Pandawan: Quantifying Progress in Linux-based Firmware Rehosting

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

Abstract

The Internet of Things (IoT) is frequently the epicenter of cyberattacks due to its weak security. Prior works introduce various techniques for analyzing the firmware of IoT devices for bugs and vulnerabilities, especially through firmware re-hosting. However, comparing the emulation outcomes of different re-hosting approaches can be very challenging. In this paper, we present `Firmware Initialization Completion Detection` (`FICD`), a technique that enables the comparison of full-system re-hosting approaches across their re-hosting capabilities. In addition, prior works lack an important capability; they do not focus on both the user and privileged aspect of IoT firmware as a unit. Since prior work is not capable of *holistically* analyzing (both the user and privileged level) IoT firmware, we develop `Pandawan`, a framework that enables the holistic re-hosting and analysis of IoT firmware at scale. We use `FICD` to illustrate `Pandawan`'s re-hosting improvements over the state-of-the-art, such as Firmadyne, FirmAE, and FirmSolo on a dataset of 1,520 firmware images. Our experiments show that `Pandawan` outperforms these systems, by executing up to 6% more user level programs and 21% more user code basic blocks, on average, than these systems. Furthermore, `Pandawan` loads 9% more IoT kernel modules and executes 26% more kernel module basic blocks on average than FirmSolo. We also use `Pandawan` to holistically analyze the firmware images by inspecting the interactions (through system calls) of user level code with kernel module code. `Pandawan` transforms the system call information into seeds for the TriforceAFL kernel fuzzer to analyze the kernel modules within the firmware images. The TriforceAFL experiment on 479 firmware images with seeds, discovered 16 bugs on 12 binary kernel modules, 6 of which are previously unknown bugs. The bugs affect 8 closed and 4 open source kernel modules.

1 Introduction

The Internet of Things (IoT) has gained immense popularity in the past decade. Billions of embedded devices and gadgets, such as WiFi routers, IP cameras, and smart wearables fill up tech store shelves and find their way into customers' homes and businesses [26]. Despite its growth, the IoT is infamous for its weak security. This is evident from impactful cybersecurity attacks assisted by the IoT such as the Mirai [2], Trickbot [33], and Meris [47] botnet attacks, which rattled the Internet infrastructure's foundations. Attackers exploited the poor security posture of the IoT vendors' devices, primarily the use of weak passwords, hardcoded backdoors, and legacy (outdated) software to compromise millions of devices and gadgets [36]. Even though these security incidents raised awareness about the security of the IoT, device vendors still prioritize profit over protecting their products against malicious actors [24]. Thus, it is imperative to improve the security of these devices and gadgets to protect them against cyber threats in the future. Fortunately, the research community has already directed its focus on exposing bugs and vulnerabilities in the firmware that runs on these IoT devices.

A subset of prior works in this research area conduct their analysis directly on firmware running on physical IoT devices. They either rely on a mobile or Web application [8, 19, 49] to communicate with the IoT firmware or execute the firmware within an emulated environment while forwarding memory accesses to the actual device [12, 30, 50]. Other works that eschew physical devices can be categorized into static and dynamic analysis approaches. Both static analysis [13, 20] and dynamic analysis such as firmware re-hosting [7, 28, 48] are popular approaches that have lately dominated the IoT firmware analysis landscape.

When considering Linux-based firmware, full-system re-hosting (or emulation) techniques are the most popular. Static analysis techniques generally suffer from a multitude of false positives or false negatives [20]. Even though re-hosting techniques revolutionized IoT firmware analysis, by extending bug and vulnerability testing towards increasingly sophisticated firmware, the field is characterized by the absence of two important aspects.

First, the community lacks a mechanism to objectively compare the capabilities of different re-hosting systems beyond crude metrics (e.g., number of bugs found within 24

hours). However, to assess the re-hosting progress we need to answer the following question:

> ***Q:*** *How can we quantify the forward progress of IoT firmware analysis approaches in full-system re-hosting?*

Prior full-system Linux-based re-hosting works showcase their progress and improvements over their predecessors by relying on ad-hoc and coarse-grained metrics (e.g., number of bugs or vulnerabilities found or networking connectivity achieved [28, 48]). However, each re-hosting system adopts a different design directly affecting the firmware execution flow and the emulation performance (e.g., emulating both user and kernel level code *vs*. emulating user level code only), thus introducing disparity in the firmware emulation outcome. The existing metrics are too generic to capture these discrepancies and quantify the contribution of each system in the IoT re-hosting landscape. Thus, to answer *Q* we first require metrics (e.g., programs executed, code coverage, kernel modules loaded, etc.) that better reflect the systems' re-hosting capabilities. Importantly, such finer-grained metrics imply a new challenge: At which point during the emulation should we measure them to ensure an objective comparison between re-hosting approaches? Clearly, relying on elapsed wall-clock emulation time as the indicator for taking a measurement is not sufficient, as emulation speed is directly impacted by a re-hosting system's design. Fundamentally, to enable objective comparisons, we require a conceptual reference point (with respect to system progress rather than wall-clock time) that can be identified in any re-hosting system. Once this reference point is reached, we can take the measurement (i.e., collect the metrics) and perform the comparison. We discuss below how it is possible to identify such a reference point to mark the completion of the firmware initialization logic.

