercise the code the binary-only IoT kernel modules, while the modules are analyzed by Lockdep. I stress that LL's instrumentation is fully-automated to account for the code

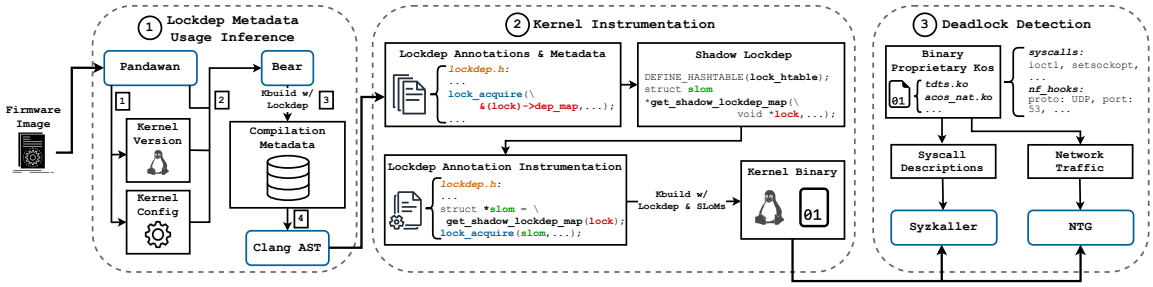


Figure 7·1: System overview of LL. The dashed boxes represent the stages that comprise LL. NTG stands for Network Traffic Generator.

changes regarding locks between different kernel versions. For example, new locks might be introduced or existing ones might be altered, removed, or get moved around. Next, I provide information about the stages comprising LL.

As depicted in Figure 7·1, LL consists of three fully-automated stages: ① The Lockdep Metadata Usage Inference stage, in which LL identifies all the locations in the target kernel's source code where lockdep metadata are used. ② The Kernel Instrumentation stage where LL produces a kernel capable of analyzing binary IoT kernel modules for deadlocks. First, LL inserts the *Shadow Lockdep* subsystem and the *Shadow Lockdep Maps* (*SLoMs*) in the target kernel's source code. Next, LL uses the information from stage ① to instrument the Lockdep subsystem to use the *SLoMs* instead of the original lockdep metadata. The *Shadow Lockdep* subsystem supplements Lockdep with the ability to analyze binary-only IoT kernel modules despite the absence of their source code. Once the kernel instrumentation is complete, LL uses the target kernel's modified source code and the Pandawan re-hosting framework to build a custom kernel capable of re-hosting and analyzing the target firmware's binary only kernel modules with Lockdep. ③ The Deadlock Detection stage where LL first uses the custom kernel produced in stage ② to re-host the binary-only kernel modules in the target firmware. Next, LL dynamically analyzes the re-hosted binary-only kernel modules (via syzkaller and a network traffic generator), using the modified version of Lockdep.

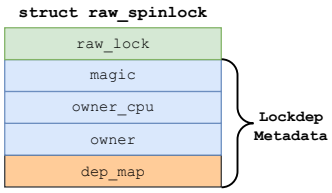
For the sake of simplicity, in the remaining sections I will focus on the kernel locking data structures (e.g., `struct raw_spinlock`, `struct mutex`, `struct semaphore`, etc.), which contain Lockdep's metadata. Of course, my Lockdep instrumentation (see Section 7.1.3) also applies to other non-locking data structures (e.g., `struct timer_list` and `struct kernfs_node`) also traced by Lockdep. I also use the term *custom* kernel and *IoT* kernel to distinguish between the kernel produced by `LL`, and the kernel distributed by the target firmware, respectively. Throughout the rest of this section I first provide information about the *SLoMs* and next I will detail the three stages comprising `LL`.

7.1.1 Shadow Lockdep Maps (*SLoMs*)

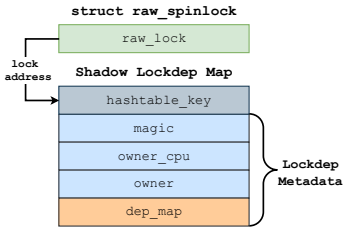
By design, Lockdep's deadlock detection logic primarily depends on two requirements; 1) the lockdep metadata (e.g., the lockdep maps), and 2) the lockdep annotations. First, the locking data structures must contain the lockdep metadata as part of their memory layout. In particular, if Lockdep is enabled during the kernel build (Kbuild) process, the compiler will insert the lockdep metadata in the kernel locking data structures. The lockdep metadata are guarded by Lockdep-specific and interdependent configuration options, such as `CONFIG_DEBUG_LOCK_ALLOC`, `CONFIG_LOCKDEP`, and `CONFIG_PROVE_LOCKING`. Figure 7·2a provides an example of how the `CONFIG_DEBUG_LOCK_ALLOC` configuration option (see lines 7-9) guards the member *dep_map* of type `struct lockdep_map` within the `struct raw_spinlock` data structure. Furthermore, Lockdep also incorporates a variety of debugging checks to verify the correct usage of locking data structures (e.g., the ownership of the lock). The metadata (e.g., `magic`, `owner_cpu`, and `owner`) related to these debugging operations can be seen in the example in lines 3-6. The corresponding option that guards these metadata in the example, `CONFIG_DEBUG_SPINLOCK` will be force-selected if `CONFIG_DEBUG_LOCK_ALLOC` is enabled during Kbuild. I also consider these debugging primitives to fall under the lockdep metadata umbrella and thus handle them appropriately.

```
1  typedef struct raw_spinlock {
2    arch_spinlock_t raw_lock;
3  #ifdef CONFIG_DEBUG_SPINLOCK
4    unsigned int magic, owner_cpu;
5    void *owner;
6  #endif
7  #ifdef CONFIG_DEBUG_LOCK_ALLOC
8    struct lockdep_map dep_map;
9  #endif
10 } raw_spinlock_t;
```

(a) Code implementation of `struct raw_spinlock` in Linux v5.4.164



(b) The `struct raw_spinlock` memory layout with stock Lockdep.



(c) The `struct raw_spinlock` memory layout with Shadow Lockdep.

Figure 7·2: Code implementation and memory layout of `struct raw_spinlock`, when stock Lockdep and Shadow Lockdep are enabled, respectively. The light blue and orange boxes represent the lockdep metadata.

The second requirement of Lockdep is the annotation (i.e., triggering a call to Lockdep) of all the locking operations in the kernel code (e.g., the `__raw_spin_lock` and `__raw_spin_unlock` functions). These annotations initiate Lockdep's deadlock analysis for a specific lock class, using the lock's lockdep metadata. Within the kernel source code, the annotations are implemented as macros (e.g., `spin_acquire` and `spin_release`). These macros expand to Lockdep's core functions (i.e., `lock_acquire` and `lock_release`), if Lockdep is enabled, and to no-code if Lockdep is disabled during compilation, like with the IoT kernels and modules. In turn, these functions trigger Lockdep's deadlock detection logic for the target lock class (see Section 2.6.1).

`LL`'s goal is to re-use Lockdep's deadlock detection logic which in turn relies on tracing any lock class uses in the kernel (modules) through the lockdep metadata. According to prior work (Angelakopoulos et al., 2023; Angelakopoulos et al., 2024a; Angelakopoulos et al., 2024b), the memory layout of data structures, common between the IoT kernel modules and the custom kernel (used for re-hosting) must be consistent. This requirement is mandatory to avoid incorrect data structure member accesses (by the modules) during re-hosting, which in turn might lead to kernel module crashes. Thus, inserting the lockdep metadata in the custom kernel's locking data structures, by simply enabling Lockdep, would violate the requirement of shared data structures being aligned and thwart the normal

execution of the IoT kernel modules.

Instead, the *SLoMs* enable Lockdep to trace the accesses of lock classes, without affecting the layout of locking data structures. A *SLoM* is a shadow memory which stores the same information as a lockdep map, without that information being embedded within the locking data structures. Figure 7·2c provides a visual representation of how a *SLoM* stores the lockdep metadata (i.e., `magic`, `owner_cpu`, `owner`, and `dep_map`) which would normally be inlined in the `struct raw_spinlock` data structure (see Figure 7·2b). The use of the *SLoMs* guarantees that the layout of the locking data structures is not affected by the lockdep metadata even if Lockdep is enabled in the custom kernel (since the metadata are decoupled from the data structures). For Lockdep to leverage the *SLoMs*, `LL` first needs to outline the lockdep metadata use cases in the custom kernel code (see Section 7.1.2), and next, instrument the Lockdep annotations to interact with the *SLoMs*, instead (see Section 7.1.3). This means that `LL` does not require access to the source code of binary-only IoT kernel modules and re-compile them with Lockdep enabled. `LL` simply enables Lockdep in the custom kernel and applies the former's mature deadlock detection analysis on these modules.

7.1.2 Lockdep Metadata Usage Inference

In stage ①, `LL` identifies all the locations in the target kernel source code that are annotated by Lockdep. Stage ② instruments the identified locations to use the *SLoMs* instead of the lockdep metadata, so that Lockdep can support the binary-only IoT kernel modules. To extract this information, `LL` statically analyzes the target firmware to identify the kernel version used by the firmware's IoT kernel and the configuration options enabled during its compilation, following the same approach as prior work (Angelakopoulos et al., 2023; Angelakopoulos et al., 2024b).

