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

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

Abstract—**Re-hosting Internet of Things (IoT) firmware can oftentimes be a tedious process, especially when analysts have to intervene with the analysis to ensure further progress. When it comes to Linux-based firmware, one crucial problem that current re-hosting systems face, is that the configuration of the custom kernels used by these systems, significantly deviates from the configuration of the IoT kernel modules used in firmware images. As a consequence, kernel artifacts, such as the memory layout of data structures might differ between the custom kernels and the IoT kernel modules. To analyze the IoT kernel modules within these kernels, the analyst often has to invest significant amount of engineering effort and time to align the offending data structures within the custom kernels. In this paper, we present `FirmDiff`, an automated binary diffing framework that enables analysts to effectively detect and align the misaligned data structures between the custom kernels produced by the FirmSolo re-hosting framework and the Linux kernel modules in IoT firmware. The goal of `FirmDiff` is to improve the configuration of FirmSolo's kernels to closely approximate the configuration of the IoT kernels in the firmware images, such that the IoT kernel modules can be analyzed without errors. We evaluate `FirmDiff` on a dataset of 10 firmware images with 148 IoT kernel modules that crash during re-hosting with FirmSolo. Using `FirmDiff`'s findings, we identify 37 misaligned data structures in FirmSolo's kernels for these images. After aligning the layout of 35 of these data structures, FirmSolo's refined kernels successfully load 28 previously crashing kernel modules.**

I Introduction

Analyzing the firmware code that runs on Internet of Things (IoT) devices has always been a formidable challenge for analysts. Generally, the IoT domain is characterized by the insufficient information disclosed by vendors about the functionality of firmware and the lack of source code. Due to this fact, every firmware analysis solution might suffer from limitations that can hinder the analysis. Thus, analysts often have to manually intervene with the analysis process and invest an extensive amount of engineering effort and time to address the issues and ensure further progress. Unfortunately, there are few automated solutions that can lift some of the manual investigation burden and improve the efficiency and effectiveness of firmware analysis.

Firmware analysis is a well-studied research area. On the one hand, static analysis techniques [8], [13], [24] offer scalable solutions, however at the expense of soundness. On the other hand, dynamic analysis approaches [2], [5], [16], [21], [26], [27], [31] provide good tradeoffs between scalability and robustness, but frequently require human mediation to aid the analysis. When it comes to complex code though, such as Linux-based (kernel level) firmware code, dynamic analysis techniques (e.g., re-hosting) are more popular since they produce fewer false positives than static analysis.

Linux-based firmware resembles general purpose operating systems (i.e., consists of a kernel, the drivers, and the user space applications) albeit storage-optimized and restricted in terms of capabilities, to be able to run on low-power, low-memory IoT devices. However, the simplicity of IoT Linux kernels is entirely outweighed by the huge variety of IoT hardware peripherals that these kernels target. Unfortunately, state-of-the-art emulators support only but a fraction of that hardware and are thus incapable of emulating these IoT kernels. To account for this issue, Linux-based re-hosting systems substitute the IoT kernels within IoT firmware (images) with custom configured and built kernels that can be emulated by current emulators. Next, these custom kernels are used to analyze either the user level code or drivers in the target firmware. Compared to the other re-hosting approaches, FirmSolo [2] is the only scalable system that automatically configures and builds custom emulation-ready kernels, whose configuration approximates the corresponding IoT kernels. FirmSolo uses these kernels to load and dynamically analyze the IoT kernel modules within firmware images.

However, FirmSolo's kernel configuration process is inherently dependent on the metadata (i.e., kernel symbols) embedded within the IoT privileged code (kernel and kernel modules) and the kernel source code used to compile the custom kernels (FirmSolo uses only open source kernel versions). By design, FirmSolo primarily ensures that kernel symbols (functions and data structures) are available to the IoT kernel modules in order for the latter to load into the custom kernel. FirmSolo focuses on lower level implementation semantics, such as ensuring that the IoT kernel modules access the members of the data structures in the custom kernels at the correct offset, only when issues arise during re-hosting. In particular, inconsistencies in the memory layout of these data structures can lead to misaligned data structure memory accesses during the analysis, which in turn can lead to these IoT kernel modules crashing. Generally, the memory layout of kernel data structures is affected by two causes: the configuration options used in the kernel build process and any modifications directly applied by the vendors in their kernels' source code. While FirmSolo attempts to fix the layout of data structures when IoT kernel

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23012
www.ndss-symposium.org