The second missing capability of full-system Linux-based re-hosting is that current state-of-the-art systems target only one aspect of IoT firmware; either its user level [7,28,46,48,55] or its privileged (kernel) level aspect [1, 38, 55]. In real world devices, though, both the user and privileged aspect of the IoT firmware run cooperatively. To effectively analyze IoT firmware code it is therefore important to follow a more *holistic* approach that includes both aspects of the IoT firmware in the re-hosting and analysis process.

On the surface it seems that FirmSolo [1] is readily capable of holistic re-hosting since it builds kernels that can re-host a variety of binary IoT kernel modules and also support IoT user level code re-hosting due to the stable Linux system call ABI. However by design, FirmSolo's kernels are specifically tailored to kernel *module* re-hosting. As a result, these kernels might lack functionality (in the form of kernel symbols – functions and data structures) required by user level firmware code to execute successfully. Figure 2 in Section 3.2 describes such an example as the capability to access a `Memory Technology Device` (MTD) peripheral through a character device, via the `mtd_[open,read,write]` functions. Since these functions are not required nor accessed by kernel modules, the aforementioned functionality will be omitted from FirmSolo's kernels. Then, if a user level program requires these functions to execute successfully, it will fail. To mitigate these scenarios it is crucial to identify and supplement FirmSolo's kernels with kernel functionality not used by the firmware's kernel modules, but required by user level firmware code to execute.

To address the aforementioned shortcomings, in this paper, we present `Firmware Initialization Completion Detection` (`FICD`), a technique that detects the reference point at which the firmware's initialization phase is complete during emulation. `FICD` makes it possible to quantify the forward progress in firmware re-hosting, by enabling the meaningful comparison of different full-system re-hosting approaches on metrics such as the number of user programs executed, user and kernel code coverage, or number of kernel modules loaded, up to that point. The reference point represents a concept common to all full-system re-hosting approaches despite their differences; the end of the firmware initialization phase. Since the firmware operation is usually predetermined (e.g., network routing), IoT devices are primarily configured during startup by the firmware's configuration and bootup scripts. Every full-system re-hosting approach will inevitably execute these scripts and thus undergo the firmware initialization phase. Thus, the reference point detected by `FICD` can be used to meaningfully compare these systems along a metric that can showcase their contribution to re-hosting (e.g., code coverage).

In addition, to fill the void in holistic full-system firmware re-hosting we introduce `Pandawan`, a re-hosting framework that builds atop of (Py)PANDA [14, 17] and FirmSolo's firmware re-hosting capabilities to provide Operating System (OS) level introspection and dynamic analysis of both user and privileged (i.e., kernel module) firmware code. To achieve holistic firmware re-hosting and analysis, `Pandawan` features the `Kernel Augmentation` (`KA`) technique. `KA` first isolates the kernel functionality typically included by vendors in their IoT firmware kernels (in the form of kernel symbols). Then, it builds `Pandawan`'s "augmented" kernels which contain this functionality to enable the successful re-hosting of firmware user level code and kernel modules alike.

To showcase the improvements of `Pandawan` over the state-of-the-art, we rely on `FICD` to compare `Pandawan`, Firmadyne, FirmAE and FirmSolo. For our experiments we use a dataset consisting of 1,520 firmware images. Our investigation shows that `Pandawan` outperforms Firmadyne, FirmAE, and FirmSolo, by executing up to 6% more user level programs and 21% more user code basic blocks, on average, than these systems. In addition, `Pandawan` loads 9% more IoT kernel modules and executes 26% more kernel module basic blocks on average than FirmSolo.

To demonstrate `Pandawan`'s utility in firmware analysis, we use it to fuzz the kernel modules within the firmware images. Fuzzing the entire system call interface would not be efficient, since the majority of system calls (depending on the

arguments they are given) would target the core kernel code instead of kernel module code. Thus, Pandawan targets only the subset of system calls that result in the execution of code in IoT kernel modules. To this end, Pandawan's holistic analysis approach fundamentally enables the tracing of system calls invoked by user level programs which lead to kernel module code execution. In turn, the traces expose entry points from user level to kernel (module) level code. Pandawan uses the entry point information to generate system call chains in a format (seeds) compatible with the TriforceAFL [34] kernel fuzzer to fuzz the kernel modules within the firmware images. Pandawan collects traces for 479 firmware images. After transforming these traces into seeds and providing them to TriforceAFL, the fuzzer triggers 16 bugs on 8 closed and 4 open source kernel modules. Twelve of the bugs found are previously known and six are previously unknown bugs.