The kernel version indicates which (open-source) kernel tree `LL` must use to build the custom kernel for the target firmware. Importantly, the extracted kernel configuration pro-

vides three contributions. First, it constitutes the configuration of LL's custom kernel, which approximates the IoT kernel. Second, it shows the compatibility of the firmware's modules with LL. LL requires that the IoT kernel and its modules are compiled with the CONFIG_DEBUG_SPINLOCK or CONFIG_PREEMPT (or both) configuration options enabled. The former option is related to debugging the use of spinlocks, and is generally not enabled in production systems. CONFIG_PREEMPT yields a preemptible kernel, suited for low-latency systems, and is frequently enabled (used by 51% of my images) for modern IoT devices.

When the aforementioned options are enabled, all the locking and unlocking operations (e.g., _raw_spin_lock and _raw_spin_unlock) are defined as exported (global) functions in the IoT kernel, instead of being inlined. Since LL has complete control over the custom kernel's code, LL modifies the Lockdep subsystem (in stage ②) to analyze IoT kernel modules, by using the lockdep annotations within the exported locking functions. If the IoT kernels are compiled without these options enabled, then a subset of the lock-related operations will be inlined within the IoT module code. These inlined functions do not contain any Lockdep annotations (i.e., the macros for the annotations expand to no code) since the IoT kernel and kernel modules are compiled without Lockdep enabled. Thus, the custom kernel's Lockdep subsystem would immediately generate errors due to the partial tracking of the acquisitions and releases of locks used by the IoT kernel modules. A solution to this problem would be to statically re-write the binary-only IoT kernel modules to insert the Lockdep annotations in any lock-related operation. However, this approach is orthogonal to LL and thus I leave it as future work.

The third contribution of the IoT kernel's configuration is to assist LL identify the C files (with .c and .h extensions), used to build the custom kernel. LL identifies which of these files use locking primitives and are annotated by Lockdep, should Lockdep be enabled during compilation. LL has to instrument these annotations to use the *SLoMs* instead of the

lockdep maps. Once this analysis is complete, LL proceeds to stage ②.

7.1.3 Kernel Instrumentation

Stage ② is responsible for building a custom kernel capable of analyzing the binary-only IoT kernel modules of the target firmware for deadlock-based bugs. To build the custom kernel, LL consumes the information extracted in stage ①, such as the target kernel's configuration and information about the accesses of lockdep metadata within the target kernel's source files. Based on this information, LL instruments all the lockdep metadata accesses to use the *SLoMs*, instead. Next, I detail the steps that LL follows during this stage.

First, LL modifies the custom kernel's source tree to add the support for the *SLoMs*. The functionality of LL's instrumentation is twofold: 1) The insertion of the *Shadow Lockdep* subsystem in the custom kernel tree. This subsystem is responsible for allocating, initializing and retrieving the *SLoMs* which contain the lockdep metadata associated with each used lock. 2) The modification of the Lockdep annotations to use the *SLoMs* in lieu of the lockdep metadata within the locking data structures. In particular, LL relies on the extracted information from stage ① regarding the lockdep metadata uses to alter every annotation (i.e., call to the Lockdep subsystem) to use a *SLoM* instead of the lockdep metadata. Furthermore, the custom kernel is compiled with the CONFIG_PROVE_LOCKING option enabled, which includes Lockdep's deadlock detection logic in the kernel. However, to ensure that the layout of locking data structures does not change with the introduction of Lockdep, LL also modifies the implementation of these data structures and removes the lockdep metadata therein. Next, LL uses the modified source code of the custom kernel and the kernel configuration extracted in stage ① along with Pandawan to build the custom kernel that can both re-host and analyze the target firmware's binary-only kernel modules with Lockdep.

7.1.4 Deadlock Detection

As soon as stage ② generates the custom kernel, LL proceeds to re-host the binary-only kernel modules of the target firmware using the custom kernel and Pandawan. This re-hosting step serves two purposes: 1) LL identifies which binary-only kernel modules can be successfully loaded by the custom kernel and by extension can be analyzed with Lockdep, and 2) LL extracts important information during the re-hosting process and transforms this information into system call descriptions for the grammar-based fuzzer syzkaller. Examples of the information extracted from the re-hosting process are the registered character devices, registered sockopt operations, and registered netfilter hooks. Since the code of the binary-only IoT kernel modules is not publicly available, there are no system call descriptions available for these kernel modules and thus syzkaller has no information on how to fuzz their code. However, the use of the descriptions generated by LL allows syzkaller to successfully fuzz-test binary-only targets.

Unfortunately though, for the same reason that LL cannot use the unmodified version of Lockdep, LL cannot use syzkaller, unmodified, to fuzz the binary-only IoT kernel modules. Syzkaller's, coverage feedback relies on the kernel's KCOV (Vyukov, 2025) instrumentation, inlined in the kernel during the compilation process if the CONFIG_KCOV configuration option is enabled. The CONFIG_KCOV option is not enabled during the compilation of the binary-only IoT kernel modules and thus the KCOV instrumentation is absent from these kernel modules. Thus, I modify syzkaller to extract basic block coverage information from QEMU (i.e., the Translation Blocks) instead of relying on the kernel's KCOV instrumentation. For this reason, I instrument QEMU to provide the executed Translation Block (TB) information to syzkaller as coverage feedback.

Finally, as networking modules are a substantial subset of the binary-only IoT kernel modules in my dataset, LL uses a custom network traffic generator to send random packets to the re-hosted custom kernel. By using the traffic generator, LL aims to trigger packet han-

dling logic within the loaded binary-only IoT kernel modules (i.e., their registered netfilter hooks).

7.2 Implementation

In this section, I provide the implementation details behind `LL`.

7.2.1 Shadow Lockdep Maps (*SLoMs*)

As stated in Section 7.1.1 it is necessary to decouple Lockdep's metadata (e.g., the lockdep maps) from the locking data structures. The decoupling process ensures that the locking data structures' memory layout, in the custom kernel, matches the layout that the binary-only IoT kernel modules expect. The *SLoMs* serve as a surrogate storage for the decoupled lockdep metadata. By modifying Lockdep's annotations, the transition from the lockdep maps to the *SLoMs* is completely transparent to the Lockdep subsystem.

I implement the *SLoMs* into a separate (shadow) memory that is initialized and indirectly associated with the locks used in the kernel via the *Shadow Lockdep* subsystem. Specifically, I leverage Linux's hashtable (KernelNewbies, 2017), introduced in the `v3.7` version of the kernel, to create a one-to-one mapping between the *SLoMs* and the kernel locks. Like other hashtables, Linux's hashtable functions like a key-value store data structure, which supports the *add*, *delete* and *search* operations. Whenever the kernel (module) code dynamically initializes a lock (e.g., a `mutex` through the `__mutex_init` function), *Shadow Lockdep* creates a new *SLoM* and stores it in the (*SLoM*) hashtable using the lock's address as the key. As discussed in Section 7.2.3, dynamic locks get assigned their lock class (key) through their initialization functions. Next, every time Lockdep requires the lockdep metadata related to the lock (e.g., to initialize/acquire/release its lock class), the kernel retrieves the *SLoM* from *Shadow Lockdep* by performing a hashtable lookup using the lock's address. The *SLoM* creation and retrieval operations are implemented via the

Shadow Lockdep's APIs that are exposed to the entire kernel (see Section 7.2.3).

I note here that LL treats statically defined locks (e.g., locks defined by macros such as DEFINE_MUTEX or DEFINE_SPINLOCK) differently from dynamically created ones. As static locks do not go through any initialization functions such as __mutex_init, it is impossible for *Shadow Lockdep* to be aware of the creation times of these locks during the kernel (module) execution and correspondingly assign these locks a *SLoM*. Fortunately, unlike dynamic locks, static locks use the lock (object) itself as their lock class (key). In addition, as static locks are not assigned a *SLoM*, the *SLoM* lookup operation for these locks would cause a miss in the hashtable. While *Shadow Lockdep* is unaware of the creation of a static lock, it is though aware of accesses to the lock. I capitalize on this observation and enhance the *SLoM* hashtable lookup operation to dynamically create and initialize a *SLoM* during a miss in the hashtable (i.e., the first time a locking operation accesses a static lock). This ensures that LL correctly handles both static and dynamic locks. Next, I discuss in detail the three stages of LL.

7.2.2 Lockdep Metadata Usage Inference

Stage ① is responsible for identifying all the locations in the target kernel source code that are annotated by Lockdep and use the lockdep metadata. In stage ②, LL uses this information to instrument these locations to use the *SLoMs* instead, enabling Lockdep to support the binary-only IoT kernel modules.

To detect the code locations where the lockdep metadata are used, LL at first uses Pandawan to analyze the target firmware image and retrieve the version and the (Kbuild) configuration of the image's IoT kernel. On the one hand, the kernel version information simply corresponds to the open-source Linux kernel code that is used in stage ② to build LL's custom kernel. On the other hand, LL uses the kernel configuration to build the custom kernel in stage ②. However, prior to building the custom kernel, LL first checks if the configuration contains the CONFIG_DEBUG_SPINLOCK or CONFIG_PREEMPT (or both) op-

tions and only then proceeds with the kernel compilation. These options force all locking operations initiated by the binary-only IoT kernel modules to be explicit calls to global functions exported by the IoT kernel and by association the custom kernel. Since `LL` has complete control over the custom kernel's code, `LL` can compile the kernel with Lockdep enabled (toggling on the `CONFIG_PROVE_LOCKING` option) and thus include all the Lockdep annotations within the (exported) locking functions. In turn, Lockdep can successfully and accurately keep track of all the lock acquisitions and releases (i.e., the calls to the exported locking functions) initiated by the binary-only kernel modules which use the exported locking functions.