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

This paper introduces `FirmDiff`, a binary diffing framework that compares the memory layout of data structures common to both the IoT kernel modules (and IoT kernel) and FirmSolo's custom kernels. `FirmDiff`'s goal is to help the analyst identify differences in the memory layout of data structures common between the IoT kernel modules and FirmSolo's custom kernels, and also provide sufficient information to the analyst to address these differences. Consequently, by aligning the layout of these data structures, the analyst can build custom kernels that closely resemble their IoT counterparts. The premise behind building these refined kernels is to load and analyze the IoT kernel modules without misaligned data structure accesses that occur during emulation, thus improving the overall firmware re-hosting. Specifically, the outcome of `FirmDiff`'s analysis are the names of data structures and their members (i.e., their offsets and names) that are misaligned between the IoT kernel modules and FirmSolo's custom kernel. Analysts can use this information to enhance FirmSolo's kernel configuration and build process to produce refined custom kernels that can load and analyze the IoT kernel modules without crashes thwarting the analysis. In particular, `FirmDiff` detects function matches between the diffed binaries and for each function pair it further matches the non-primitive type variables (i.e., of a kernel data structure type) used within these functions. The differences in the memory accesses of these variable pairs (i.e., the corresponding data structure members) indicate discrepancies between the configuration of FirmSolo's custom kernel and the IoT kernel or data structure modifications in the IoT kernel. In turn, the analyst can address these differences and modifications in FirmSolo's kernels, to prevent (kernel module) crashes occurring during the IoT kernel module analysis.

As its foundation, `FirmDiff` uses Ghidriff [6] a binary diffing tool that conducts fast and effective comparisons of similar binaries at function granularity. We evaluate `FirmDiff` on a dataset of 7 firmware images, originating from FirmSolo [2]. Furthermore, we also analyze 3 additional modern images from the Greenhouse dataset (published in 2023) to assess whether `FirmDiff`'s analysis capabilities work for modern firmware. Overall, these firmware images contain 148 kernel modules that crash during the analysis with FirmSolo. We showcase how the information produced by `FirmDiff` aids analysts to supplement the configuration process of FirmSolo to produce custom kernels that better approximate the IoT kernels used by the firmware images in our dataset. `FirmDiff` identifies 37 misaligned data structures in FirmSolo's kernels for these images. After aligning the layout of 35 of these data structures, FirmSolo's refined kernels successfully load 28 previously crashing kernel modules.

In summary we make the following contributions.

- We propose a novel data structure memory layout comparison technique that identifies inconsistencies between the configuration of FirmSolo's custom kernels and the IoT kernels of firmware images. By addressing these inconsistencies, analysts can produce refined kernels that can load and effectively analyze IoT kernel modules without crashes occurring during the analysis.
- We present `FirmDiff`, the prototype implementation of this technique. `FirmDiff` constitutes a binary diffing framework that compares the memory layout of data structures that are common between FirmSolo's custom kernels and the IoT kernel modules.
- To showcase `FirmDiff`'s effectiveness in improving firmware analysis we compare the refined kernel configurations and kernels produced with the help of `FirmDiff` against the ones produced by FirmSolo. We show that the custom kernels produced with the help of `FirmDiff`, achieve better firmware re-hosting compared to the kernels produced by FirmSolo.

To further foster research in the area, we will make `FirmDiff`'s source code available to the public [1].

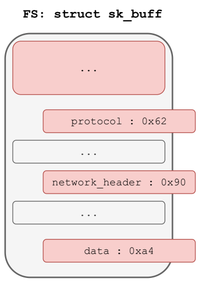
II Background

In this Section, we provide background information about Ghidra, Ghidriff, and FirmSolo, which are the foundations of `FirmDiff`.

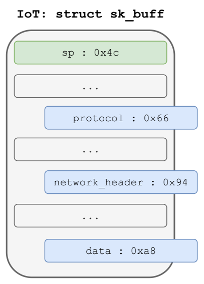
II-A Ghidra

Ghidra [10] is a state-of-the-art binary reverse engineering tool. Similar to the other reverse engineering systems[1], [15], [22], Ghidra offers various types of analyses, such as disassembling and decompiling binary code, extracting metadata information (e.g, the endianness, the architecture and debugging information – DWARF), binary rewriting, etc. In addition, alongside the main analysis tool, Ghidra also offers a built-in binary diffing utility, called `Ghidra Version Tracking` (GVT). GVT's primary functionality is to compare two (similar) binaries and by using different correlators (e.g., *Exact Function Bytes Match*, *Exact Function Instructions Match*, *Similar Symbol Name Match*, etc), match the functions between the compared binaries. As the names suggest these example correlators match functions (between the compared binaries) if the bytes comprising their body are identical, match functions if they consist of the same instructions, and if the function names match, respectively. We note here that the use of these correlators enables GVT to match functions that are not 100% identical, and might contain changes (e.g., patches) between the two binaries.