In summary we make the following contributions:

- We introduce Firmware Initialization Completion Detection, a technique that enables the meaningful comparison of full-system re-hosting approaches despite their discrepancies and quantify their progress in the firmware re-hosting domain.
- We propose the Kernel Augmentation technique which configures and builds kernels that contain functionality usually included by IoT vendors in their privileged firmware code, so that these kernels are conducive to holistic firmware re-hosting and analysis.
- We present Pandawan, the prototype implementation of Kernel Augmentation. Pandawan constitutes an OS level introspection and dynamic analysis framework that enables holistic (user and privileged level) re-hosting and analysis of IoT firmware.
- To quantify Pandawan's improvements over the state-of-the-art we use FICD to compare it with Firmadyne, FirmAE and FirmSolo re-hosting frameworks. Furthermore, we use Pandawan's holistic analysis capabilities to analyze the firmware images in our dataset. The information collected is then used by the TriforceAFL kernel fuzzer to analyze the kernel modules within the firmware images.

To further foster the research in this area, we will make Pandawan's source code publicly available[1].

2 Background

Before discussing the design of Pandawan, we first provide background information about full-system firmware re-hosting and the systems we will reference throughout the paper; PyPANDA, Firmadyne, FirmAE and FirmSolo.

2.1 Full-system Firmware Re-hosting

Full-system firmware re-hosting (or emulation) [18] is a technique used to run a binary firmware image without the need for the IoT hardware it was designed for. Particularly, the firmware (image) code is executed within an emulated environment (e.g., QEMU [4]), where hardware accesses are handled by the emulator. Full-system firmware re-hosting has become popular mainly due to two major benefits; 1) allowing the emulation of IoT firmware at scale, and 2) exposing firmware code to dynamic analysis (e.g., fuzzing) and operating system level introspection. However, full-system firmware re-hosting can be challenging. A recurring issue in the IoT infrastructure is the *lack of standarization* [15]. Particularly, the absence of common specifications and protocols leads to the production of billions of devices with diverse and proprietary hardware (e.g., NVRAM and Wireless NICs). Hence, IoT firmware and specifically its kernel component is dependent on each particular IoT device's System on Chip (SoC) and hardware peripherals.

Unfortunately, state-of-the-art emulators such as QEMU only support a fraction of the hardware peripherals used by IoT and embedded devices [18] and thus are unable to emulate the IoT firmware kernels. As a result, current full-system re-hosting approaches either target only user level IoT firmware code [7, 28, 48] or the IoT kernel modules [1, 38], by substituting the original firmware kernel with a custom kernel compatible with the emulators.

2.2 (Py)PANDA

One of the core foundations of Pandawan is PyPANDA [14], a Python frontend to PANDA [17], an Operating System (OS) level introspection framework built on top of QEMU. Besides emulating IoT related architectures (e.g., ARM and MIPS), (Py)PANDA also enables users to control the guest code execution and inspect the guest's internal state (e.g., memory). The OS level guest introspection is possible through a set of PANDA's existing C/C++ plugins and through PyPANDA's Python plugin interface. These plugins unlock unique capabilities not available in the upstream version of QEMU, such as system call and basic block tracing, and guest OS function hooking. Pandawan builds upon PyPANDA's Python interface to introduce new custom plugins and enable holistic analysis and introspection of IoT firmware.

2.3 Firmadyne & FirmAE

Firmadyne [7] and FirmAE [28] are among the state-of-the-art when it comes to full-system firmware re-hosting and dynamic analysis approaches. Specifically, they are designed to emulate the user level code of IoT firmware images and subject it to various bug and vulnerability analyses, such as testing against exploits from the Metasploit framework [39] or fuzzing. By design, both systems replace the original firmware kernels, which are incompatible with QEMU, with custom pre-built kernels that can be booted under QEMU. Since both the original IoT kernels and the pre-built kernels are based on Linux, they share the same stable system call interface required by user level firmware code to run.

Unfortunately, by relying on custom pre-built kernels,

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

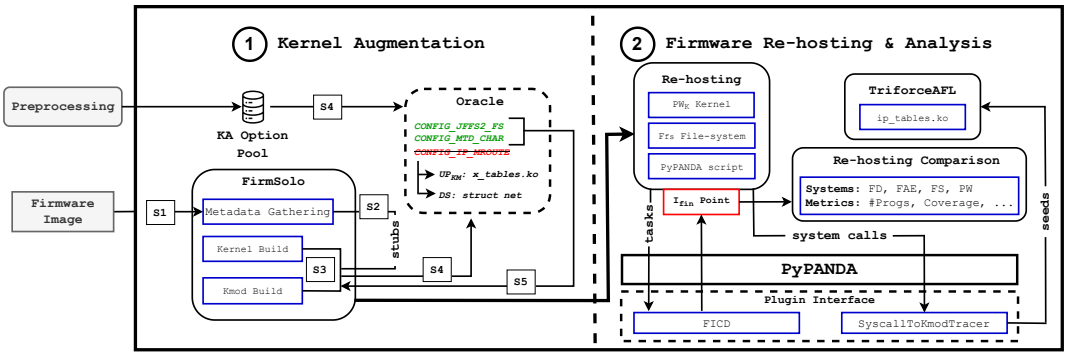


Figure 1: System overview of Pandawan: The figure features the two components of Pandawan. The first component, Kernel Augmentation is a 5-step process that produces Pandawan's PW_k kernels. The second component, Firmware Re-hosting & Analysis uses a PyPANDA script to emulate the PW_k kernel and the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

