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

Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele
Boston University
{jaggel, gian, megele}@bu.edu

Abstract

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 have serious security vulnerabilities, and the risk linked to kernel vulnerabilities is particularly high, as these can lead to full system compromise. However, previous work only focuses on the user space component of embedded firmware. In this paper, we present *Firm*ware *Sol*ution (*FirmSolo*), a system designed to incorporate the kernel space into firmware analysis. *FirmSolo* features the *Kernel Configuration Reverse Engineering* (K.C.R.E.) process that leverages information (i.e., exported and required symbols and version magic) from the kernel modules found in firmware images to build a kernel that can load the modules within an emulated environment. This capability allows downstream analysis to broaden their scope into code executing in privileged mode.

We evaluated *FirmSolo* on 1,470 images containing 56,688 kernel modules where it loaded 64% of the kernel modules. To demonstrate how *FirmSolo* aids downstream analysis, we integrate it with two representative analysis systems; the TriforceAFL kernel fuzzer and Firmadyne, a dynamic firmware analysis tool originally devoid of kernel mode analysis capabilities. Our TriforceAFL experiments on a subset of 75 kernel modules discovered 19 previously-unknown bugs in 11 distinct proprietary modules. Through Firmadyne we confirmed the presence of these previously-unknown bugs in 84 firmware images. Furthermore, by using *FirmSolo*, Firmadyne confirmed a previously-known memory corruption vulnerability in five different versions of the closed-source Kcodes' *NetUSB* module across 15 firmware images.

1 Introduction

The Internet of Things (IoT) is predicted to grow to 41.6 billion devices by 2025 [41]. Unfortunately, most IoT vendors hail from the electronics side of device manufacturing and lack security awareness on the software side. This has led to the untenable situation of large-scale deployments of IoT gadgets that feature known and unknown software vulnerabilities. The negative impact of this situation was impressively demonstrated by a distributed denial of service attack orchestrated by the creators of the Mirai botnet [1]. In 2016, Mirai compromised hundreds of thousands of DVRs and home routers to launch a DDoS attack against Internet mainstays like Netflix, Dyn, and GitHub. Mirai relied on the insecure practice of configuring all devices of a given product line with a default username and password. More recent IoT threats [18] upped the ante and started to leverage exploits against known vulnerabilities in prolifically deployed and Internet-reachable gadgets. Given the poor security posture of the devices that comprise the IoT, it is vital that the community devises techniques that enable the automated vulnerability analysis of the firmware powering these devices.

Prior work (e.g., [11, 34, 37]) in this area provides a solid foundation towards this goal. However, the static analysis techniques presented in these papers, while scalable, suffer from large numbers of false positives. Additionally, prior work on dynamic analysis features its own limitations. First, a set of previous research [20, 44] provides highly precise analysis capabilities that unfortunately do not scale to the size and variety of the IoT, since these systems require access to a physical device whose software should be analyzed. A second category of prior work are scalable analysis systems (e.g., [5, 7, 10, 42]), which are limited in the breadth of their analysis capabilities. For example, Zaddach et al. [10] analyze the security of Web applications frequently found on IoT devices. Firmadyne [5] broadens that scope to include other user level components running on the same IoT device.

Existing systems, however, cannot analyze privileged (i.e., kernel level) code; neither the kernel proper nor any of the 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. To make matters worse, IoT devices frequently feature binary-only proprietary modules, making it challenging for third party analysts to audit them for security vulnerabilities. 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.

To fill this gap, we propose *FirmSolo*, the first scalable software system to automatically expose IoT privileged kernel-level code distributed in the form of Linux kernel modules to downstream analysis. Crucially, *FirmSolo* does not require, but can take advantage of, the availability of module source code. As such, it supports open-source as well as proprietary binary-only modules. A prerequisite to any dynamic analysis of a kernel module is that the module is successfully loaded by the kernel, a process guarded by the kernel's module loading facility (see Section 2.3). All but the most trivial modules will require interaction with the kernel proper through kernel symbols; functions or kernel-defined data structures. 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. 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 kernel's configuration such that a 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 we refer to as **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, we integrated *FirmSolo* with two representative analysis systems; the TriforceAFL [26] kernel fuzzer and the Firmadyne [5] 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. To the best of our knowledge, *FirmSolo* is the first system that makes the binary kernel modules found in IoT firmware amenable to downstream dynamic analysis systems. In summary, this paper makes the following contributions:

- We propose a novel automated hybrid reverse engineering approach (K.C.R.E.) of the IoT kernel modules distributed in a firmware image to infer the configuration of the corresponding Linux kernel.
- We present *FirmSolo*, a fully-automated prototype implementation of this approach. We assess *FirmSolo*'s efficacy by building kernels for 1,470 firmware images, that rely on 77 different kernel versions. These kernels can successfully load 64% of the 56,688 kernel modules contained in their corresponding firmware images.
- To demonstrate *FirmSolo*'s utility to enable downstream analyses to assess firmware images more holistically, we modify the TriforceAFL kernel fuzzer to support firmware kernel module fuzzing. With *FirmSolo*'s help, TriforceAFL analyzed 75 modules from our set and detected 19 previously unknown bugs in 11 distinct proprietary modules, across 10 distinct products.
- To analyze the modules in our set for bugs at scale, we extend the open-source Firmadyne system and evaluate it over the set of 1,470 images. *FirmSolo*'s kernels load 69% of the 18,018 kernel modules used by the images and confirm a previously-known vulnerability in five versions of a closed-source proprietary module (i.e., KCodes' `NetUSB.ko`) spanning 15 IoT firmwares as well as the presence of the previously-unknown bugs identified by TriforceAFL across 84 images.

We responsibly disclosed our findings to all the affected vendors and will release the implementation of *FirmSolo* under an open-source license.

2 Background

In this section we provide background information on the Linux kernel's configuration, kernel module loading procedure, and the assumptions our system relies on.

2.1 Linux Kernel

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 5 in Appendix C 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>`.

2.2 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). Thus, *FirmSolo* has to ensure that any external symbol used by a module is exported either by another module or the kernel proper. 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. *FirmSolo* makes use of this information, if present, in an image's kernel binary (see Section 3.1).

2.3 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`). In Section 4.2, we highlight the importance of steps 3, 5 and 6 to *FirmSolo*.

2.4 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, we 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 4.2) 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.

3 System Overview

In this section we 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 we 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 1 illustrates *FirmSolo* as a fully-automated three-stage process: 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 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 The Downstream Analysis stage leverages the kernel (K_{FS}) from the above step to subject the modules found in image I to various security analyses. As representative examples of such analyses we 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*

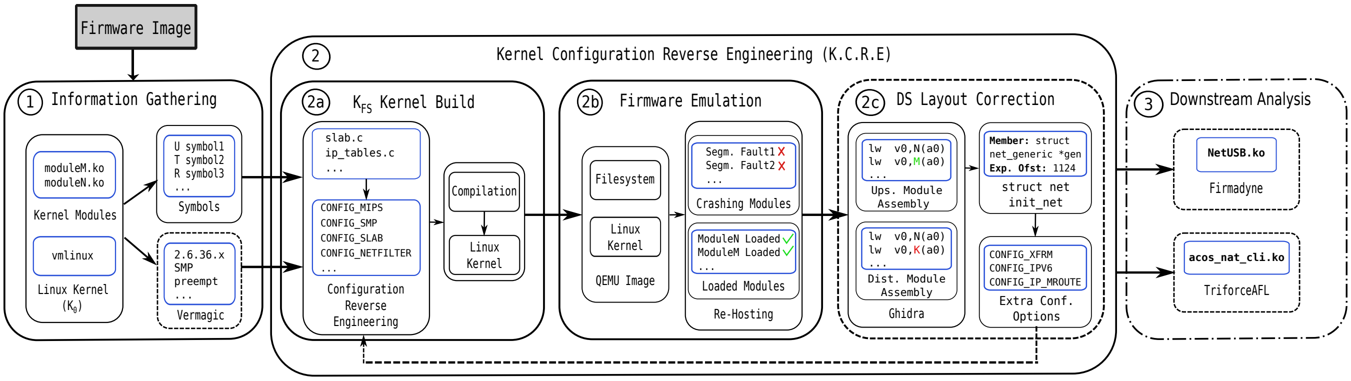


Figure 1: System overview of *FirmSolo*: The dashed boxes indicate that a step or information are optional. Also while stage 3 is not part of *FirmSolo*, we feature two representative downstream analysis.

in a firmware image, we 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 our assumptions in Section 2.4, a *subset of open-source modules* in $\mathbb{D}_{KM}$ will have corresponding modules in $\mathbb{U}_{KM}$, an insight we will exploit to align the layout of data structures in Section 3.2.

3.1 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. We provide more detailed information about the metadata collected by *FirmSolo* in Section 4.1. The metadata collected in this stage are used in stage ② to configure and compile the K_{FS} kernel.