[1]https://github.com/BUseclab/FirmDiff



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



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

```
struct sk_buff {
        struct sk_buff          *next;
...
#ifdef CONFIG_XFRM
        struct  sec_path        *sp;
#endif
        unsigned int            len, data_len;
...
        __be16                  protocol;
...
}
```

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

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

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

II-B Ghidriff

Ghidriff [6] is a command line binary diffing tool designed for fast and effective comparison of similar binaries (i.e., multiple versions of the same binary). Essentially, Ghidriff is a Python wrapper on top of Ghidra's headless mode that also leverages GVT for binary diffing. Originally, GVT was tied to Ghidra's GUI and was unavailable in Ghidra's headless mode. Ghidriff is the first system that disassociates GVT's functionality from Ghidra's GUI to enable binary diffing in the headless mode.

Note that Ghidriff goes beyond merely running GVT in headless mode and also generates additional information. For example, Ghidriff also provides the similarity ratio of matched functions (i.e., decompiled code, instruction, basic block similarity ratio, etc), code diffs of modified functions, the names and code of added and removed functions between the compared binaries. `FirmDiff` relies on Ghidriff's capability to infer matching functions, even in stripped binaries (i.e., their function names are stripped), to further match the variables used in these functions. We discuss how `FirmDiff` implements the variable matching process in Section IV.

II-C FirmSolo

FirmSolo is a full-system re-hosting framework targeting Linux-based firmware code. Specifically, FirmSolo is designed to re-host privileged level firmware code, in the form of kernel modules. Similar to other re-hosting frameworks such as Firmadyne [5], FirmAE [16], Honware [27] and EASIER [21], FirmSolo replaces the IoT kernel within the firmware image with a custom kernel supported by emulators, such as QEMU [3]. The kernel replacement is crucial since the IoT kernels are built specifically to run on physical IoT hardware. Since, state-of-the-art emulators do not support the numerous different hardware that billions of IoT devices use [11], these emulators cannot emulate their IoT kernels.

FirmSolo automatically reverse engineers the IoT kernel of the target firmware image and in turn uses the information extracted to configure and build a custom kernel that closely approximates the image's IoT kernel (but is supported by QEMU). Specifically, FirmSolo extracts the kernel symbols used by the IoT kernel modules (and the IoT kernel if it is available) of the target image and maps these symbols to the configuration options that guard their implementation in the kernel source code. These configuration options comprise the configuration of FirmSolo's custom kernel. Once it is built, the custom kernel can in turn be used by dynamic analysis systems to load and analyze the kernel modules within the target firmware image. Unfortunately, the IoT kernel modules can crash during their emulation. Certain crashes occur when the IoT kernel modules incorrectly access data structures shared with the custom kernel. FirmSolo implements a data structure memory alignment algorithm that aligns the layout of common data structures between the custom kernel and IoT kernel modules. However, this algorithm operates only on IoT kernel modules that conform to these requirements:

1) The crashing IoT kernel modules are open-source and their equivalent upstream kernel modules must be compiled by FirmSolo.
2) The crashing IoT kernel modules are not stripped (i.e., the function names used by the kernel modules are available).
3) There is an available crash dump (i.e., a kernel `Oops` message).
4) The crash occurs while the IoT kernel modules are being loaded into the custom kernel (i.e., while executing their `init_module` function).
5) The crash occurs within a known function that belongs to the crashing IoT kernel modules or their dependencies and not within a function of the custom kernel.
6) The misaligned data structures are not modified at source by the vendors (FirmSolo uses the unmodified open source equivalents of the IoT kernels).

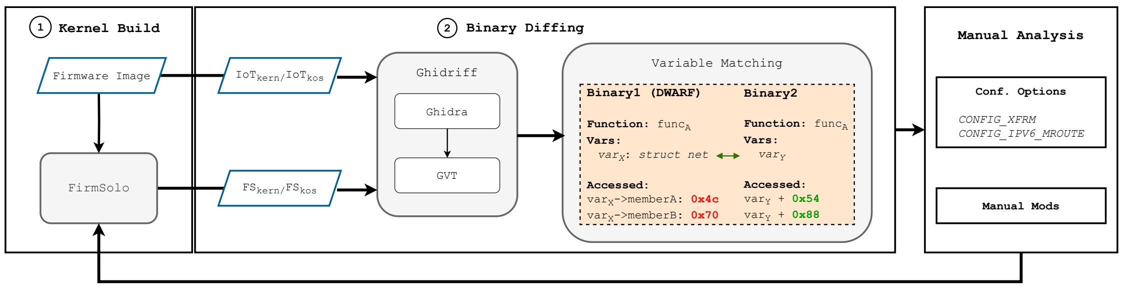


Fig. 2: The overview of `FirmDiff`. The figure depicts the 2 stages of `FirmDiff`. In the Kernel Build stage, `FirmDiff` consumes a firmware image and analyzes it with FirmSolo. In the Binary Diffing stage, `FirmDiff` diffs the FS_{kos} and IoT_{kos} open-source modules with Ghidriff and matches the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

These limitations prevent FirmSolo from aligning all the data structures whose layout in the custom kernel differs from the layout that the IoT kernel modules expect. As a result FirmSolo cannot address the kernel module crashes in these scenarios. `FirmDiff` only requires that there exist open source IoT kernel modules so that it can analyze these modules along with their counterparts compiled by FirmSolo (see Section IV-B).