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

2.4 FirmSolo

FirmSolo [1] is the first framework that facilitates dynamic analysis of Linux-based IoT kernel modules at scale. Unlike Firmadyne and FirmAE, FirmSolo targets the privileged level aspect of IoT firmware. Specifically, it automatically configures and builds kernels that can load the kernel modules contained in the file-systems of IoT firmware images. Existing dynamic analysis systems (e.g., kernel fuzzers) can leverage FirmSolo as the foundation to analyze the IoT firmware kernel modules for bugs and vulnerabilities. While FirmSolo is also capable of re-hosting user level firmware code through its compatibility with Firmadyne, it was not designed for that purpose. In particular, the kernels produced by FirmSolo are configured exclusively based on the metadata extracted from the IoT kernel modules and the original firmware kernels if they are available. Figure 2a shows why this configuration process is not always effective.

3 Overview

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

In Section 4.1 we discuss in detail how to apply FICD on different full-system re-hosting approaches and compare them based on their emulation capabilities.

To fill the void in holistic firmware re-hosting we present Pandawan, a framework capable of holistic (both user and privileged) re-hosting and analysis of IoT firmware code. As illustrated in Figure 1, Pandawan constitutes a two-component framework: ① The Kernel Augmentation (KA) component supplements FirmSolo's kernels (FS_k) with kernel functionality frequently included by vendors in their IoT firmware kernels (IoT_k) and used by user level firmware code. We refer to this functionality as user-code-required functionality. The outcome of component ① are Pandawan's "augmented" kernels (PW_k), which support both user level and kernel module code (i.e., holistic) re-hosting. ② The Firmware Re-hosting & Analysis component is responsible for the holistic re-hosting of the firmware code while emulating the PW_k kernels under PyPANDA. This component also employs PyPANDA's plugins to conduct a series of OS introspection and holistic analyses (e.g., tracing system calls that lead to kernel module code execution) during the firmware emulation run. Then, the results of the analyses can be transformed into seeds to fuzz the firmware kernel modules with a kernel fuzzer (e.g., TriforceAFL).

Throughout the rest of the paper we use the notations *user* code to refer to user level firmware code and *kernel* code to refer to privileged/kernel level code. The privileged level component can be further classified into the core kernel and kernel module code. We also reference the terms *program*, *process*, and *task*. Intuitively, a program corresponds to an executable in the firmware's file-system. We define a process as the running instantiation of a program during emulation. Finally, we refer to a task as the concatenized argv vector, representing (as a string) a program (name) along with its arguments.

3.1 Firmware Initialization Completion Detection

As mentioned previously, quantifying the progress of full-system re-hosting approaches is challenging, due to their unique implementation features that impact the emulation of a firmware image (F_{image}) for each re-hosting framework. For

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

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

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

(b) Ghidra snippet of `ncc`'s `loadCfg` function

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

(c) Ghidra snippet of `ncc`'s `flash2rootfs` function

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

example, the combination of both *user* and *kernel* code being re-hosted may result in different emulation speeds and thus in a different amount of overall code executed at the same point in time compared to only *user* code emulation. Thus, simply using ad hoc comparative methods such as the wall-clock time as the common denominator to assess the progress of re-hosting systems does not produce objective results. Instead, `FICD` aims to detect the point (I_{fin}) during the emulation of F_{image} where its initialization phase has finished. At I_{fin}, full-system re-hosting approaches can be compared using different metrics, such as the number of executed user level programs, *user* and *kernel* code coverage, or number of kernel modules loaded, which we call `emulation-based` metrics. In Section 5.5 we use these `emulation-based` metrics to compare the emulation progress of `Pandawan`, Firmadyne, FirmAE and FirmSolo. However, these metrics can of course be supplemented with additional ones depending on the analysis' requirements (see Section 6). We note that I_{fin} is not fixed (in time) for any two re-hosting systems, especially if their emulation speed varies.

The rationale behind `FICD` is that all the firmware activity up to the I_{fin} point is automated via the configuration and startup scripts within F_{image}'s file-system. Beyond I_{fin}, there is none to little automated activity occurring, since IoT firmware primarily invokes new functionality (i.e., tasks), such as setting new firewall rules or recording a video, based on its interactions with external actors (i.e., human interaction [27]). We consider detecting events that lead to new activity after I_{fin} as a problem orthogonal to this research and we leave it as future work.