Finally, `LL` uses the IoT's kernel configuration to identify all the C source files (`.c` and `.h`) that make use of lockdep annotations and metadata, when Lockdep is enabled. While a trivial string-based search (e.g., *grep*) would yield multiple results, it would still be insufficient due to the presence of macros that indirectly use lockdep maps. Instead, `LL` uses Clang's (Clang Project, 2025b) libtooling and AST matchers (Clang Project, 2025a) which identify both direct accesses of lockdep maps (e.g., in functions) or indirect uses, by expanding macros. Specifically, `LL` first conducts a compilation of the custom kernel using the IoT kernel's configuration, also with Lockdep enabled (but without any instrumentation). Simultaneously, `LL` relies on Bear (Nagy, 2025) to generate a database with all the C source files (and their compilation commands) used to build the kernel. Next, `LL` uses Clang to analyze the files in the database and detect all the locations that access lockdep metadata (i.e., all the code annotated by Lockdep). This information is then passed to stage ②.

7.2.3 Kernel Instrumentation

The goal of stage ② is to build a custom kernel that can re-host and analyze the target firmware's binary-only IoT kernel modules with Lockdep. `LL` modifies the source code of the custom kernel, specifically the lockdep metadata accesses found in stage ①, to add

support for the *Shadow Lockdep* subsystem and the *SLoMs*.

Hashtable	Description	API	Description
lock_htable	Caches the shadow lockdep maps (*SLoMs*)	`gen_shadow_lockdep_map`	Generate a *SLoM* and insert it in `lock_htable`
		`get_shadow_lockdep_map`	Look for an existing *SLoM* in `lock_htable` or create a *SLoM* for a static lock
key_htable	Caches the `lock class keys` for locks	`get_shadow_lockdep_key`	Get or create a `lock class key` for a module lock

Table 7.1: Shadow Lockdep's hashtables and the functions that access the tables.

Shadow Lockdep. Internally, the *Shadow Lockdep* subsystem handles operations (store/lookup) for hashtables (shadow memory) and exposes a custom API to the entire kernel for interacting with these hashtables. In particular, *Shadow Lockdep* effectively manages two hashtables (i.e., *lock_htable*, and *key_htable*) shown in column one of Table 7.1. The *lock_htable* is the primary hashtable maintained by *Shadow Lockdep*. Its purpose is to store information about the *SLoMs* (hashtable values) of all the locks (hashtable keys) used by the custom kernel and the binary-only IoT kernel modules. The *Shadow Lockdep* subsystem exposes two API functions `gen_shadow_lockdep_map` and `get_shadow_lockdep_map` that access the *lock_htable* and are globally accessible throughout the entire kernel code (via the `EXPORT_SYMBOL` macro). The first function is responsible for creating a new *SLoM* and adding it to *lock_htable*. The second function's purpose is to retrieve an already existing *SLoM* from *lock_htable*. As I discuss later, `LL` instruments the kernel's lock initialization functions (e.g., `mutex_lock`) to call `gen_shadow_lockdep_map` function and create a *SLoM* for a dynamically created lock. In contrast, static locks are not created by the standard lock initialization functions. Instead, these locks are initialized directly by the `DEFINE_<LOCK>` macros, which cannot be instrumented to call `gen_shadow_lockdep_map`. For this reason, I extend `get_shadow_lockdep_map` to create and initialize a *SLoM* (using the type of the lock) in case a static lock is accessed for the first time, either by the core kernel or a binary-only kernel module.

```
1 #define mutex_init(mutex)\
2 do {                       \
3  static struct lock_class_key \
4                  __key; \
5  __mutex_init((mutex), #mutex, \
6               &__key);\
7 } while (0)
```

(a) The `mutex_init` macro.

```
1 void __mutex_init(struct mutex
        *lock, const char *name,
        struct lock_class_key *key)
2 {
3 ...
4 #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
5  osq_lock_init(&lock->osq);
6 #endif
7
8  debug_mutex_init(lock, name, key);
9 }
```

(b) The `__mutex_init` function.

```
1 void debug_mutex_init(struct mutex
        *lock, const char *name,
2       struct lock_class_key *key)
3 {
4 #ifdef CONFIG_DEBUG_LOCK_ALLOC
5 ...
6  lockdep_init_map(&lock->dep_map,\
7                   name, key, 0);
8 #endif
9  lock->magic = lock;
10 }
```

(c) The `debug_mutex_init` function.

Figure 7·3: Chained initialization of a kernel mutex lock in Linux v5.4.164.

The *key_htable* generates unique `lock class keys`, representing lock classes (hashtable values) for locks (hashtable keys) that are dynamically created and used by the binary-only IoT kernel modules. Normally when Lockdep is enabled, every lock is assigned a lock class key (i.e., a lock class), that is always the same for every instance of the lock. For example, Figure 7·3 illustrates how a lock of type `struct mutex` gets assigned a lock class key from Lockdep. For most lock types, such as my example, the lock class key (see lines 3-4 in Figure 7·3a) is a statically defined data structure of type `struct lock_class_key`. Since all instances of the same lock are initialized by the same function (e.g., `param_lock` is initialized by `init_param_lock` function, in `kernel/module.c`, which calls the `mutex_init` macro), they will all be assigned the same lock class key. Through the `mutex_init` macro, the lock class key is propagated to the `__mutex_init` function (see lines 5-6 in Figure 7·3a), then to the `debug_mutex_init` function (see line 8 in Figure 7·3b). Finally, the lock class key gets associated with the lock through Lockdep's `lockdep_init_map` function (see lines 6-7 in Figure 7·3c).

Unfortunately, within the binary-only IoT kernel modules, the data structure `struct lock_class_key` is interpreted as a type without contents (zero size), since these modules are compiled with Lockdep disabled. As a result, the pointer to a zero size `struct lock_class_key` propagated to the `lockdep_init_map` function will always point to the same address (in the module's `BSS` section) regardless of the lock type. Thus, every non-

static lock (type) initialized by a binary-only IoT kernel module would always be assigned the same class, which is incorrect. To mitigate this issue, I rely on the fact that dynamically created locks are assigned specific names (e.g., `param_lock`) during their initialization (see the second argument of `__mutex_init` in Figure 7·3b). Based on this observation, whenever a binary IoT kernel module dynamically creates a lock, `LL` uses the name of the lock as a unique identifier to create a new lock class for the lock and store it in *key_htable*. Thus, `LL` will award the same lock class to all lock instances using the same name and created by the same kernel module.

```
#define lock_acquire_exclusive(l, s, t, n, i)           lock_acquire(l, s, t, 0, 1, n, i)
...
#define spin_acquire(l, s, t, i)                    lock_acquire_exclusive(l, s, t, NULL, i)
#define spin_release(l, n, i)                       lock_release(l, n, i)
...
#define mutex_acquire(l, s, t, i)                   lock_acquire_exclusive(l, s, t, NULL, i)
#define mutex_release(l, n, i)                      lock_release(l, n, i)
```

(a) Stock Lockdep annotations in `include/linux/lockdep.h`

```
#define lock_acquire_exclusive(l, s, t, n, i, m)            shadow_map_lock_interim(l, s, t, 0, 1, n, i, m)
...
#define spin_acquire(l, s, t, i)                    lock_acquire_exclusive(l, s, t, NULL, i, SPINLOCK_MAGIC)
#define spin_release(l, n, i)                       shadow_map_release_interim(l, n, i)
...
#define mutex_acquire(l, s, t, i)                   lock_acquire_exclusive(l, s, t, NULL, i, MUTEX_MAGIC)
#define mutex_release(l, n, i)                      shadow_map_release_interim(l, n, i)
```

(b) `LL`'s Lockdep annotation instrumentation in `include/linux/lockdep.h`

```
void shadow_map_lock_interim(void *lock, unsigned
      int subclass,
         int trylock, int read, int check,
      struct lockdep_map *nest_lock,
      unsigned long ip, unsigned int magic)
{
  struct shadow_lockdep_map *slom;
  slom = get_shadow_lockdep_map(lock, subclass,
       magic);
  lock_acquire(slom->dep_map, subclass, trylock,
       read, check, nest_lock, ip);
}
```

(c) `LL`'s `lock_acquire` wrapper function.

```
void shadow_map_release_interim(void *lock, int
      nested,
         unsigned long ip)
{
  struct shadow_lockdep_map *slom;
  slom = get_shadow_lockdep_map(lock, 0, -1);
  lock_release(slom->dep_map, nested, ip);
}
```

(d) `LL`'s `lock_release` wrapper function.

Figure 7·4: `LL`'s Lockdep annotation instrumentation.

Lockdep Annotations. At its core, Lockdep uses three main functions to conduct its deadlock detection analysis (i.e., the `lockdep_init_map`, `lock_acquire`, and `lock_release` functions), all of which operate on lockdep maps. The function `lockdep_init_map` is used to initialize the lockdep map and assign a lock class to a lock.