III Motivation

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

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

IV Overview

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

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

IV-A Kernel Build

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

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

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

IV-B Binary Diffing

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

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

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

For each compared pair of kernel module binaries, `FirmDiff` follows these steps: 1) It analyzes both binaries with Ghidra. 2) It runs GVT on the two binaries and matches their functions. 3) Finally, for each function match `FirmDiff` invokes the variable matching algorithm. This algorithm correlates the variables with a data structure type (e.g., `struct net`) between the matched functions. It leverages the DWARF information embedded within the FS_{kos} and FS_{kern} during their compilation, to detect the data structure types of the variables. `FirmDiff` disregards variables with a primitive type (e.g., int) since their memory layout is static and not affected by any configuration options in the kernel. Once the variables with a data structure type are paired (between the diffed binaries), `FirmDiff` proceeds to extract the memory accesses for each variable in a pair. These memory accesses correspond to data structure members accessed (i.e., their offsets from the base of the data structure). `FirmDiff` displays the member accesses (member names and their offsets) for each variable pair side by side, enabling a quick and easy comparison. If the offsets do not match, the memory layout of the target data structure (i.e., the type of the variable pair under test) is misaligned between the modules in FS_{kos} and IoT_{kos}. After the variable matching algorithm concludes, `FirmDiff` stores all the information produced (i.e., the variable pairs and their member accesses for each function) within the `JSON` and markdown files produced by Ghidriff.

IV-C Manual Analysis

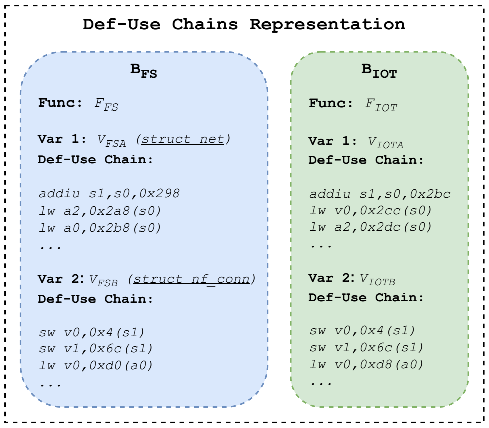
After the information produced by `FirmDiff` is available, the analyst can parse that information to extract the misaligned data structures highlighted by `FirmDiff`. Afterwards, by using other tools available (e.g., [4], [10], [14], [18]) the analyst can pinpoint the root cause behind the data structure misalignments. Specifically, for each variable pair (with a data structure type) the analyst can infer if and which members are shifted and by how many bytes. Based on the number of bytes shifted, they can in turn detect which configuration options need to be toggled in FirmSolo's kernel configuration process or manually modify the data structures' source code to align these members between the compared binaries. To infer which options need to be toggled, the analyst can consult FirmSolo's built-in utility which provides a mapping for each data structure (in the FS_{kern} kernel) and the options that affect its layout. The outcome of this analysis is a refined FS_{kern} that agrees with the IoT_{kos} about the layout of the data structures fixed by the analyst. The IoT_{kos} should not crash when correctly accessing the members of the aligned data structures while being analyzed within the refined FS_{kern}.

V Implementation

In this Section we provide the implementation details behind `FirmDiff`.

V-A Kernel Build

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



```
addiu s1,s0,0x298
lw a2,0x2a8(s0)
lw a0,0x2b8(s0)
...

sw v0,0x4(s1)
sw v1,0x6c(s1)
lw v0,0xd0(a0)
...
```

```
addiu s1,s0,0x2bc
lw v0,0x2cc(s0)
lw a2,0x2dc(s0)
...

sw v0,0x4(s1)
sw v1,0x6c(s1)
lw v0,0xd8(a0)
...
```