Since the activity after the firmware initialization phase is limited, I_{fin} can be considered as any point during the emulation of F_{image} after it has stopped executing new tasks. Specifically, `FICD` defines a grace period (or *time frame*) tf during which the emulation of F_{image} can stay alive while F_{image} does not execute any new tasks (to see how we choose tf refer to Section 5.5). If during the emulation of F_{image}, time greater than tf elapses without executing a new task, `FICD` marks that point as I_{fin}. Here, metrics can be collected and compared. Even though I_{fin} might not be fixed (timewise) between different full-system re-hosting approaches that emulate F_{image}, it will always represent F_{image}'s end of initialization phase (regardless of the re-hosting framework).

3.2 Holistic Re-hosting & Analysis

As discussed previously, there is currently no prior work that supports holistic firmware re-hosting and analysis. At first glance, FirmSolo (see Section 2.4) seems to be readily capable of supporting holistic firmware re-hosting and analysis by emulating both firmware *user* and *kernel* (i.e., kernel module) code. On the contrary though, our investigations reveal that FirmSolo's FS_k kernels lack important `user-code-required` functionality required by *user* code to execute.

Specifically, the absence of the `user-code-required` functionality in the FS_k kernels might result in firmware *user* code to either prematurely terminate its execution or exhibit unwanted behavior, such as getting stuck in an infinite loop. We provide a motivating example of such a case for `D-Link DIR-826L` Wi-Fi router firmware in Figure 2a. The example illustrates the serial console output of F_{image}'s emulation run, where the capability to access a `Memory Technology Device` (MTD) through a character device is missing from the FS_k kernel. As a result, the `ncc` program which handles the configuration of the IoT device and uses character devices

to interact with the MTD peripheral, gets stuck in an infinite loop when executed in FirmSolo (lines 7-9 in Figure 2a). Specifically, `ncc` executes the `loadCfg` function which in turn calls the `flash2rootfs` function. We provide the Ghidra decompilation snippets for these functions in Figures 2b and 2c. The `loadCfg` function calls `flash2rootfs` within an infinite loop (lines 6-21 in Figure 2b) and exits the loop only when `flash2rootfs` returns a non zero value (line 16 in Figure 2b). The return value of `flash2rootfs` depends on the successful opening of the `/dev/mtdblock4` character device (lines 5-10, 11-17 and 19 in Figure 2c). The ability to access a MTD peripheral through a character device is included in the kernel via the `CONFIG_MTD_CHAR` configuration option. As none of the firmware kernel modules use any functionality guarded by `CONFIG_MTD_CHAR`, FirmSolo did not enable this configuration option in its FS_k kernel. In Section 3.2.2, we explain how `Kernel Augmentation` adds `user-code-required` functionality (e.g., `CONFIG_MTD_CHAR`) required by user level firmware programs to execute successfully, into `Pandawan`'s kernels. Next, we detail `Pandawan`'s components.

3.2.1 Preprocessing

Based on our observations (see Section 5.3) vendors use similar configuration parameters (options) to configure their IoT firmware kernels (IoT_k). In particular, it is common for privileged firmware code of different IoT devices to share the same capabilities, such as supporting MTD peripherals or networking subsystems (i.e., `netfilter`), which are accessed by *user* code. Thus, by knowing the configuration of the IoT_k kernels of firmware images we also acquire the `user-code-required` functionality required by *user* code to execute. This `user-code-required` functionality is crucial for firmware images without a `KALLSYMS` entry. The `KALLSYMS` entry is a table containing information about the kernel symbols exported by the kernel (and thus about its configuration). The entry is optionally embedded in the kernel binary if the kernel is compiled with the `CONFIG_KALLSYMS` configuration option. We consider the configuration of the IoT_k kernels of firmware images without a `KALLSYMS` entry partially known since the only source about these images' kernel configuration originates from their kernel modules. Thus, to account for the missing kernel configuration (and in turn `user-code-required` functionality) from these firmware images, `Pandawan` collects popular configuration options from the IoT_k kernels of firmware images with a `KALLSYMS` entry. Then, `Pandawan` enables these options in its PW_k kernels for the firmware images without a `KALLSYMS` entry, to provide the `user-code-required` functionality needed by *user* code to execute.

3.2.2 Kernel Augmentation

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

To produce the PW_k kernel for F_{image}, `Pandawan` follows these five steps: S1: Extract the file-system of F_{image} and gather metadata information, such as the `KALLSYMS` entry from the IoT_k kernel (if both the kernel and entry exist in F_{image}). S2: Furthermore, to aid the loading process of additional kernel modules, supplement PW_k kernels (their code) with "stubs" of symbols that are not present in the upstream kernel source code, but required by F_{image}'s kernel modules. S3: Build the upstream counterparts (UP_{kos}) of open-source kernel modules within F_{image}'s file-system. The UP_{kos} modules are compiled with debugging information available (i.e., `DWARF`). We note here that S3 and S4 will only executed for firmware images that do not have an available `KALLSYMS` entry. We consider the kernel configuration of firmware images with a `KALLSYMS` entry to be "fully" known. Particularly, the entry reveals information about the specific symbols needed by both the *user* and *kernel* code in these images. In these cases, the respective configuration options for these symbols (also used in the IoT_k kernels) will be added (by default) in S5 in the PW_k kernels. S4: Invoke the *Oracle* process with the UP_{kos} modules and either the MIPS or ARM configuration option pool (depending on F_{image}'s architecture). The *Oracle* is a filtering process which checks and removes configuration options from the MIPS or ARM pools that affect the layout of kernel data structures used by the UP_{kos} modules (produced in S2). S5: Finally, feed the "safe" configuration options that remain from S4 to its kernel configuration and build process to produce the PW_k kernels.