3.2 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. We 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], as these are well-supported by QEMU [2].

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)

[1] Choosing a real-SoC will yield a kernel that runs only on said SoC, but will not be supported by QEMU (see Section 2.4)

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

it can prevent the loading of additional modules, if they rely on symbols exported by the offending module d. We 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 us 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 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 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 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 3.3). 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 4.2). 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.

3.3 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. We demonstrate this aspect by integrating *FirmSolo* with two representative analysis systems; TriforceAFL [26] and Firmadyne [5]. 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 Implementation

We next discuss the implementation details of *FirmSolo*.

4.1 Information Gathering

Stage ① consumes the extracted file-system of the original firmware image I, collects the set of $\mathbb{D}_{KM}$ kernel modules from I 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 2.2, 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 I. 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`.

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 2.2 and Appendix B). 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).

4.2 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 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` [21] 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` [25] 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.

```
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 2: `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.

Listing 2 shows an example for the symbol `malloc_sizes`. First *FirmSolo* detects 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

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

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

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 I. 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 in Appendix B), since the vermagic options might not have any correspondence in these symbols. Thus, we 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 [29] 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 2.3), *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 5.2), 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).

(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 we 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 [15]. Specifically, *FirmSolo* uses a Ghidra [13] 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 3 in Appendix D, showcases the variable `def-use` chain extraction process in the case of the $\mathbb{C}_{KM}$ module `ifb.ko`. *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 instructions 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 3c). 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 we 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 our dataset (see Section 5.1) 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 2.3). 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 we 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 Firmware Analysis

To demonstrate the utility of *FirmSolo*, we provide two representative use cases of dynamic analysis tools that benefit from *FirmSolo*; TriforceAFL [26] and Firmadyne [5].

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, we 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, we 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. We provide more information about our fuzzing strategy in Appendix A.

Firmadyne. Firmadyne relies on a single modified kernel version and heavily relies on the Kprobes [17] 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 our 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.

5 Evaluation

In this section we 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, we evaluate *FirmSolo* along three dimensions.

Q1 *Efficacy:* What is the contribution of the first two stages of *FirmSolo* to successfully loading IoT kernel modules (§ 5.2)?

Q2 *Precision:* How close is the kernel configuration that *FirmSolo* infers to a ground-truth kernel configuration (§ 5.3)?

Q3 *Broader Applicability:* How does *FirmSolo* aid downstream systems to analyze kernel modules (§ 5.4)?

In the following, we first introduce the dataset that we use to evaluate *FirmSolo*. We then present the experiments that we conduct to evaluate our approach in detail.

5.1 Evaluation Dataset

Our dataset consists of 8,737 images used by Firmadyne and shared with us by its authors. Additionally we 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, we were able to use 15 for the evaluation, as for 33 images we 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 our 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. We also refined the Firmadyne dataset and removed 4,020 images belonging to open-source projects such as OpenWRT [31] and DD-WRT [30] as the repetitive analysis of essentially thousands of duplicates would distort the results. Furthermore, we 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, our evaluation dataset consists of 1,470 firmware images, where 969 target MIPS and 501 target ARM platforms. The firmware images in our dataset span across 77 unique kernel versions, between version `2.6.18` and version `4.4.198`. The compiler tool chains we use to compile the K_{FS} kernels in our 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.

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 1: Symbol, C-source file and option information collected by *FirmSolo*.

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 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**

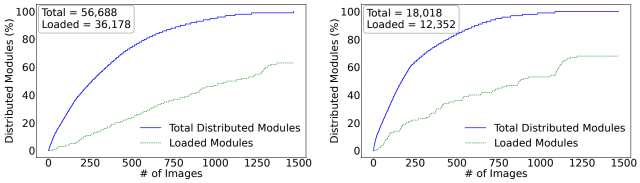
Table 3: Module dataset categorization.