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

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

V-B Binary Diffing

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

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

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

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

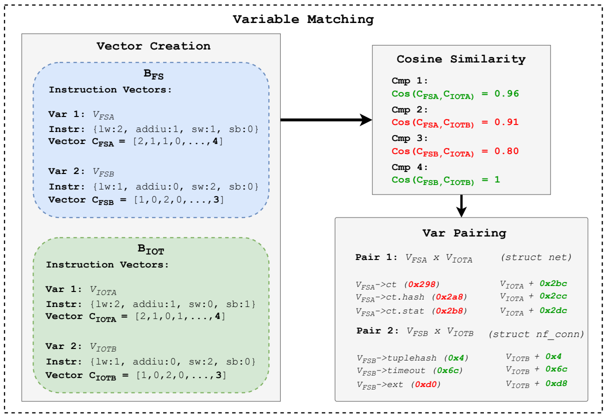

Variable Matching
Vector Creation
B_{FS}
Instruction Vectors:
Var 1: V_{FSA}
Instr: {lw:2, addiu:1, sw:1, sb:0}
Vector C_{FSA} = [2,1,1,0,...,4]
Var 2: V_{FSB}
Instr: {lw:1, addiu:0, sw:2, sb:0}
Vector C_{FSB} = [1,0,2,0,...,3]
B_{IOT}
Instruction Vectors:
Var 1: V_{IOTA}
Instr: {lw:2, addiu:1, sw:0, sb:1}
Vector C_{IOTA} = [2,1,0,1,...,4]
Var 2: V_{IOTB}
Instr: {lw:1, addiu:0, sw:2, sb:0}
Vector C_{IOTB} = [1,0,2,0,...,3]
Cosine Similarity
Cmp 1:
Cos(C_{FSA},C_{IOTA}) = 0.96
Cmp 2:
Cos(C_{FSA},C_{IOTB}) = 0.91
Cmp 3:
Cos(C_{FSB},C_{IOTA}) = 0.80
Cmp 4:
Cos(C_{FSB},C_{IOTB}) = 1
Var Pairing
Pair 1: V_{FSA} x V_{IOTA} (struct net)
V_{FSA}->ct (0x298) V_{IOTA} + 0x2bc
V_{FSA}->ct.hash (0x2a8) V_{IOTA} + 0x2cc
V_{FSA}->ct.stat (0x2b8) V_{IOTA} + 0x2dc
Pair 2: V_{FSB} x V_{IOTB} (struct nf_conn)
V_{FSB}->tuplehash (0x4) V_{IOTB} + 0x4
V_{FSB}->timeout (0x6c) V_{IOTB} + 0x6c
V_{FSB}->ext (0xd0) V_{IOTB} + 0xd8

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

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

Variable Pairing. The variable pairing step borrows its logic from the layout alignment algorithm that FirmSolo uses. We present the inner workings of variable pairing in Figure 4. First, FirmDiff maps the Def-Use chains for the variables V_{FSA}, V_{FSB}, V_{IoTA} and V_{IoTB} into the vectors C_{FSA}, C_{FSB}, C_{IoTA} and C_{IoTB}, respectively. Specifically, each ISA mnemonic (e.g., sw) is mapped to a coordinate in the vector. The value of each coordinate is equal to the number of times the corresponding mnemonic occurs in the Def-Use chain of the target variable. In addition, we set the last element of each vector as the total number of instructions accessing the corresponding variable to make the vector similarity measurement discussed below more accurate. Specifically, to match the variables V_{FSA} and V_{FSB} with the variables V_{IoTA} and V_{IoTB}, FirmDiff uses the cosine similarity measurement [28] between C_{FSA}, C_{FSB} and C_{IoTA}, C_{IoTB}. The variables with the highest similarity score are considered a pair. In our example, variable V_{FSA} is paired with V_{IoTA} and variable V_{FSB} is paired with V_{IoTB}. Furthermore there can only be one match between the variables in F_{FS} and F_{IOT}, to not confuse the analyst during the manual analysis process.

Once the variable matches are available, `FirmDiff` transfers the types from the variables in F_{FS} to their counterpart in F_{IOT}. If the variable has a primitive type (e.g., `int`), `FirmDiff` discards the pair since its layout is always static and cannot be modified manually or by toggling configuration options (i.e., it cannot be incorrectly accessed). Thus, `FirmDiff` keeps only the variable pairs with a data structure type, since their layout can be modified and potentially be misaligned between B_{FS} and the B_{IoT}. For these variable pairs, `FirmDiff` outputs their type and their accessed members (i.e., names and offsets) in ascending order by offset. The information about which members are accessed and their offsets is embedded within the Def-Use chains and DWARF information extracted previously. We note here that `FirmDiff` only knows the names of the data structure members accessed for the variables in F_{FS}, due to the availability of the DWARF information in B_{FS}. For variables that belong in F_{IOT} (since B_{IOT} is usually stripped), `FirmDiff` displays only the offsets of the accessed members. Once the variable matching algorithm concludes, `FirmDiff` stores the variable pairs and their accesses for each matched function within a `JSON` and a markdown file. Next, these findings can be parsed and/or processed programatically by the analyst.