3.2.3 Firmware Re-hosting & Analysis

In component ②, `Pandawan` relies on the custom file-system creation and network configuration logic of FirmAE [28] to produce a dedicated PyPANDA script for F_{image} (see Section 4.2.3). We note here that the custom file-system also supports `Pandawan`'s (and FirmSolo's) kernel module loading logic. Next, `Pandawan` uses the modified file-system, the PW_k kernel and the PyPANDA script to emulate F_{image} under PyPANDA and conduct a series of analyses on the target firmware using PyPANDA's plugins. Examples of the analyses implemented are the collection of coverage about the *user* code executed during the firmware emulation, such as the executed program names and their executed QEMU Translation Block (TB) addresses, and the tracing of system calls (i.e., ID and arguments) invoked by *user* code during the emulation.

Furthermore, through its holistic re-hosting, `Pandawan` captures the interactions between the *user* code and *kernel* code. This introspection provides information about the inner workings of both the user and privileged code of F_{image}, which can then be used by dynamic analysis systems to thoroughly analyze F_{image}. Since prior works only focus on one aspect of the firmware; either the user or the privileged level, they lack

insights about how both these firmware aspects actually work together. Thus, these approaches have to resort to heuristics or impose limitations during their firmware analysis. For example, FirmSolo does not have the ability to extract information about system calls that interact with the kernel modules. It is limited only to fuzzing these modules through their `IOCTL` interface.

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

4 Implementation

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

4.1 Firmware Initialization Completion Detection

`FICD` seeks to identify the reference point I_{fin} in the emulation of a firmware image where the firmware completed its initialization phase. To this end, `FICD` considers that a firmware image reached I_{fin} if no previously unseen (i.e., unique) tasks are launched within tf seconds. We refer to tf as the *time frame* parameter. To assess whether a task is unique, `FICD` uses the *Levenshtein*[2] edit distance (`ed`) to check the similarity of a task executed at a given point in time with all the tasks that have been executed prior to that point. We define a task (t) as previously unseen if it has a similarity below a certain threshold h (see Section 5.5) with all the previously executed tasks. That is, t is previously unseen iff $\forall_{t_i} ed(t,t_i) < h$ where t_i refers to all tasks started before t. The use of `ed` rather than strict equality is crucial to mitigate cases where nearly identical tasks (e.g., `/bin/iptables -t` **`filter`** `-F` and `/bin/iptables -t` **`nat`** `-F`) impact the discovery of the I_{fin} point. Our implementation of `FICD` uses a thread that runs alongside the main emulation and evaluates the above expression every five seconds. Once tf expired without a previously unseen task being launched, `FICD` declares I_{fin} reached and signals that measurements can be taken.

4.2 Holistic Re-hosting & Analysis

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

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

4.2.1 Preprocessing

Before proceeding to the execution of `Pandawan`'s components we first execute the *Preprocessing* step. In this step we pre-compute the `user-code-required` functionality that needs to be included in the PW_k kernels in component ① to support the successful re-hosting of *user* code. Specifically, we extract the `KALLSYMS` kernel symbols (where available) from all the available IoT_k kernels in our dataset (950 kernels). Then, we group the symbols into two sets S_{MIPS} and S_{ARM} matching the IoT_k kernel architecture respectively. Next, we use FirmSolo's `symbol-to-configuration-option` mapping mechanism to map each symbol s in S_{MIPS} and S_{ARM} to the corresponding configuration option that guards s's implementation in the kernel source code. This yields two pools (sets) of configuration options, one for MIPS (O_{MIPS}) and one for ARM (O_{ARM}). These pools represent the `user-code-required` functionality that is added to the PW_k kernels in component ①.

4.2.2 Kernel Augmentation

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

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

Step S2 To improve its kernel module re-hosting capabilities, `Pandawan` implements the technique used by EASIER [38] for loading out-of-tree kernel modules. Specifically, `Pandawan` supplements the PW_k kernels with "stubs" of kernel symbols (not present in the upstream kernel source code) required by F_{image}'s kernel modules. `Pandawan` detects these symbols in step S1. The symbol stubs simply return a `NULL` value and make the corresponding symbol available to the kernel modules through the kernel's `EXPORT_SYMBOL` macro. Without these symbol stubs the kernel modules would immediately fail to load during the kernel module loading process. Next, if F_{image} contains a `KALLSYMS` entry, `Pandawan` immediately skips to step S5.

Step S3 `Pandawan` executes its kernel build process to build and acquire the upstream versions (UP_{kos}) of open-source kernel modules within F_{image} (since `Pandawan`, like FirmSolo, uses the upstream version of the IoT_k kernel). The UP_{kos} kernel modules are compiled with debugging information enabled (DWARF), required in step S4.