Finally, Table 3 summarizes the different categories of the kernel modules that comprise our dataset. We 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). We consider miscellaneous all the modules whose category is unknown.

5.2 Efficacy

In this section we 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 4.1, these originate from the ELF headers of the kernel modules and the `KALLSYMS` entry of K_0 (if available). In our 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 1 summarizes the symbol information



(a) *FirmSolo* firmware emulation step results

(b) Firmadyne emulation results

Figure 2: Cumulative Distribution Functions representing the modules in $\mathbb{D}_{KM}$ loaded by *FirmSolo* and Firmadyne, respectively.

obtained through the Information Gathering stage ①.

2) K.C.R.E. We 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.

Symbols to Configuration Options. On average, *FirmSolo* maps the 8,557 undefined symbols of each image I 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 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 we first tested Qiling [33], a binary emulation framework with kernel module support, against a random sample of ten kernel modules from our 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. We contacted the Qiling developers and the reply indicated that MIPS and ARM are not well tested. In light of these events we consider improving Qiling outside the scope of this work.

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: 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*.

Instead, to provide a complete evaluation of *FirmSolo* and better showcase the contribution of K.C.R.E. we test our system under four scenarios with different settings. In detail, we 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 2 clearly show that with K.C.R.E. our system is substantially more successful as it loads 85% more modules than Experiment 2.

Specifically, *FirmSolo* loads 36,178 (64%) of the 56,688 $\mathbb{D}_{KM}$ modules in our dataset, while implementing the substitution mechanism. Figure 2a shows the cumulative distribution function over the number of successfully loaded $\mathbb{D}_{KM}$ modules when applying *FirmSolo* on our dataset. As illustrated in Table 3, these modules fall into various categories, 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, we do not count as success the 4,601 module substitutions that *FirmSolo* implements in the course of the experiments, in our results. The substitutions account for 8% of the total $\mathbb{D}_{KM}$ modules in our 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 2.3 (4,290 substitutions). Crucially, the substitution system enabled the loading of 5,290 (9%) additional $\mathbb{D}_{KM}$ modules in our set which would otherwise not load and not be available to downstream analysis.

Failed Cases. The remaining 20,510 modules did not load in our experiments, due to various reasons. Our analysis showed that the majority of these modules (64 %) fail to pass the symbol resolution step (see Section 2.3), due to required symbols not exported by the kernel proper (see Section 5.2). We present these failure reasons in Table 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 our experiments and based on our manual analysis it is specific to modules compiled from an open source distribution (e.g., OpenWRT), that is loaded in our upstream kernels. Since we removed all the OpenWRT images from our dataset, we 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 our 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).

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 our set with a total of 3,265 (6%) modules. We do not consider these cases as a limitation of our 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 our generic file-system. We also include these cases in Table 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 5.4 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 our 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 our `kmem_cache_alloc` heuristic and `module` as our 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. Our 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 we found out that the vendors have modified their kernels' source code regarding the check, thus the check fails on our 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 we do not consider these crashes a limitation of *FirmSolo*. We 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)). Our 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`).

5.3 Precision

To assess the precision of the K.C.R.E. process when inferring the kernel configuration, we compare its results against a ground-truth firmware image built for the open source Backfire OpenWRT release [27].[4] The open source nature of OpenWRT means we 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, we 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.

We measure *FirmSolo*'s precision along two dimensions and perform each measurement without the use of the optional

[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`

KALLSYMS. First, on a macro-scale we quantify how many of the 530 modules can be loaded by the K_{FS} kernel produced by *FirmSolo*. The K_{FS} 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 4.2). 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 we assess whether *FirmSolo* sets the *relevant* configuration options correctly. In this context, we 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, we are 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).

5.4 Broader Applicability

To demonstrate the utility of *FirmSolo*, this section discusses our experience with two representative downstream analysis systems enhanced by *FirmSolo*: TriforceAFL [26] and Firmadyne [5]. Note that *FirmSolo*'s novel capability is widely applicable to a variety of other analysis systems beyond these examples (e.g., PANDA [14] or FirmAE [19]).

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

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 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.