V-C Manual Analysis

To identify any potential misaligned data structures between the FS_{kern} and the modules in IoT_{kos} (and the IoT_{kern}), the analyst can process the data structure type variable pairings and their memory accesses for the matched functions in B_{FS} and B_{IOT}. Specifically, misaligned data structures are recognizable based on the differences (or shifts) in the offsets of their accessed members within two matched functions. As `FirmDiff` does not provide further information about the root cause behind these shifts, the analyst leverage other complementary static and dynamic analysis tools (e.g., cscope [18], pahole [4], Ghidra [10] and GDB [14]) to find the reason behind the misalignments. As mentioned in Section IV-B the layout of data structures is affected by the configuration options in the kernel configuration and any modifications applied by vendors in their kernels' source code. FirmSolo provides a builtin utility that maps each data structure in the FS_{kern} to the configuration options that affect the layout of the data structure. The analyst can first consult this utility to explore possible alignment solutions requiring only toggling configuration options. When no effective solution is achieved, it is an indication that the IoT kernel source code has been modified by the vendors (see Section VI-C). Thus, the analyst can either modify FS_{kern}'s source code (i.e., pad the layout of data structures) and/or its configuration to correctly align the layout of the misaligned data structures and re-invoke stage ① to build the refined FS_{kern} kernel. If the refined FS_{kern} kernel and IoT_{kos} agree about the layout of their common data structures then the IoT_{kos} should not crash when accessing the correctly aligned members of these data structures (while being loaded or executed within the FS_{kern}).

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

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

VI Evaluation

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

RQ1 How efficient is `FirmDiff` in highlighting misaligned data structures in the FS_{kern} and FS_{kos} (§ VI-B)?
RQ2 How effective is `FirmDiff` in improving the kernel configuration of the FS_{kern} (§ VI-C)?
RQ3 Can `FirmDiff` improve FirmSolo's firmware re-hosting (§ VI-D)?

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

VI-A Dataset

We evaluate `FirmDiff` on a set of 10 firmware images with 7 of the images originating from the FirmSolo dataset. Specifically, we randomly select the set from the images that have persistent kernel module crashes in FirmSolo, even after applying its memory layout alignment algorithm. We aim to illustrate that `FirmDiff` is capable of aiding the analyst in correctly aligning the memory layout of the misaligned data structures in these cases. We also evaluate `FirmDiff` on 3 randomly picked firmware images from the Greenhouse [26] dataset which was published in 2023. We choose a set of images that FirmSolo can successfully emulate and load their kernel modules, but is unable to address their kernel module crashes. The reason for this additional set of images is to showcase that `FirmDiff`'s analysis is also applicable to modern firmware images. The images in our dataset belong to 6 vendors and their kernels range from version `2.6.21.5` to `4.4.198`. We provide the relevant information in Table I.

VI-B Misaligned Data Structures

In this Section we evaluate the accuracy of `FirmDiff`'s variable matching algorithm and also compare its efficiency with

FirmSolo's misaligned data structure discovery capabilities, to answer RQ1.

Overall, `FirmDiff` analyzes 842 open source kernel modules (i.e., open source kernel modules in IoT_{kos} and their counterparts in FS_{kos}). During this analysis, `FirmDiff` matches 29,485 functions and pairs 28,269 variables (with a data structure type). Out of the total functions matched, 20,564 (70%) functions actually contain zero variable pairs with a data structure type. `FirmDiff`'s analysis takes 17 minutes on average. Due to the volume of the results, we manually verify `FirmDiff`'s findings for a subset of 18 crashing open source kernel modules and 1 kernel binary and confirm the existence of 37 misaligned data structures. We detail how we proceed to correct the layout of these data structures in Section VI-C.

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

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

Image 3 is the only case where we proceed to analyze the FS_{kern} and IoT_{kern} kernels with `FirmDiff`. In particular, the image contains two crashing kernel modules, one open source (`nf_conntrack_prot_gre.ko`) and one closed source (`jcp.ko` [2]). The `jcp.ko` kernel module implements a proprietary protocol that exposes `USB` devices over the network. Since the `nf_conntrack_prot_gre.ko` exists in both the FS_{kos} and IoT_{kos}, both versions are analyzed by `FirmDiff`. However, `jcp.ko` does not have an upstream counterpart compiled by FirmSolo. Thus, we cannot analyze `jcp.ko` with `FirmDiff` to identify potential misaligned data structures between this kernel module and the FS_{kern} kernel. Unfortunately, `FirmDiff`'s analysis on `nf_conntrack_prot_gre.ko` is ineffective and does not provide information about the misaligned data accessed by this kernel module. For these reasons, we also proceed to analyze the IoT_{kern} and FS_{kern} for image 3 since the kernels have information about all the data structures that are shared between the kernel modules (open and closed source) and the core kernel. We aim to detect the misaligned data structures accessed by both crashing kernel modules in this case. We note though, that the entire analysis takes over 5 hours, hence it is not scalable for a large number of firmware images. In addition, compared to the hundreds of functions matched within the kernel modules, in this case there are 14,066 functions matched between the IoT_{kern} and FS_{kern} kernels. Analyzing all the function matches for potential misaligned data structures requires extensive effort and time. For this reason, we inspect only the first 200 functions matched between IoT_{kern} and FS_{kern} for image 3 and identify 4 misaligned data structures.