The `lock_acquire` and `lock_release` functions trace the acquisition and release of a lock (class), respectively and trigger Lockdep's deadlock detection for the target class. However, these functions are rarely used directly within the kernel code. Instead, the kernel defines wrapper functions and macros around these functions, corresponding to each lock type (e.g., `mutex_acquire` for mutexes) which are used to annotate each locking operation (e.g., `mutex_lock`). `LL` relies on the information extracted from stage ①, to find and instrument all invocations to these wrappers to use the *SLoMs* instead of the lockdep maps inlined within the locking data structures. Specifically, `LL` makes use of *Shadow Lockdep*'s API functions (see Table 7.1) to either generate new or retrieve existing *SLoMs* and lock class keys, from the *lock_htable*, and *key_htable* hashtables. Figure 7·4 provides more detailed information about `LL`'s instrumentation regarding Lockdep's annotations. Specifically, as shown in Figure 7·4a, Lockdep defines lock type-specific wrappers (macros) around its core deadlock analysis functions (i.e., `lock_acquire` and `lock_release`), which are used throughout the kernel code to annotate locking operations (e.g., `mutex_lock`). Figure 7·4b shows how `LL` modifies these macros to expand to its own custom wrappers, `shadow_map_lock_interim` and `shadow_map_release_interim`. For example, `mutex_acquire` expands to `shadow_map_lock_interim` (see lines 1 and 6 in Figure 7·4b). In turn, `LL`'s custom wrappers are responsible for retrieving the *SLoMs* for the target lock (see lines 6-7 in Figure 7·4c and lines 4-5 in Figure 7·4d). Next, `LL` calls Lockdep's `lock_acquire` (see line 8 in Figure 7·4c) and `lock_release` (see line 6 in Figure 7·4d) using the lockdep metadata stored within the *SLoMs*, to initiate the deadlock detection analysis for the target lock (class).

Finally, `LL` uses the instrumented custom kernel code, the IoT kernel's configuration (extracted in stage ①) and Pandawan's kernel build facility to build the custom kernel. The custom kernel is capable of both re-hosting and analyzing the binary-only IoT kernel modules for deadlock-based bugs with Lockdep. Next, I discuss two special cases of modi-

```
1  struct module {
2    enum module_state state;
3    ...
4  #ifdef CONFIG_TRACEPOINTS
5    unsigned int num_tracepoints;
6    tracepoint_ptr_t *tracepoints_ptrs;
7  #endif
8  ...
9  #ifdef CONFIG_TRACING
10   unsigned int num_trace_bprintk_fmt;
11   const char **trace_bprintk_fmt_start;
12 #endif
13 #ifdef CONFIG_EVENT_TRACING
14   struct trace_event_call **trace_events;
15   unsigned int num_trace_events;
16   struct trace_eval_map **trace_evals;
17   unsigned int num_trace_evals;
18 #endif
19 ...
20 }
```

Figure 7·5: The memory layout of `struct module` in Linux v5.4.164.

fications implemented by `LL`, required for the normal loading and execution the binary-only IoT kernel modules in the custom kernel.

Struct module. Except the locking data structures (e.g., mutex, spinlock, semaphore, etc), `LL` has to make sure that the layout of the `struct module` data structure is unaffected when enabling Lockdep in the custom kernel. The `struct module` data structure (i.e., the representation of a kernel module within the kernel) is frequently accessed by the IoT kernel modules and thus its layout in the custom kernel has to be consistent with the layout that the IoT kernel modules expect. Unfortunately, `struct module` is indirectly affected by Lockdep when the latter is enabled during the compilation of the kernel. Specifically, enabling Lockdep will force-select all the tracing options in the kernel (e.g., `CONFIG_TRACING`, `CONFIG_TRACEPOINTS`, and `CONFIG_TRACE_EVENTS`) which introduce additional members within `struct module` (see lines 4-7, 9-18 in Figure 7·5). When it comes to the binary-only IoT kernel modules these tracing options are usually disabled, since they are primarily used for debugging and monitoring the behavior of the kernel, thus they are not enabled in production systems such as the IoT kernels. Thus, to make sure that these options do not affect the layout of `struct module` when Lockdep is enabled, I enhance *Shadow Lockdep* to also store the tracing information in lieu of `struct module`, similar to how *SLoMs* store the lockdep metadata in-

```
1 stmdb       sp!,{r4,lr}
2 cpy         r3,sp
3 bic         r3,r3,#0x3fc0
4 bic         r3,r3,#0x3f
5 ldr         r1,[r3,#0x10]
6 cmp         r1,#0x3
7 bls         LAB_00042f44
8 ...
```

Figure 7·6: Assembly for accessing the `preempt_counter` on the stack

stead of the locking data structures. In turn, whenever these information is needed by the kernel code (e.g., by the `module_trace_bprintk_format_notify` function within `kernel/trace/trace_printk`), it is retrieved from *Shadow Lockdep* similar to how the kernel retrieves the *SLoMs*. I note here that `struct module` is the only case of a data structure that is indirectly impacted by Lockdep and must be correctly aligned to prevent misaligned member accesses from the binary-only IoT kernel modules during re-hosting.

Thread size. While re-hosting the modern Netgear images in my datasets, I notice that certain kernel modules (e.g., `acos_nat.ko`) produce a *preemption leakage* error (i.e., the `preempt_counter` is corrupted) when loaded in `LL`'s custom kernels. In turn, this error causes Lockdep to produce a false error (a false positive). Upon further inspection of the modules' code under Ghidra (NSA Research Directorate, 2025), I deduce that the new IoT kernels may be modified such that the `THREAD_SIZE` (i.e., the per thread kernel stack size) is double the normal size (i.e., 16KB instead of 8KB) for ARM architectures. I confirm my findings by also inspecting the GPL source code of a Netgear image in my dataset. Due to this modification, the affected kernel modules access the `thread_info` data structure at an incorrect offset on the stack. The `thread_info` data structure stores important metadata about kernel threads, such as the current value of the `preempt_counter` (used to toggle on/of thread preemption). The incorrect access of `thread_info` also results in the incorrect reading and writing of the `preempt_counter` when preemption is enabled or disabled (e.g., with the `preempt_enable` and `preempt_disable` functions). Thus, the `preempt_counter` is not updated correctly leading to the kernel and Lockdep issuing

warnings. To address this issue, LL analyzes the IoT kernel modules of the target image, with a Ghidra script, in stage ② before the compilation of the custom kernel to identify the correct value of THREAD_SIZE. Specifically, the Ghidra script detects patterns in the assembly code of a kernel module that suggest an access of thread_info on the stack. Figure 7·6 showcases the assembly instructions that access thread_info (see the lines 2-4). These instructions correspond to $r3 = sp \ \& \sim (0x3fc0 \mid 0x3f) = sp \ \& \sim 0x3fff$, where $0x3fff = 0x4000 - 1$ (i.e., 16KB THREAD_SIZE - 1) The MIPs kernel modules also use similar patterns. By searching for these instruction patterns LL is able to find the THREAD_SIZE for all images in the dataset and set the correct values in the custom kernels.

7.2.4 Deadlock Detection

Once the custom kernel is built in stage ②, LL proceeds to re-host the binary-only IoT kernel modules of the target firmware with Pandawan. The purpose of this step is to reveal which of these modules can be successfully re-hosted by LL's custom kernel and also provide information that can be used by LL to create system call descriptions for syzkaller.

To discover deadlocks within binary-only IoT kernel modules, LL uses the custom kernel and the syzkaller kernel fuzzer to re-host and exercise the code of the (successfully loaded) binary-only IoT kernel modules, while they are being analyzed by Lockdep. Despite its overall complexity, syzkaller primarily consists of two components, the syz-manager (the actual fuzzing process running on the host) and the syz-executor (an agent executing system call sequences within the guest VM), which communicate with each other over the network. Internally, syz-executor sets up a memory (buffer) which caches the coverage information related to the executed system calls. Whenever the syz-executor executes a system call, KCOV places the collected coverage in the buffer and in turn syz-executor sends the coverage information back to the syz-manager through the network.

To add support for binary-only targets, I modify the syz-executor program to retrieve coverage feedback from QEMU (i.e., the kernel level Translation blocks – TBs) instead

of KCOV, analogous to TriforceAFL (NCC Group Plc, 2017) and LibAFL (Fioraldi et al., 2022). To this end, I implement a hypercall which is trapped by QEMU and used to transfer information from the syz-executor to the QEMU hypervisor. In particular, syz-executor uses the hypercall to make the virtual address of the coverage buffer known to QEMU. Since QEMU has complete control over the guest (i.e., the target firmware) VM's memory, I use QEMU's existing API's to directly access the buffer's guest physical pages. Next, QEMU traces the module TBs and places the coverage information in the coverage buffer. I make sure that QEMU only traces TBs that belong to kernel modules by restricting coverage collection to the address regions where the binary-only IoT kernel modules are loaded (e.g., `0xbf000000` - `0xc0000000` on a classic 3GB/1GB memory split ARM 32-bit target). In addition, syz-executor uses the hypercall to only collect coverage related to the executed system calls by the binary-only IoT kernel modules, to avoid overflowing the coverage buffer due to unintended noise (e.g., a module timer frequently firing).