We fuzz our 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 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). We manually verify each bug by detecting the source of each crash in Ghidra. We 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. We 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. We could confirm that 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. Our 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 we 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 we conservatively label all these crashes as false positives. Note that we do not consider 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, we responsibly disclosed all previously unknown bugs we detected to the affected vendors. During the disclosure 7 vendors, (all except Tenda) corresponded directly with us requesting a PoC and, so far, 2 (Supermicro and Asus) confirmed our findings.

2) Firmadyne. To assess the utility of *FirmSolo* for Firmadyne, we run the 1,470 images from our evaluation data set in Firmadyne. Recall that in the evaluation in Section 5.2 *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 our 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 2b shows the cumulative distribution function over the number of $\mathbb{D}_{KM}$ modules loaded in the Firmadyne experiments.

We also encountered 141 additional cases during our 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, our 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. We leverage this capability and launch 10 publicly available proof of concept exploits from ExploitDB [36], targeting the modules contained in our dataset (full list in Table 6 in the Appendix E). Additionally, we use Firmadyne to assess which 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 our 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 our 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 were not loaded by the startup scripts in their firmware images.

In summary, our 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 vulnerabilities.

6 Limitations

As any automated solution, *FirmSolo* is subject to a set of limitations. First, *FirmSolo* currenlty 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. We 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.

7 Related Work

To best of our knowledge *FirmSolo* is the first system to enable the loading of binary kernel modules at scale in the embedded systems re-hosting and dynamic analysis landscape.

Kernel Configuration Recovery. Socala et al. [40] and Pagani et al. [28] leveraged 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. Also, as stated in Section 2.4 the available original firmware kernels cannot be emulated by state-of-the-art emulators like QEMU. *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.

Firmware Analysis. Chen et al. [5] and Kim et al. [19] 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. [42] 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. In contrast *FirmSolo*, produces kernels that are able to load said modules and enable downstream analysis on both user and privileged firmware code.

Other systems, such as AVATAR [44], SURROGATES [20], and Inception [9] implemented a hybrid approach, where an emulator (driven by a symbolic execution engine) and an actual device are combined, to conduct dynamic analysis on firmware binaries. With the presence of the target device these systems were able to overcome the peripheral availability problem, encountered in most full system emulation approaches. However, not all devices provide the necessary debugging interfaces (e.g., JTAG) for integration with these systems. Importantly, the reliance of these systems on the physical IoT hardware inherently renders them non-scalable.

Symbolic Execution. Static analysis and symbolic execution [12, 16, 34, 37, 38] are popular techniques used for analyzing Linux based drivers and binary firmware. Static analysis is essential to guide symbolic execution engines [3, 4, 6], 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 full-system emulation and fully automated analysis pipeline make it applicable to many different types of devices and their privileged firmware code.

Data Structure Inference. Dynamic analysis systems, such as REWARDS [23] , HOWARD [39] and DSIbin [35] mostly rely on memory access patterns with the help of symbolic execution to deduce the layout of data structures. However, since the original kernels cannot be emulated by state-of-the-art emulators such as QEMU, these systems are not applicable to our dataset. Static analysis systems, such as OSPREY [46] and hybrid systems such as TIE [22] and ORIGEN [15] rely on data and control flow analyses to recover the layout of data structures in binaries. 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 K_{FS}. However, not even ORIGEN can completely replace step $\textcircled{2c}$ since its dynamic analysis component, essential to ORIGEN's functionality, suffers from the dynamic analysis limitations stated previously.

Firmware Fuzzing. Pustogarov et al. developed EASIER [32], a system directed towards Android device drivers. Through abstraction techniques and heuristics EASIER is capable of loading custom kernel modules on an instrumented Android kernel, and in combination with a system call fuzzer, EASIER is able to analyze Android kernel drivers. Currently, EASIER supports only modules in the Android hardware drivers category for three kernel versions. Furthermore, EASIER is mainly designed to analyze kernel drivers individually and not at scale. To load and analyze kernel drivers, EASIER configures its kernels with basic `.config` files and uses function abstraction techniques (i.e., replacing symbols required by modules with empty stubs) to satisfy the kernel modules' symbol dependencies. This implementation imposes the following shortcomings. First, as we show in the evaluation of stage $\textcircled{2}$ in Section 5.2 generically configured kernels load significantly fewer kernel modules than *FirmSolo*'s kernels. Second, while the abstraction techniques used by EASIER are effective for loading kernel modules in isolation, they fail to capture the inter-module and kernel-module dependencies (e.g., symbol exporting and sharing). Factoring in these dependencies is essential for modules to load at scale into the kernel and also function properly. Thus, the adoption of an analysis process similar to *FirmSolo*'s (i.e., K.C.R.E.) would be necessary for EASIER to overcome the above limitations and expand to other module types and kernels. On the contrary, *FirmSolo* is designed to be generic enough to support a wide variety of kernel versions and modules at scale.