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

VI-C Kernel Configuration

Identifying misaligned data structures is only the first step. To improve firmware re-hosting and analysis, the analyst needs to address these misalignments. In this Section we detail how `FirmDiff`'s analysis aids the analyst to improve the configuration of FirmSolo's kernels, to answer RQ2.

For this experiment, we further analyze the 37 misaligned data structures highlighted by `FirmDiff` (see Section VI-B), using complementary source code and binary introspection tools [4], [10], [14], [18]. We successfully align 35 out of these 37 data structures in the corresponding FS_{kern} kernels, by either toggling configuration options in the kernel, directly modifying the source code of the data structures or both. We note that we are able to identify the combination of configuration options and modifications that align the layout of these data structures through trial and error. Of course human expertise and knowledge also plays a crucial role in this case. Specifically, we are able to correct the layout of 20 data structures by simply toggling configuration options in the kernels, 10 data structures by directly modifying their implementation in the kernel source code, and 5 data structures by both toggling options and modifying their source code. In contrast, FirmSolo's memory alignment algorithm is only able to correct the layout of 4 misaligned data structures (see Table II in Appendix A). Unfortunately, we are unable to align the layout of 2 data structures (`struct net` and `net_device`) for a single image (i.e, image 10). In particular, we cannot guess the correct combination of modifications and configuration options that align the layout of these data structures. As we discuss below, extensive modifications by the vendors in their kernel source code can impede the analyst's layout alignment process.

Vendor Modifications. The majority of the misaligned data structures we identify in the three modern images (12 of 15 data structures), are modified in their source code by the vendors. Listing 7 provides the code diff of the implementation of `struct net_device` between the upstream version of

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

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

Fig. 5: Implementation of `struct net_device` in the open source Linux-4.4.60.

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

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

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

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

VI-D Firmware Re-hosting

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

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

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

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

VII Discussion

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

`FirmDiff` like other static analysis systems has limitations. First, `FirmDiff`'s analysis depends tightly on the accuracy of Ghidra. If Ghidra incorrectly analyzes the target binaries (e.g., incorrectly disassembles the code of functions), `FirmDiff`'s analysis will in turn be ineffective. Second, unlike FirmSolo, `FirmDiff` cannot pinpoint the origin of the crashes within the kernel modules. `FirmDiff`, only provides information about the layout of data structures used by the crashing kernel modules. The analyst is responsible for identifying if the actual root cause behind the crash is a misaligned data structure access or another type of bug.

VIII Related Work

Linux-based Firmware Re-hosting. Costin et al. [9] is one of the first works that pioneered firmware re-hosting. They employed QEMU's [3] userland emulation and `chroot` to execute and analyze the services (e.g., webservers) within the file-systems of firmware images. Similar, Greenhouse [26] focuses on isolating and re-hosting the individual services within firmware images using QEMU's userland emulation. Firmadyne [5] and FirmAE [16] are full-system re-hosting frameworks which re-host and analyze the user level code within firmware images. They achieve full-system re-hosting by replacing the binary kernels within firmware images with custom pre-built and emulation-ready kernels. These kernels can execute the user level code of firmware images and expose it to various bug and vulnerability analyses. Similar, Honware [27] uses re-hosting to create fake (honeypot) firmware devices, deployed on the Internet, to study real world attacks.

EASIER [21] and FirmSolo [2] also belong in the full-system re-hosting frameworks family. Contrary to the aforementioned firmware re-hosting systems, EASIER and FirmSolo target the firmware kernel modules in IoT firmware. Both systems replace the firmware kernels with custom built kernels that are supported by QEMU. The kernels are tailored primarily to load the firmware kernel modules and expose them to dynamic analyses (e.g., fuzzing). All the aforementioned re-hosting systems (except FirmSolo and EASIER) use pre-built kernels that target thousands of images, but cannot load any IoT_{kos}. Even though EASIER can load IoT kernel modules, it can only do so for ARM firmware and a small number of modules. In contrast, FirmSolo's kernels can analyze the kernel modules of thousands of (both MIPS and ARM) firmware images. However, as discussed in Section III FirmSolo's custom kernels are not always configured properly, thus resulting in the IoT kernel modules crashing when loaded or executed in these kernels. `FirmDiff`'s analysis aids the analysts to manually configure and build custom kernels that better approximate their IoT counterparts to address these crashes.

Hardware-in-the-loop. Systems in this category follow a hybrid approach which relies on both re-hosting and the physical IoT device to analyze IoT firmware code. AVATAR [29], SURROGATES [17] and Inception [7] are examples of systems that forward I/O operations to the physical device through its debugging interface (e.g., JTAG) while firmware code is execute and analyzed within the emulator (e.g., QEMU). While these systems can effectively re-host and analyze IoT firmware code, they are not scalable due to the requirement that the physical IoT device is available.