Step S4 In this step, `Pandawan` invokes the *Oracle* process whose goal is to provide the necessary `user-code-required` functionality to each PW_k kernel. First though, *Oracle* makes

sure to filter out any option $o \in O_{MIPS}$ (or O_{ARM}) that affects the layout of kernel data structures used by the kernel modules within F_{image}. If the PW_k kernel and F_{image}'s kernel modules do not agree about the layout of their common data structures, then misaligned data structure member accesses and in turn kernel module crashes might occur during emulation.

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

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

4.2.3 Firmware Re-hosting & Analysis

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

File-system and PyPANDA script creation. At first, `Pandawan` uses the custom file-system and network configuration process of FirmAE to create the F_{fs} file-system and set the network connectivity of F_{image}. Specifically, the custom file-system creation process proceeds as follows. Similar to FirmAE and FirmSolo, `Pandawan` uses `binwalk`[3] to extract F_{image}'s file-system, but ensures that it supports `Pandawan`'s kernel module loading logic. The F_{fs} file-system also contains a custom serial console binary, which `Pandawan` uses to issue commands to the emulated firmware (e.g., run TriforceAFL's fuzzing harness). Since the original startup scripts within the F_{fs} file-system are responsible for loading any of the kernel modules, `Pandawan` has to ensure that these scripts do not load the kernel modules that crashed during the hybrid kernel build process in S5. `Pandawan` applies any kernel module substitutions in-place, meaning that if a kernel module c that crashed during S5 has a substitution s (i.e., a counterpart in UP_{kos}), then c is replaced by s within F_{fs}, else c is deleted from F_{fs} altogether. Thus, when a startup script attempts to load c during emulation, it will either load s instead or no module at all.

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

`Pandawan` also addresses kernel module crashes that occur during its re-hosting experiments (see Section 5.3). In these cases, `Pandawan` first uses FirmSolo's data structure layout recovery mechanism to address the errors. If the crashes persist then `Pandawan`, during the creation of F_{fs}, substitutes the crashing modules with their counterparts in UP_{kos} (if they exist) else it removes the crashing modules from F_{fs} entirely.

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

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

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

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

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

Seed Creation. However, to create useful seeds for the TriforceAFL fuzzer, Pandawan also requires information about the arguments (i.e., values) of the system calls that lead to the execution of kernel module code. Another PyPANDA plugin, the syscalls_logger, is responsible for collecting the arguments of all the system calls executed during the emulation. Pandawan combines the data collected by the syscalls_logger and SyscallToKmodTracer plugins to create seeds and fuzz the firmware kernel modules. Seed generation for fuzzers is not a new research topic. Prior works [3, 9, 37, 42, 53] leverage either system call traces, concolic execution or static analysis to extract information about the program execution and use it to create seeds for popular fuzzers such as syzkaller [23] or AFL/AFL++ [21, 51]. Our approach is similar to [37] in the sense that it relies on system call traces to create seeds for the TriforceAFL kernel fuzzer. However, we target only kernel modules that are accessed by system calls either through a socket or a file descriptor since TriforceAFL's fuzzing agent [35] is inherently compatible with these types of system calls.

As we show in Section 5, the majority of TriforceAFL compatible system calls (*sc*) that lead to kernel module code execution are networking (e.g., sys_setsockopt) or file-based system calls (e.g., sys_write). Thus, to create valid seeds for the fuzzer, Pandawan first has to detect which sys_socket and sys_open system call creates the file descriptor accessed by each *sc*, respectively. To gain this information, Pandawan leverages the first argument of a *sc* which corresponds to the file descriptor number fd. Then for each *sc*, Pandawan parses the information of syscalls_logger and detects the sys_socket or sys_open system call that was invoked by the same process as *sc* and returned a value equal to fd. System calls that operate on the same fd are grouped together in the same seed.

Fuzzing. To start fuzzing the kernel modules in F_{image}, Pandawan runs the PW_k kernel along with the F_{fs} file-system and the seeds found previously under TriforceAFL. The I_{fin} point for F_{image} plays a critical role also in this scenario. In particular, Pandawan spawns a thread that runs alongside the TriforceAFL process and waits until F_{image} reaches its I_{fin} point, since all the target kernel modules would be loaded by that point. Then, the thread connects through a UNIX socket to the custom serial console within the F_{fs} file-system and initiates the fuzzing harness to begin the kernel module fuzzing. We note here that Pandawan limits the range of system calls that the fuzzing harness can execute to the twelve system calls illustrated in Table 6 in Appendix C. The rationale behind this limitation is to only execute system calls compatible with TriforceAFL's fuzzing agent.

5 Evaluation

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

RQ1 Is Pandawan capable of holistically re-hosting firmware code (§ 5.3)?

RQ2 How effective is Pandawan on enabling holistic firmware analysis (§ 5.4)?