Other works such as FirmAFL [47] rely on a hybrid form of firmware emulation (user and full system modes) in conjunction with fuzzing [45] to analyze a firmware image. Works like HALucinator [7] use Hardware Abstraction Layers (HALs) to better imitate peripheral behavior during emulation, along with a fuzzer [43], to perform dynamic analysis on embedded firmware. As with the majority of the other approaches, FirmAFL also takes into consideration only the user space aspects of the firmware images. Also in the case of HALucinator the analyst has to pinpoint all the necessary HALs in order for the model to work correctly, thus impacting the scalability of the framework. In contrast, *FirmSolo* does not suffer from scalability limitations, due to manual intervention and supports both user and privileged code analysis.

8 Conclusion

In this paper we presented *FirmSolo*, a framework that makes privileged firmware code in the form of kernel modules available to downstream analysis. By implementing the Kernel Configuration Reverse Engineering technique *FirmSolo* builds a kernel for each firmware image that is capable of loading its kernel modules. To illustrate *FirmSolo*'s utility, we demonstrated two use cases; TriforceAFL and Firmadyne finding previously-known and unknown bugs in binary kernel modules.

9 Acknowledgements

We would like to thank our anonymous reviewers and the shepherd for their valuable comments and feedback. This research was enabled by the National Science Foundation under grants CNS-1942793 and CNS-2127232 and by support from the MIT Lincoln Laboratory under the IGLOO project.

References

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

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

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[4] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2012.

[5] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 2011.

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

[8] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

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

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

[11] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[12] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the USENIX Security Symposium*, 2013.

[13] NSA's Research Directorate. Ghidra. https://ghidra-sre.org/, 2021.

[14] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)*, 2015.

[15] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[16] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[17] J. Keniston. Kprobes Documentation. https://www.kernel.org/doc/Documentation/kprobes.txt, 2021.

[18] S. Khandelwal. Mirai Variant. https://thehackernews.com/2019/03/mirai-botnet-enterprise-security.html, 2019.

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

[20] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[21] Bell Labs. Cscope. http://cscope.sourceforge.net/, 2012.

[22] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: principled reverse engineering of types

in binary programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[23] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the CERIAS Annual Information Security Symposium*, 2010.

[24] S. Lu, X. Kuang, Y. Nie, and Z. Lin. A hybrid interface recovery method for android kernels fuzzing. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2020.

[25] U. Magnusson. Kconfiglib. https://github.com/ulfalizer/Kconfiglib, 2021.

[26] NCC Group Plc. TriforceAFL: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2017.

[27] OpenWrt Project. Openwrt backfire. https://archive.openwrt.org/backfire/10.03.1/, 2021.

[28] Fabio Pagani and Davide Balzarotti. Autoprofile: Towards automated profile generation for memory analysis. *ACM Transactions on Privacy and Security*, 2021.

[29] T. Petazzoni. Buildroot. https://buildroot.org/, 2021.

[30] DD-WRT Project. DD-WRT. https://dd-wrt.com/, 2021.

[31] Openwrt Project. OpenWrt Project. https://openwrt.org/, 2021.

[32] I. Pustogarov, Q. Wu, and D. Lie. Ex-vivo dynamic analysis framework for android device drivers. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[33] Qiling. Qiling framework. https://github.com/qilingframework/qiling, 2022.

[34] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In *Proceedings of the USENIX Security Symposium*, 2012.

[35] Thomas Rupprecht, Xi Chen, David H. White, Jan H. Boockmann, Gerald Lüttgen, and Herbert Bos. Dsibin: Identifying dynamic data structures in c/c++ binaries. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[36] Offensive Security. Exploit Database. https://www.exploit-db.com/, 2021.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

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

[39] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[40] Arkadiusz Socała and Michael Cohen. Automatic profile generation for live linux memory analysis. *Digital Investigation*, 2016.

[41] IDC Corp. USA. Iot Growth. https://www.idc.com/getdoc.jsp?containerId=prUS45213219, 2019.

[42] A. Vetterl and R. Clayton. Honware: A virtual honeypot framework for capturing cpe and iot zero days. In *Proceedings of the APWG Symposium on Electronic Crime Research (eCrime)*, 2019.

[43] N. Voss. AFL-Unicorn. https://github.com/Battelle/afl-unicorn, 2021.

[44] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[45] M. Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2017.

[46] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *in Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[47] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the USENIX Security Symposium*, 2019.

A Fuzzing Strategy