Finally, `LL` parses the emulation logs of the re-hosting step with Pandawan and retrieves information about the registered character devices, registered sockopt callbacks (domain and optname values), and registered netfilter hooks by the target firmware's binary-only IoT kernel modules. In cases where a kernel module exposes a character device, `LL` uses FirmSolo's `ioctl` command extraction script to identify potential command numbers that can be used to interact with the module through its `ioctl` interface. Afterwards, `LL` uses the `ioctl` and `sockopt` information to automatically create system call descriptions that are used by syzkaller to execute the code within the binary-only IoT kernel modules while these modules are being analyzed by Lockdep. These descriptions are generic, only using the information extracted above since there is no module source code available or semantic information within the module binaries to create accurate descriptions. In addition, for binary-only IoT kernel modules with registered netfilter hooks, `LL` analyzes the identified hook callbacks with Ghidra (NSA Research Directorate, 2025) to extract information about

the type of traffic (e.g., IP version, protocol number, port number) that could potentially trigger these callbacks, if sent to these modules. Next, LL uses a custom network traffic generator that generates random network packets based on the information extracted previously and sends these packets to the re-hosted custom kernel and binary-only IoT kernel modules to trigger these hooks while being analyzed by Lockdep.

7.3 Evaluation

In this section I evaluate LL's capability (via the *Shadow Lockdep* subsystem and the *SLoMs*) to analyze binary-only IoT kernel modules for deadlock-based bugs. First, I provide information on my dataset and next I detail the experiments I conduct to assess LL's capabilities.

7.3.1 Dataset

My dataset consists of 949 firmware images, 213 modern images (using kernels above v4.x) from Greenhouse (Tay et al., 2023) (shared with me by its authors) and 736 images downloaded manually by us. In general, I opt for images with kernel versions above v3.7, which are used by IoT devices, currently supported by IoT vendors and available for purchase, and also support the Linux hashtables (added in the v3.7 kernel version). First, I exclude 395 firmware images whose file-system I could not extract (e.g., due to the image being encrypted). Next, I filter out 145 images not supported by Pandawan and by extension LL (currently supports only the ARM and MIPS 32bit architectures) and 156 images that use kernels below v3.7. From the remaining 253 images that LL can re-host, I only keep the ones with kernels built with the CONFIG_PREEMPT (128) and CONFIG_DEBUG_SPINLOCK (1) configuration options, required by LL's deadlock detection (see Section 7.1.3). Thus, my evaluation dataset consists of 129 firmware images containing 19,529 kernel modules, 6,500 of which are binary-only (i.e., there is no open-source variant in the kernel tree).

There are 12 kernel versions in my dataset, spanning from v3.14.77 to v5.4.213.

7.3.2 Detection of Known Deadlock Bugs

In this section, I evaluate `LL`'s ability (specifically the *Shadow Lockdep* subsystem) to detect deadlocks in open-source binary targets that are intentionally compiled without Lockdep. As hundreds of deadlock-based bugs have been found in the Linux kernel by syzkaller over the years, I choose to run `LL`'s analysis on in-tree kernel modules for which deadlock bugs were previously disclosed. In addition, I make sure to choose bugs found in kernel versions close to the ones used by the firmware images in my dataset (i.e., v4.x - v5.x) to further highlight `LL`'s analysis compatibility with the IoT privileged firmware. I also make sure that the bugs come with an available reproducer program for rapid testing. Note that the reproducer programs do not actually trigger the corresponding deadlock, but merely trigger the detection of a potential deadlock scenario by Lockdep (see Section 2.6.1). To this end, I choose 3 distinct previously known deadlock bugs (Syzbot, 2018b; Syzbot, 2018a; Syzbot, 2020) discovered by syzbot (i.e., syzkaller's automated fuzzing tool) in Linux's Point-to-Point (PPP) network protocol, the Network bonding facility, and the USB usbmon functionality. The bugs affecting the PPP (*ppp_generic*) and bonding kernel modules are associated with a recursive locking (doubling locking) scenario of a lock (i.e., `all_ppp_mutex` and `bond->stats_lock`) and are fixed in the kernel versions v4.15 and v4.18, respectively. The usbmon bug is related to a circular locking scenario (i.e., an ABBA deadlock) involving the `mmap_sem` lock (used in Linux's memory management subsystem) and `fetch_lock` used by the usbmon kernel module and was fixed in the kernel version v5.4.

To treat these kernel modules as binary-only IoT kernel modules, I implement the following. First, I use the stock kernel versions v4.14 (affected by the PPP and bonding bugs) and v4.19 (affected by the usbmon bug) to build the kernel modules. Specifically, I reuse the same kernel configurations used by syzbot during its tests. However, I compile

Module	Vendor	Kernel	TBs	Speed (exec/time)	Bugs
Syzkaller					
acos_nat	Netgear	5.4.213	453	112/h	1
acos_nat	Netgear	5.4.164	387	94/h	1
app-dpi	TPLink	4.19.183	85	21/m	1
NTG					
dpicore	Netgear	4.19.294	153	n/a	1*
tdts	Netgear	4.19.294	512	n/a	1
ndi	Netgear	4.19.294	154	n/a	1*
				Total	6

Table 7.2: Deadlock analysis results. The table contains both the results from the fuzzing experiments with syzkaller as well as the experiments with LL's custom NTG (Network Traffic Generator). The * indicates that bug was found when unloading the kernel module after NTG interacts with it.

PPP, bonding and usbmon as kernel modules (i.e., `ppp_generic.ko`, `bonding.ko`, and `usbmon.ko`) without Lockdep enabled and I also toggle on the `CONFIG_PREEMPT` option (see Section 7.1.1). I note here that to trigger the warning for the double locking bug in the `bonding.ko` module, I compile this kernel module also with the `CONFIG_DEBUG_SPINLOCK` enabled. This is necessary to mitigate LL's overestimation of the classes of spinlocks initialized by binary kernel modules compiled without Lockdep enabled (see Section 7.4). Separately, I adapt LL's kernel instrumentation on the kernel versions v4.14 and v4.19 to build the custom kernels with support for the *Shadow Lockdep* subsystem and the *SLoMs*. Next, I re-host the affected kernel modules using my custom kernels with Lockdep enabled, and execute the syzbot reproducer programs to trigger the Lockdep warnings. My experiments, successfully reproduce the Lockdep warning (or "splat") identical to the syzbot bug reports. Thus, LL can find known deadlocks in binary kernel modules compiled without Lockdep.

7.3.3 Kernel Module Deadlock Analysis

In this section, I evaluate LL on the binary-only IoT kernel modules in my dataset to assess LL's ability to detect potential deadlocks. First, LL re-hosts the binary-only IoT kernel modules to identify which of these kernel modules can successfully load in the custom

kernels. My re-hosting experiments show that LL successfully loads 3,454/6,500 (53%) binary-only kernel modules. I note that without LL's instrumentation these kernel modules would immediately crash during my deadlock analysis experiments due to misaligned data structure accesses, an issue identified in prior works (Angelakopoulos et al., 2023; Angelakopoulos et al., 2024b; Angelakopoulos et al., 2024a; Pustogarov et al., 2020). Since not every kernel module uses locks, I only choose to analyze the binary-only kernel modules in my dataset that meet the following criteria; 1) The kernel modules use locks (i.e., make calls to exported locking functions, such as `mutex_lock`) and thus can be susceptible to deadlock-based bugs (1,769 modules), and 2) The kernel modules export a character device or register a get- or setsockopt callback or networking hooks with the kernel's Netfilter subsystem (680 modules). The premise behind using the second criterion, is the ability to trigger entry points (e.g., system calls, such as `ioctl` and `setsockopt` or traffic patterns) that lead to the execution of the code in these modules. In general, due to the closed-source nature of the binary-only IoT kernel modules, there is little to no information and guidelines publicly available about their functionality and expected usage, thwarting any attempts for analyzing the majority of these modules. In total 520 binary-only IoT kernel modules in 117 firmware images meet both criteria. However, due to limited computational resources I choose to analyze only 80/520 unique (by module file name and kernel version) modules.

Specifically, LL analyzes 60 modules that export a character device or use a sockopt, and constructs system call descriptions for syzkaller to fuzz these kernel modules. In addition, LL identifies 89 registered netfilter hooks in 35 binary-only IoT kernel modules. I note that a kernel module can both expose a character device and register netfilter sockopt or hook callbacks. As mentioned in Section 7.2.4, LL analyzes the hook callbacks with Ghidra to extract the IP version, protocol and port numbers that may be used by these hooks. The network traffic generator consumes the information and creates random packets and sends them to the re-hosted target to exercise the modules' code. I run the fuzzing experiments

five times for 24 hours and the experiments with the network traffic generator, ten times. For my experimental setup, I use machines with Intel Xeon CPUs with a minimum of 16GB of RAM. I provide the relevant information about my deadlock analysis experiments in Table 7.2. In total, my deadlock analysis produces 6 potential deadlock-based bugs in 6 binary-only IoT kernel modules, which I validate manually.

I note that throughout my experiments I do not encounter any crashes (false positives) related to the use of the *SLoMs*. This observation further reinforces my claim regarding the ability of Lockdep to analyze binary-only targets with the help of *Shadow Lockdep* and the *SLoMs*. I reported the found bugs to the affected vendors. So far TPLink and Zyxel have communicated with me about the issues. Next, I provide more information about the bugs that I found in the `tdts.ko` and its dependent modules, `ndi.ko` and `dpicore.ko`.