Data Structure Layout Recovery. Recovering the layout of data structures from (kernel) binaries is a well covered research topic. The majority of works [20], [23], [25] recover the data structure memory accesses via symbolic execution. Other approaches [12], [19], [30] extract the layout of data structures within binaries through various control and data flow techniques. Unfortunately, both the absence of semantic information in the IoT binaries and/or the requirement to emulate the IoT kernels (see Section II-C) pose great limitations in discovering the layout of the data structures used by the IoT kernel modules. In contrast, `FirmDiff` provides a fast and effective static analysis approach, which overcomes the aforementioned limitations. First, `FirmDiff` does not require to emulate the IoT firmware kernels and second by using the debugging information embedded in FirmSolo's kernel modules, `FirmDiff` is able to recover the type and (partially) the layout of data structures used by the IoT kernel modules.

IX Conclusion

In this paper, we present `FirmDiff`, a binary diffing framework that compares the memory layouts of data structures common to FirmSolo's custom kernels and IoT kernel modules. `FirmDiff`'s goal is help the analyst detect potential differences in the memory layouts of these data structures and also address these differences. In turn, the analyst can build robust kernels that can load and analyze the IoT kernel modules without crashes occurring during emulation.

X Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback and comments. This research was enabled by National Science Foundation under grant CNS-1942793.

Appendix

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

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

The table illustrates the unique misaligned data structures we identify with the help of `FirmDiff`, as well as the misaligned data structures detected by FirmSolo during the analysis of the kernel modules in our dataset. With the help of `FirmDiff` we are able to detect more unique misaligned data structures (16) compared to FirmSolo (4). In addition, by using the information provided by `FirmDiff` we are able to align the layout of 35 misaligned data structures, while FirmSolo correctly aligns only 4 misaligned data structures. Thus, `FirmDiff` outperforms FirmSolo both in the detection and layout alignment discovery.

References

[1] V. 35, “Binary Ninja,” https://binary.ninja/, 2024.

[2] I. Angelakopoulos, G. Stringhini, and M. Egele, “FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules,” in *Proceedings of the USENIX Security Symposium*, 2023.

[3] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, 2005.

[4] A. Carvalho, “Pahole,” https://lwn.net/Articles/335942/, 2009.

[5] D. D. Chen, M. Egele, M. Woo, and D. Brumley, “Towards Automated Dynamic Analysis for Linux-based Embedded Firmware,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] Clearbluejar, “Ghidriff,” https://github.com/clearbluejar/ghidriff, 2023.

[7] N. Corteggiani, G. Camurati, and A. Francillon, “Inception: System-Wide Security Testing of Real-World Embedded Systems Software,” in *Proceedings of the USENIX Security Symposium*, 2018.

[8] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A Large-Scale Analysis of the Security of Embedded Firmwares,” in *Proceedings of the USENIX Security Symposium*, 2014.

[9] A. Costin, A. Zarras, and A. 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.

[10] N. R. Directorate, “Ghidra,” https://ghidra-sre.org/, 2024.

[11] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, “SoK: Enabling Security Analyses of Embedded Systems via Rehosting,” in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2021.

[12] Q. Feng, A. Prakash, M. Wang, C. Carmony, and H. 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.

[13] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, “Static Analysis for Discovering IoT Vulnerabilities,” 2021.

[14] GNU, “GDB The GNU Project Debugger,” https://www.sourceware.org/gdb/, 2024.

[15] Hex-Rays, “IDAPro,” https://hex-rays.com/ida-pro/, 2024.

[16] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, “FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[17] K. Koscher, T. Kohno, and D. Molnar, “SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems,” in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[18] B. Labs, “Cscope,” http://cscope.sourceforge.net/, 2012.

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

[20] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the CERIAS Annual Information Security Symposium*, 2010.

[21] 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.

[22] Radare, “Radare2,” https://rada.re/n/, 2024.

[23] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos, “Dsibin: Identifying dynamic data structures in c/c++ binaries,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

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

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

[26] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith, A. Doupé, T. Bao, Y. Shoshitaishvili, and R. Wang, “Greenhouse: Single-Service rehosting of Linux-Based firmware binaries in User-Space emulation,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[27] 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.

[28] Wikipedia, “Cosine Similarity,” https://en.wikipedia.org/wiki/Cosine_similarity, 2024.

[29] J. Zaddach, L. Bruno, A. Francillon, and D. 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.

[30] Z. Zhang, Y. Ye, W. You, G. Tao, W.-C. Lee, Y. Kwon, Y. Aafer, and X. 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)*, 2021.

[31] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proceedings of the USENIX Security Symposium*, 2019.