```
struct ifreq {
#define IFHWADDRLEN     6
        union
        {
            char    ifrn_name[IFNAMSIZ];
        } ifr_ifrn;

        union {
           struct  sockaddr ifru_addr;
...
           struct  if_settings ifru_settings;
        } ifr_ifru;
};
```

Listing 3: 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 we 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`). We use the `void` pointer to pass the fuzzer mutated data to the kernel module. In the case of a networking module we use the pointer to pass a `struct ifreq` to the module (see Listing 3). 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, we 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 [13] software reverse engineering tool. While several previous works on kernel driver IOCTL fuzzing and IOCTL `cmd` excavation [8,24,32] 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, we 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 we use AFL's `afl-cmin` tool which keeps the minimum number of seeds that produce distinct instrumentation results during a fuzzing dry run.

B Version Magic

```
#define VERMAGIC_STRING \
  UTS_RELEASE " "\
  MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT\
  MODULE_VERMAGIC_MODULE_UNLOAD
       MODULE_VERMAGIC_MODVERSIONS     \
  MODULE_ARCH_VERMAGIC
```

Listing 4: Linux-2.6.30 vermagic macro defined in include/linux/vermagic.h

The *vermagic* macro shown in Listing 4, provides information about basic options that must match between a module and the kernel proper. These options are related to `Symmetric Multiprocessing`, `Preemption`, `Module Versioning` support, etc. The `UTS_RELEASE` macro specifies the full kernel version (base version + extension).

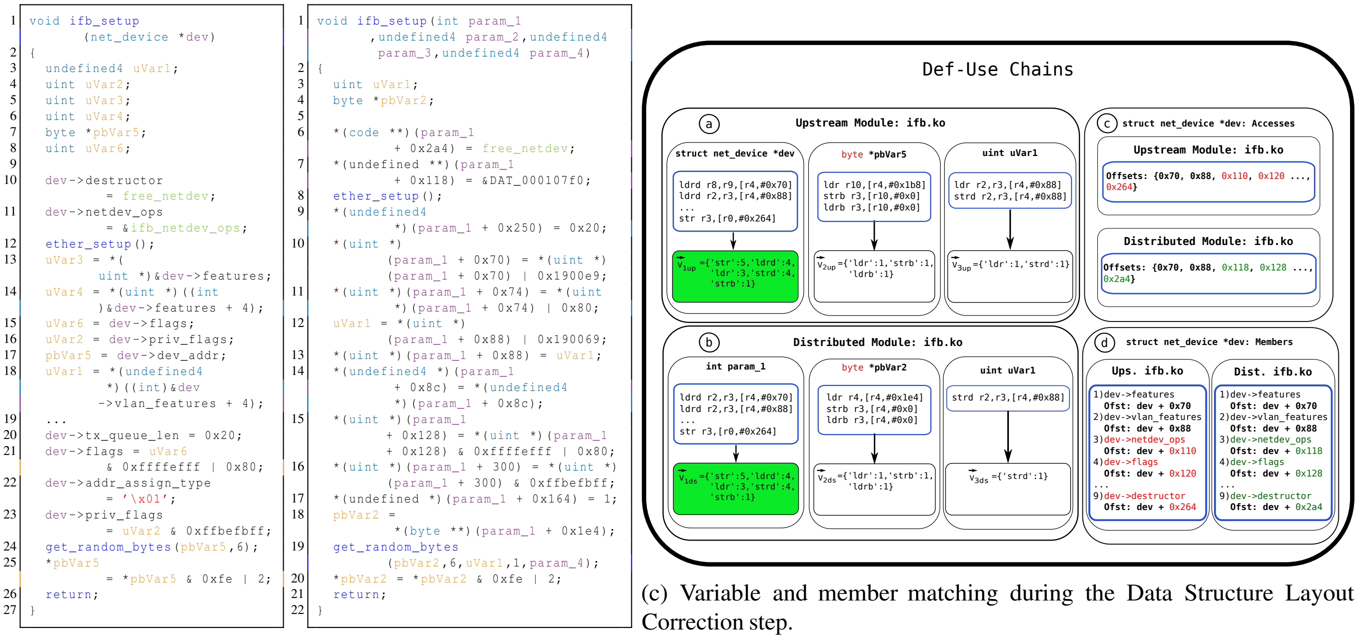
C BRIDGE_NETFILTER Definition

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

Listing 5: Linux-2.6.30 BRIDGE_NETFILTER option definition

D Data Structure Layout Correction

Figure 3 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 3a and 3b 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 3c 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 our 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 their instruction vectors (i.e., $\vec{v}_{1up} \approx \vec{v}_{1ds}$, $\vec{v}_{2up} \approx \vec{v}_{2ds}$



```
void ifb_setup
        (net_device *dev)
{
  undefined4 uVar1;
  uint uVar2;
  uint uVar3;
  uint uVar4;
  byte *pbVar5;
  uint uVar6;

  dev->destructor
          = free_netdev;
  dev->netdev_ops
          = &ifb_netdev_ops;
  ether_setup();
  uVar3 = *(
         uint *)&dev->features;
  uVar4 = *(uint *)((int
         )&dev->features + 4);
  uVar6 = dev->flags;
  uVar2 = dev->priv_flags;
  pbVar5 = dev->dev_addr;
  uVar1 = *(undefined4
          *)((int)&dev
         ->vlan_features + 4);
  ...
  dev->tx_queue_len = 0x20;
  dev->flags = uVar6
          & 0xfffffefff | 0x80;
  dev->addr_assign_type
          = '\x01';
  dev->priv_flags
          = uVar2 & 0xffbefbff;
  get_random_bytes(pbVar5,6);
  *pbVar5
          = *pbVar5 & 0xfe | 2;
  return;
}
```

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