Specifically, `tdts.ko` is a proprietary kernel module developed by TrendMicro and used by a Netgear firmware image. The purpose of this kernel module is to enhance the target firmware with *Intrusion Prevention System* (IPS) and *Deep Packet Inspection* (DPI) capabilities. The kernel module is distributed along with a user-level agent which parses a custom *Trend Real-time Format* (TRF) binary file containing the traffic signatures, rules and policies to be enforced by the firmware. The user agent is responsible for communicating this information to the kernel module via an exposed IOCTL interface. In turn, the kernel module contains functionality that filters network traffic based on these rules and policies. Moreover, `tdts.ko` provides an interface (i.e., functions) that can be indirectly (through pointer manipulation) used by other kernel modules to access these filters. In my case, Netgear's proprietary DPI `dpicore.ko` module is designed to interact with `tdts.ko`. Particularly, this kernel module registers custom hooks with the Netfilter subsystem that use `tdts.ko`'s exported interface (through global pointers) for fine-grained network traffic filtering and inspection. In turn, the `dpicore.ko` module is dependent on the Network Device Interface `ndi.ko` module which also registers hook callbacks with the netfilter sub-

system.

Unfortunately, LL's automated analysis on the tdts.ko and ndi.ko modules is not fruitful. There are two reasons behind these failures. First, the core network data structures struct net, struct net_device and struct skb_buff are incorrectly aligned in the custom kernel, resulting in tdts.ko and ndi.ko crashing when accessing these data structures. I note that the misalignments are not related to the introduction of Shadow Lockdep. These data structures are modified by Netgear in the IoT kernel's source code, and these modifications are absent from the open-source kernel tree used by LL to compile the custom kernel. Initially, my attempts to reverse engineer the layout of these data structures with FirmDiff (Angelakopoulos et al., 2024a) were successful only for struct net and struct net_device. It turns out, Netgear has extensively changed the memory layout of struct skb_buff (e.g., the ordering of the data structure's members), a case that would confuse any automated re-hosting system. For this reason, I had to consult the GPL code of the target kernel to align the layout of struct skb_buff. Second, since LL only uses the binary-only IoT kernel modules in its analysis, it does not make use of the user agent and the TRF file which the tdts.ko module depends on to operate normally. In addition, LL relies on Pandawan's inference of module dependencies, which does not account for indirect (using global pointers) dependencies such as the one between the dpicore.ko and tdts.ko modules. To be able to test the code of tdts.ko kernel module I manually define these dependencies during LL's analysis.

The deadlock is of type *inconsistent lock state*, meaning that a lock can be acquired in both outside and inside an interrupt context by the same thread. This bug usually occurs when the kernel (module) code uses non interrupt-safe locking functions such as __raw_spin_lock instead of __raw_spin_lock_bh to acquire the lock outside and inside of an interrupt context. This means that while a kernel thread holds the lock, it can be interrupted and re-acquire the lock within the interrupt handler, thus blocking indefinitely.

Module	Vendor	Kernel	Bugs
acos_nat	Netgear	5.4.213	2
acos_nat	Netgear	5.4.164	2
bled	Ubiquiti	5.4.213	1
br_hijack	Netgear	5.4.213	1
br_hijack	Netgear	5.4.164	1
ivi	TPLink	4.1.52	1
ivi	Zyxel	4.1.51	1
nciTMSkmod	Zyxel	4.1.51	1
shortcut-fe-ipv6	Netgear	4.4.60	1
		Total	11

Table 7.3: The memory corruption bugs found by `LL`.

Challenges Observed. Here, I stress the fact that detecting deadlock-based bugs in binary-only IoT kernel modules is not a simple task (only 8% of the bugs found by syzbot (Syzbot, 2025) are deadlock-based bugs). First, Lockdep is inherently dependent on code coverage. However, modifying syzkaller's mutation logic to focus only on code that uses locks and on deadlock-based bugs is out-of-scope for this work. Crucially, syzkaller dedicates considerable time fuzzing code paths that lead to (shallow) memory corruption bugs, an issue well-known to prior work (Stephens et al., 2016; Yan et al., 2020; Chen and Chen, 2018; Luo et al., 2023). Interestingly, syzkaller detects 11 memory corruption bugs in 9 binary-only IoT kernel modules that are not related to this work (see Table 7.3). The bugs fall under different categories; arbitrary memory accesses (9), NULL pointer dereference (1), Very large virtual memory allocation (1).

Second, current re-hosting systems, such as Pandawan (used by `LL`) cannot flawlessly discern the configuration of the IoT kernels used by firmware images. Modifications implemented by IoT vendors in their kernels (Angelakopoulos et al., 2024a) prevent re-hosting systems from accurately reconstructing these kernels. As a result, core data structures, such as `struct net_device`, and `struct sk_buff` might be misaligned in `LL`'s kernels. The use of such data structures by the binary-only IoT kernel modules might lead to kernel module crashes. I found 12 such crashes during the experiments with the network traffic

generator. I do not consider these crashes a limitation of LL.

7.4 Discussion

In this Section I discuss the limitations of LL, as well as future directions. As explained in Section 7.1.2, LL requires that the locking operations are not inlined within the kernel module code (e.g., when compiled with CONFIG_PREEMPT configuration option enabled) by and are explicit calls to exported functions in the kernel (e.g., mutex_lock). However, certain lock operations such as the <write/read>_seqcount_begin or the rcu_read_lock_bh functions related to the *seqcount* and *RCU* (*Read, Copy, Update*) locking primitives, are always inlined in the kernel (module) code. Thus, LL's modified Lockdep cannot trace these locking primitives, when used in the binary-only IoT kernel modules. Fortunately, these operations are considerably less used in kernel modules (<10 invocations for both within files in the drivers/ directory), compared to the most popular locking operations (e.g., mutex_lock with 14,745 invocations within files in the drivers/ directory). The solution would be to fingerprint the use cases of inlined locking primitives in the module assembly and statically rewrite the binaries to insert the *Shadow Lockdep*'s APIs in these locations. However, besides implementing an accurate fingerprinting method, one would have to account with the typical challenges of static re-writing, such as ensuring the binary's normal operation.

Furthermore, as stated in Section 7.3.2, LL overestimates the number of lock classes assigned to spinlocks initialized by binary kernel modules compiled without Lockdep enabled. In Section 7.2.3 I detailed that LL relies on the lock name provided as an argument to the locking initialization functions to correctly assign lock classes to the locks used by the kernel modules. Every lock (e.g., mutex) has a dedicated function to initialize it dynamically (e.g., mutex_init), whether Lockdep is enabled or disabled in the kernel. However, spinlocks are the only locks that are initialized statically (i.e., through the macro

__RAW_SPIN_LOCK_UNLOCKED), when Lockdep is disabled (like in the IoT kernel modules). For this reason, LL assigns a unique lock class to every spinlock, even to the instances of the same lock. Due to this limitation, LL might exhibit false negatives (i.e., miss deadlock-based bugs), such as the double locking bugs (see Section 7.2.3). Finally, LL does not address cases of nested locks (i.e., locks that can be acquired more than once in the same context). The semantic information regarding lock nesting, required by Lockdep, is absent from the binary-only IoT kernel modules since they are not compiled with Lockdep enabled. However, in my experiments I have not seen any false positives involving the use of nested lock primitives. Despite these "shortcomings", LL successfully detects potential deadlocks in binary-only IoT kernel modules.

Chapter 8

Conclusions

As part of this thesis, I present and evaluate (holistic) re-hosting frameworks and firmware analysis techniques to detect diverse types of bugs and vulnerabilities in privileged binary Linux firmware code and ultimately improve the security of IoT firmware.

8.1 Scope of Contributions

The work discussed in this thesis analyzes Linux-based IoT firmware, specifically the open and closed-source IoT kernel modules contained within the file-system of firmware images. The IoT has been gaining a lot of popularity in recent years, with the number of Internet-connected devices estimated to reach the 40 billion mark by 2030 (IoT Analytics GmbH, 2024). Unfortunately, IoT is infamous for its weak security which is frequently exploited by attackers to conduct large scale attacks. The most notable incident involving IoT is the Mirai botnet DDoS attack (Antonakakis et al., 2017) which brought down high profile websites in 2016 (e.g., Netflix, Dyn, and Github) and became the catalyst for the emergence of various IoT botnets over the years. Despite the awareness raised by such attacks, IoT consistently fails to adhere to the appropriate and best security practices and design specifications (Microsoft, 2023). Thus, to address these security concerns, I focus on analyzing the (binary) firmware that runs on IoT devices.

8.2 Summary of Major Contributions

Prior work on firmware re-hosting and analysis primarily targets the user-level aspect of IoT firmware, while neglecting the equally important kernel-level aspect. In addition, existing firmware analysis solutions are limited in terms of their analysis scope. The majority of these works focus on identifying *Integrity* and *Confidentiality*, such as memory corruption bugs, while *Availability* bugs (e.g., deadlock-based bugs) are left unexplored. This thesis claims that to improve the security of IoT firmware, combining holistic firmware re-hosting with the thorough analysis of IoT firmware code to detect diverse bugs and vulnerabilities in open and closed-source privileged firmware code, is essential. To this end, in the course of this thesis, I introduce novel (holistic) firmware re-hosting techniques and methods to improve re-hosting fidelity, to enable the effective analysis of privileged Linux-based binary firmware code. In addition, I introduce an innovative firmware analysis technique that expands the scope of current approaches to also detecting deadlock-based bugs in binary-only privileged firmware code.