```
void ifb_setup(int param_1
      ,undefined4 param_2,undefined4
       param_3,undefined4 param_4)
{
  uint uVar1;
  byte *pbVar2;

  *(code **)(param_1
          + 0x2a4) = free_netdev;
  *(undefined **)(param_1
          + 0x118) = &DAT_000107f0;
  ether_setup();
  *(undefined4
          *)(param_1 + 0x250) = 0x20;
  *(uint *)
         (param_1 + 0x70) = *(uint *)
         (param_1 + 0x70) | 0x1900e9;
  *(uint *)(param_1 + 0x74) = *(uint
          *)(param_1 + 0x74) | 0x80;
  uVar1 = *(uint *)
         (param_1 + 0x88) | 0x190069;
  *(uint *)(param_1 + 0x88) = uVar1;
  *(undefined4 *)(param_1
          + 0x8c) = *(undefined4
          *)(param_1 + 0x8c);
  *(uint *)(param_1
         + 0x128) = *(uint *)(param_1
         + 0x128) & 0xffffefff | 0x80;
  *(uint *)(param_1 + 300) = *(uint *)
         (param_1 + 300) & 0xffbefbff;
  *(undefined *)(param_1 + 0x164) = 1;
  pbVar2 =
          *(byte **)(param_1 + 0x1e4);
  get_random_bytes
         (pbVar2,6,uVar1,1,param_4);
  *pbVar2 = *pbVar2 & 0xfe | 2;
  return;
}
```

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

Def-Use Chains
a
Upstream Module: ifb.ko
struct net_device *dev
ldrd r8,r9,[r4,#0x70]
ldrd r2,r3,[r4,#0x88]
...
str r3,[r0,#0x264]
$\vec{v}_{1up}$ ={'str':5,'ldrd':4, 'ldr':3,'strd':4, 'strb':1}
byte *pbVar5
ldr r10,[r4,#0x1b8]
strb r3,[r10,#0x0]
ldrb r3,[r10,#0x0]
$\vec{v}_{2up}$ ={'ldr':1,'strb':1, 'ldrb':1}
uint uVar1
ldr r2,r3,[r4,#0x88]
strd r2,r3,[r4,#0x88]
$\vec{v}_{3up}$ ={'ldr':1,'strd':1}
b
Distributed Module: ifb.ko
int param_1
ldrd r2,r3,[r4,#0x70]
ldrd r2,r3,[r4,#0x88]
...
str r3,[r0,#0x264]
$\vec{v}_{1ds}$ ={'str':5,'ldrd':4, 'ldr':3,'strd':4, 'strb':1}
byte *pbVar2
ldr r4,[r4,#0x1e4]
strb r3,[r4,#0x0]
ldrb r3,[r4,#0x0]
$\vec{v}_{2ds}$ ={'ldr':1,'strb':1, 'ldrb':1}
uint uVar1
strd r2,r3,[r4,#0x88]
$\vec{v}_{3ds}$ ={'strd':1}
c
struct net_device *dev: Accesses
Upstream Module: ifb.ko
Offsets: {0x70, 0x88, 0x110, 0x120 ..., 0x264}
Distributed Module: ifb.ko
Offsets: {0x70, 0x88, 0x118, 0x128 ..., 0x2a4}
d
struct net_device *dev: Members
Ups. ifb.ko
1)dev->features Ofst: dev + 0x70
2)dev->vlan_features Ofst: dev + 0x88
3)dev->netdev_ops Ofst: dev + 0x110
4)dev->flags Ofst: dev + 0x120
...
9)dev->destructor Ofst: dev + 0x264
Dist. ifb.ko
1)dev->features Ofst: dev + 0x70
2)dev->vlan_features Ofst: dev + 0x88
3)dev->netdev_ops Ofst: dev + 0x118
4)dev->flags Ofst: dev + 0x128
...
9)dev->destructor Ofst: dev + 0x2a4

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

Figure 3: 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.

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 3c showcases which members of `struct net_device` are misaligned in our 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..

E Firmadyne Exploits

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 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 our dataset