First, I propose a fully-automated firmware re-hosting framework that exposes Linux-based IoT kernel modules and drivers to downstream analysis, named `FirmSolo`. Unlike prior work that uses pre-compiled kernels to only re-host user-level firmware applications, `FirmSolo` relies on a novel hybrid binary reverse engineering technique, called K.C.R.E. to build dedicated custom kernels for each target firmware that can load the binary IoT kernel modules therein. In turn, existing downstream analysis systems can use `FirmSolo` to dynamically analyze these kernel modules for bugs and vulnerabilities. The evaluation of `FirmSolo` on 1,470 firmware images containing 56,688 kernel modules, shows that `FirmSolo` can successfully load 64% of these kernel modules. In addition, the fuzz-testing of 75 kernel modules reveals 19 previously unknown bugs in 11 distinct proprietary kernel modules.

Next, I introduce a method, called `FICD`, to objectively compare full-system re-hosting

frameworks based on emulation-specific metrics (e.g., code coverage). In addition, I present Pandawan, a holistic re-hosting and analysis framework that targets both the user and kernel-level firmware code as a unit, and implements the FICD technique. With FICD it is now possible to quantify the forward progress in firmware re-hosting, through the meaningful comparison of different full-system re-hosting approaches on their emulation capabilities. I use FICD to compare the Pandawan, Firmadyne, FirmAE, and FirmSolo re-hosting frameworks. The comparison of these re-hosting systems on a dataset of 1,520 firmware images shows that Pandawan outperforms Firmadyne, FirmAE, and FirmSolo, by executing up to 6% more user level programs and 21% more user code basic blocks, on average, than these systems. In addition, Pandawan loads 9% more IoT kernel modules and executes 26% more kernel module basic blocks on average than FirmSolo. Furthermore, Pandawan's holistic re-hosting and analysis identifies 16 bugs on 8 closed and 4 open source kernel modules.

I then propose a binary diffing framework, named FirmDiff, that aims to improve firmware re-hosting approaches and increase the effectiveness of the analysis on binary IoT kernel modules. FirmDiff compares the memory layout of data structures common to both the IoT kernel modules (and IoT kernel) and FirmSolo's custom kernels. With the help of FirmDiff an analyst can identify and address differences in the memory layout of data structures common between the IoT kernel modules and FirmSolo's custom kernels. Addressing these layout discrepancies improves the re-hosting fidelity and in turn the soundness of the analysis on binary IoT kernel modules. I evaluate FirmDiff on a dataset of 10 firmware images. FirmDiff identifies 37 misaligned data structures in FirmSolo's kernels for these images. After aligning the layout of 35 of these data structures, FirmSolo's refined kernels successfully load 28 previously crashing kernel modules.

I finally propose a firmware analysis framework that analyzes binary-only IoT kernel modules for deadlock-based bugs, named LL. LL introduces Shadow Lockdep, a subsystem

(complementary to Lockdep) and the `Shadow Lockdep Maps` (*SLoMs*), a shadow memory that stores Lockdep's metadata information, which enable Lockdep's deadlock detection analysis on binary targets. I evaluate `LL` on 80 binary-only IoT kernel modules that make use of locks. `LL`'s deadlock detection analysis discovers 6 potential deadlocks in 6 binary-only IoT kernel modules.

In conclusion, this thesis claims that a combination of holistic re-hosting and comprehensive analysis of IoT firmware enables the detection of diverse bugs and vulnerabilities in both open and closed-source privileged firmware code, thus improving the security of IoT firmware. To support this statement, this work presents the following:

- A fully-automated firmware re-hosting framework that exposes Linux-based privileged firmware code (in the form of kernel modules) to downstream analysis (Angelakopoulos et al., 2023).
- A method for the objective comparison of full-system re-hosting approaches based on their emulation capabilities, and a holistic firmware re-hosting and analysis framework that targets both the user and kernel-level aspect of Linux-based IoT firmware (Angelakopoulos et al., 2024b).
- A binary diffing framework which aids analysts to improve the fidelity of Linux-based IoT re-hosting frameworks and thus increase the soundness of the analysis (Angelakopoulos et al., 2024a).
- A dynamic firmware analysis framework that focuses on detecting deadlock-based bugs on binary-only IoT kernel modules.

References

Angelakopoulos, I., Stringhini, G., and Egele, M. (2023). FirmSolo: Enabling dynamic analysis of binary linux-based IoT kernel modules. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5021–5038, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/angelakopoulos.

Angelakopoulos, I., Stringhini, G., and Egele, M. (2024a). FirmDiff: Improving the Configuration of Linux Kernels Geared Towards Firmware Re-hosting. In *Proceedings of the Binary Analysis Research Workshop (BAR)*. https://www.ndss-symposium.org/ndss-paper/auto-draft-431/.

Angelakopoulos, I., Stringhini, G., and Egele, M. (2024b). Pandawan: Quantifying progress in linux-based firmware rehosting. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5859–5876, Philadelphia, PA. https://www.usenix.org/conference/usenixsecurity24/presentation/angelakopoulos.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., and Zhou, Y. (2017). Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

Bai, J.-J., Li, T., and Hu, S.-M. (2022). DLOS: Effective static detection of deadlocks in OS kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 367–382, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/bai.

Bars, N., Schloegel, M., Scharnowski, T., Schiller, N., and Holz, T. (2023). Fuzztruction: Using fault injection-based fuzzing to leverage implicit domain knowledge. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1847–1864, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/bars.

Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S. A., Stoller, S. D., Zadok, E., and Seyster, J. (2013). Adaptive runtime verification. In Qadeer, S. and Tasiran, S., editors, *Runtime Verification*, pages 168–182, Berlin, Heidelberg.

Bell Labs (2012). Cscope. http://cscope.sourceforge.net/.

Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

Brotherston, J., Brunet, P., Gorogiannis, N., and Kanovich, M. (2022). A compositional deadlock detector for android java. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 955–966. https://doi.org/10.1109/ASE51524.2021.9678572.

Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA.

Cai, Y., Meng, R., and Palsberg, J. (2020). Low-overhead deadlock prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1298–1309, New York, NY, USA. https://doi.org/10.1145/3377811.3380367.

Cai, Y., Yao, P., Ye, C., and Zhang, C. (2023). Place your locks well: Understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3727–3744, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao.

Cai, Y., Ye, C., Shi, Q., and Zhang, C. (2022). Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 784–796, New York, NY, USA. https://doi.org/10.1145/3540250.3549110.

Carvalho, A. (2009). Pahole. https://lwn.net/Articles/335942/.

Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, page 380–394, USA. https://doi.org/10.1109/SP.2012.31.

Chen, D. D., Egele, M., Woo, M., and Brumley, D. (2016). Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf.

Chen, P. and Chen, H. (2018). Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*.

Chen, W., Wang, Y., Zhang, Z., and Qian, Z. (2021). Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 749–763, New York, NY, USA. https://doi.org/10.1145/3460120.3484564.

Chipounov, V., Kuznetsov, V., and Candea, G. (2011). S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 265–278, New York, NY, USA. https://doi.org/10.1145/1950365.1950396.

Clang Project (2025a). AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html.

Clang Project (2025b). Clang C Language Family Frontend for LLVM. https://clang.llvm.org/.

Claroty (2022). IoT Vulnerability Disclosures Grew 57% from 2H 2021 to 1H 2022. https://claroty.com/press-releases/iot-vulnerability-disclosures-grew-57-percent-from-2h21-to-1h22.

Clearbluejar (2023). Ghidriff. https://github.com/clearbluejar/ghidriff.

Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. (2020). HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., and Vigna, G. (2017). DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA. https://doi.org/10.1145/3133956.3134069.

Corteggiani, N., Camurati, G., and Francillon, A. (2018). Inception: System-Wide security testing of Real-World embedded systems software. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 309–326, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani.

Costin, A., Zarras, A., and Francillon, A. (2016). Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 437–448, New York, NY, USA. https://doi.org/10.1145/2897845.2897900.

Craig, L., Fasano, A., Ballo, T., Leek, T., Dolan-Gavitt, B., and Robertson, W. K. (2021). PyPANDA: Taming the PANDAmonium of Whole System Dynamic Analysis. In *Proceedings of the Binary Analysis Research Workshop (BAR)*.

D. Ganesan (2017). Lack of Standardization can Halt the IoT Juggernaut. https://www.dqindia.com/lack-of-standardization-can-halt-the-iot-juggernaut/.

Davidson, D., Moench, B., Ristenpart, T., and Jha, S. (2013). FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, Washington, D.C. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson.

Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., and Whelan, R. (2015). Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA. https://doi.org/10.1145/2843859.2843867.

Ellzey, M., Holland, A., and Lindner, R. (2022). The QNapping of QNAP Devices. https://censys.com/blog/the-qnapping-of-qnap-devices.

Elwel, P. (2018). Dwc_otg deadlock. https://github.com/raspberrypi/linux/commit/d93cf8be07b1ae31e2d6f2ba76731c93380ede8a.

Engler, D. and Ashcraft, K. (2003). Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 237–252, New York, NY, USA. https://doi.org/10.1145/945445.945468.

Eslamimehr, M. and Palsberg, J. (2014). Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 353–365, New York, NY, USA. https://doi.org/10.1145/2635868.2635918.

Fasano, A., Ballo, T., Muench, M., Leek, T., Bulekov, A., Dolan-Gavitt, B., Egele, M., Francillon, A., Lu, L., Gregory, N., Balzarotti, D., and Robertson, W. (2021). SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 687–701, New York, NY, USA. https://doi.org/10.1145/3433210.3453093.

Feng, Q., Prakash, A., Wang, M., Carmony, C., and Yin, H. (2016). ORIGEN: Automatic Extraction of Offset-Revealing Instructions for Cross-Version Memory Analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 11–22, New York, NY, USA. https://doi.org/10.1145/2897845.2897850.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi.

Fioraldi, A., Maier, D. C., Zhang, D., and Balzarotti, D. (2022). LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1051–1065, New York, NY, USA. https://doi.org/10.1145/3548606.3560602.

FreeRTOS (2023). FreeRTOS: Real-time operating system for microcontrollers. https://www.freertos.org/index.html.

GeeksforGeeks (2025). Coffman conditions. https://www.geeksforgeeks.org/operating-systems/conditions-for-deadlock-in-operating-system/.

GNU Project (2024). GDB The GNU Project Debugger. https://www.sourceware.org/gdb/.

Google (2016). OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.

Google (2023). Syzkaller. https://github.com/google/syzkaller.

Hernandez, G., Fowze, F., Tian, D. J., Yavuz, T., and Butler, K. R. (2017). FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2245–2262, New York, NY, USA. https://doi.org/10.1145/3133956.3134050.

Hex-Rays (2024). IDAPro. https://hex-rays.com/ida-pro/.

IoT Analytics GmbH (2024). State of IoT 2024. https://iot-analytics.com/wp/wp-content/uploads/2024/09/INSIGHTS-RELEASE-Number-of-connected-IoT-devices-vf.pdf.

Joshi, P., Naik, M., Sen, K., and Gay, D. (2010). An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

Junnila, A. (2023). How IoT Works – Part 4: User Interface. https://trackinno.com/iot/how-iot-works-part-4-user-interface/.

Kammerstetter, M., Platzer, C., and Kastner, W. (2014). Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 329–340, New York, NY, USA. https://doi.org/10.1145/2590296.2590301.

Keniston, J. (2021). Kprobes Documentation. https://www.kernel.org/doc/Documentation/kprobes.txt.

KernelNewbies (2017). How does the kernel implements Hashtables? https://kernelnewbies.org/FAQ/Hashtables.

Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., and Kim, Y. (2020). Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 733–745, New York, NY, USA. https://doi.org/10.1145/3427228.3427294.

Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. (2018). Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA. https://doi.org/10.1145/3243734.3243804.

Koscher, K., Kohno, T., and Molnar, D. (2015). SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher.

Lee, J., Avgerinos, T., and Brumley, D. (2011). TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., Lu, K., and Wang, T. (2021). UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei.

Lin, Z., Zhang, X., and Xu, D. (2010). Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, CERIAS '10, West Lafayette, IN.

Litux (2005). Deadlocks. https://litux.nl/mirror/kerneldevelopment/0672327201/ch08lev1sec3.html.

Liu, Q., Zhang, C., Ma, L., Jiang, M., Zhou, Y., Wu, L., Shen, W., Luo, X., Liu, Y., and Ren, K. (2022). 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*, ASE '21, page 792–804. https://doi.org/10.1109/ASE51524.2021.9678653.

Lu, S., Kuang, X., Nie, Y., and Lin, Z. (2020). A Hybrid Interface Recovery Method for Android Kernels Fuzzing. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 335–346.

Luo, C., Meng, W., and Li, P. (2023). SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2693–2707.

Magnusson, U. (2021). Kconfiglib. https://github.com/ulfalizer/Kconfiglib.

Malwarebytes (2025). Stuxnet. https://www.malwarebytes.com/stuxnet.

Microsoft (2023). Microsoft Digital Defense Report 2023. https://www.microsoft.com/en-us/security/security-insider/microsoft-digital-defense-report-2023.

Molnar, I. and Van de Ven, A. (2025). Runtime locking correctness validator. https://docs.kernel.org/locking/lockdep-design.html.

Muench, M., Stijohann, J., Kargl, F., Francillon, A., and Balzarotti, D. (2018). What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Muller, S. K. (2024). Language-Agnostic Static Deadlock Detection for Futures. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, page 68–79, New York, NY, USA. https://doi.org/10.1145/3627535.3638487.

Nagy, L. (2025). Build EAR. https://github.com/rizsotto/Bear.

NCC Group Plc (2017). TriforceAFL: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

Newsham, T. (2017). TriforceLinuxSyscallFuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

NSA Research Directorate (2025). Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra.

Offensive Security (2021). Exploit Database. https://www.exploit-db.com/.

OpenWrt Project (2021). Openwrt Backfire. https://archive.openwrt.org/backfire/10.03.1/.

Openwrt Project (2021). OpenWrt Project. https://openwrt.org/.

Pagani, F. and Balzarotti, D. (2022). Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security*, 25:1–26.

Pailoor, S., Aday, A., and Jana, S. (2018). MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor.

Petazzoni, T. (2021). Buildroot. https://buildroot.org/.

Project, D.-W. (2021). DD-WRT. https://dd-wrt.com/.

Pustogarov, I., Wu, Q., and Lie, D. (2020). Ex-vivo Dynamic Analysis Framework for Android Device Drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105.

Qiling (2022). Qiling Framework. https://github.com/qilingframework/qiling.

Radare (2024). Radare2. https://rada.re/n/.

Rapid7 (2023). Metasploit Framework. https://www.metasploit.com/.

Renzelmann, M. J., Kadav, A., and Swift, M. M. (2012). SymDrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, Hollywood, CA. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann.

Rupprecht, T., Chen, X., White, D. H., Boockmann, J. H., Lüttgen, G., and Bos, H. (2017). Dsibin: identifying dynamic data structures in c/c++ binaries. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, page 331–341.

Satopaa, V., Albrecht, J., Irwin, D., and Raghavan, B. (2011). Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, page 166–171, USA. https://doi.org/10.1109/ICDCSW.2011.20.

Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., and Abbasi, A. (2022). Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1239–1256, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.

Seyster, J., Radhakrishnan, P., Katoch, S., Duggal, A., Stoller, S. D., and Zadok, E. (2011). Redflag: a framework for analysis of Kernel-level concurrency. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'11, page 66–79, Berlin, Heidelberg.

Shen, Z., Roongta, R., and Dolan-Gavitt, B. (2022). Drifuzz: Harvesting bugs in device drivers from golden seeds. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1275–1290, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/shen-zekun.

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. (2015). Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157.

Slowinska, A., Stancescu, T., and Bos, H. (2011). Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Socała, A. and Cohen, M. (2016). Automatic profile generation for live Linux Memory analysis. *DFRWS EU 2016: Digital Forensics Research Conference Europe*, 16:S11–S24. https://www.sciencedirect.com/science/article/pii/S1742287616000050.

Solutions, S. (2024). Understanding the Wilcoxon Signed Rank Test. https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/how-to-conduct-the-wilcox-sign-test/.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Syzbot (2018a). Possible deadlock in bond_get_stats. https://syzkaller.appspot.com/bug?id=e6b73430f9405afe79f46530dfa138e33ca74778.

Syzbot (2018b). Possible deadlock in ppp_dev_uninit. https://syzkaller.appspot.com/bug?extid=367889b9c9e279219175.

Syzbot (2020). Possible deadlock in mon_bin_vma_fault. https://syzkaller.appspot.com/bug?extid=56f9673bb4cdcbeb0e92.

Syzbot (2025). Syzbot fixed bugs. https://syzkaller.appspot.com/upstream/fixed.

Tay, H. J., Zeng, K., Vadayath, J. M., Raj, A. S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z. L., Dong, F., Smith, Z., Doupé, A., Bao, T., Shoshitaishvili, Y., and Wang, R. (2023). Greenhouse: Single-Service rehosting of Linux-Based firmware binaries in User-Space emulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5791–5808, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/tay.

Vector 35 (2024). Binary Ninja. https://binary.ninja/.

Verkada (2021). Summary: March 9, 2021 Security Incident Report. https://www.verkada.com/security-update/report/.

Vetterl, A. and Clayton, R. (2019). Honware: A Virtual Honeypot Framework for Capturing CPE and IoT Zero Days. In *2019 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–13.

Vyukov, D. (2025). KCOV: code coverage for fuzzing. https://docs.kernel.org/dev-tools/kcov.html.

Wikipedia (2024). Cosine Similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

XFS team (2025). Xfstests. https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/.

Yan, S., Wu, C., Li, H., Shao, W., and Jia, C. (2020). Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 598–609, New York, NY, USA. https://doi.org/10.1145/3320269.3384736.

Ye, C., Cai, Y., and Zhang, C. (2024). When threads meet interrupts: Effective static detection of Interrupt-Based deadlocks in linux. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6167–6184, Philadelphia, PA. https://www.usenix.org/conference/usenixsecurity24/presentation/ye.

Zaddach, J., Bruno, L., Francillon, A., and Balzarotti, D. (2014). AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Zalewski, M. (2017). American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

Zhang, M. and Sekar, R. (2013). Control Flow Integrity for COTS Binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.-C., Kwon, Y., Aafer, Y., and Zhang, X. (2021). OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832.

Zhao, W., Lu, K., Wu, Q., and Qi, Y. (2023). Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. (2019). FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, Santa Clara, CA. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

Zheng, Y., Li, Y., Zhang, C., Zhu, H., Liu, Y., and Sun, L. (2022). Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 417–428, New York, NY, USA. https://doi.org/10.1145/3533767.3534414.

Zhou, J., Silvestro, S., Liu, H., Cai, Y., and Liu, T. (2017). Undead: detecting and preventing deadlocks in production software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, page 729–740.

CURRICULUM VITAE