RQ3 How efficient is FICD when it comes to quantifying the forward progress in full-system IoT firmware re-hosting (§ 5.5)?

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

5.1 Dataset

For our evaluation, we use the dataset of FirmSolo and a subset of firmware images used in Greenhouse [46]. The Greenhouse dataset was shared with us upon request to the authors. We use the FirmSolo dataset since it is supported by FirmSolo, Firmadyne, and FirmAE. We do not use the entire Greenhouse dataset (consisting of 7,347 firmware images) since the majority of the images use the same kernel version as the images in the FirmSolo dataset. Instead, we pick at random 50 images from the 97 images that use a kernel version between 4.4.198 (latest version used by FirmSolo) and 4.9.206 (the latest kernel

version used by the ARM/MIPS 32bit images in Greenhouse). Our dataset consists of 1,520 firmware images containing a total of 61,319 binary kernel modules. The firmware images in the dataset target the MIPS (984 images) and the ARM (536 images) platforms. Finally, the firmware images span a total of 95 unique Linux kernel versions, ranging from version `2.6.18` to version `4.9.206`.

5.2 Experimental Setup

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

5.3 Holistic Re-hosting

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

Preprocessing. We execute this step only once in our experiments, to generate the configuration option pools O_{MIPS} and O_{ARM}, which constitute the `user-code-required` functionality that is added in the PW_k kernels in component ①. We include these configuration option pools as part of `Pandawan`'s repository. During this step, we extract 754 options in the O_{MIPS} and 726 options in the O_{ARM} pools, respectively. We opt to only keep the most popular options in both sets. In Figures 3a and 3b in Appendix A, we provide the Cumulative Distribution Function (CDF) of the configuration options in O_{MIPS} and O_{ARM} pools, respectively, over the images that have `KALLSYMS` available. To find the most popular options in each pool we use the Kneedle algorithm [40] to deduce the "knee" points in both figures (marked with the red lines). We filter out the options that reside beneath these knee points. Specifically, the knee point in Figure 3a is 30 (images), 418 (options), while the knee point in Figure 3b is 85 (images), 513 (options). In the end, 336 (754 - 418) and 213 (726 - 513) options remain in the O_{MIPS} and O_{ARM} pools, respectively.

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

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

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

Serial Console Connectivity. As discussed in Section 4.2.3, to run the fuzzing harness within a target firmware image, `Pandawan` has to first connect through a `UNIX` socket to a custom serial console within the image's file-system. To discover the number of firmware images that are accessible through their serial console, we conduct an experiment where we emulate each firmware image with `Pandawan` and attempt to connect to its serial console through a `UNIX` socket after 30 seconds of emulation time and run a dummy command (e.g., `/bin/ls`). A successful connection with a serial console means that we get the output of the dummy command in the serial log output of the firmware image. We are able connect to the serial console of 1,200 (86%) out of the 1,389 firmware images that `Pandawan` successfully re-hosts. We "break down" the reasons behind the console connectivity issues for the 189 failed cases in Table 5 in Appendix B.

Kernel Module Re-hosting. During the firmware emulation runs in component ②, `Pandawan` loads 14,413 (24%) kernel modules out of the 61,319 IoT kernel modules in our dataset. As illustrated in [1], the configuration and bootup scripts

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

Table 1: `Pandawan`'s plugin performance ablation study. The experiment includes every plugin combination used by `Pandawan`. The second column provides the average I_{fin} points marked by `FICD` for every plugin combination. The third column provides the performance overhead incurred by each plugin combination. The table also includes the average I_{fin} measured for FirmSolo, Firmadyne and FirmAE on the same dataset with all the plugins enabled.

dictate which kernel modules will be loaded into the kernel during the firmware initialization, which is merely a fraction of all the kernel modules in the images' file-systems. Finally, `Pandawan` executes 336 kernel module TBs on average.

In cases where kernel module crashes occur during emulation, `Pandawan` uses its custom crash solving method (see Section 4.2.3) to address the issue. In total, `Pandawan` addreses 249 kernel module errors.

User Code Re-hosting. When considering its *user* code re-hosting capabilities, `Pandawan` executes 30 user level programs and 15,671 unique QEMU TBs on average. The startup scripts within the firmware images dictate which programs are executed during bootup. We make sure to substitute symlinks with their actual targets in our results so that the TBs executed get attributed to their actual origin. For example, if the firmware executes `/bin/ls` which is a symlink to `/bin/busybox`, we only include and count the TBs executed towards the `busybox` executable in our results. Note that each TB is attributed only once (counted one time only) to their origin. For instance, a TB that belongs to a shared library and executed by multiple programs will only be counted once towards the shared library it belongs to. In Section 5.5, we discuss how these results compare against Firmadyne, FirmAE and FirmSolo.

Kernel Augmentation Ablation Study. To demonstrate the contribution of `KA` in firmware re-hosting, we also conduct an ablation study where we measure the effectiveness of the two key features of `KA`: 1) The addition of the `user-code-required` functionality by the `Oracle` and 2) the addition of the kernel symbol stubs (required by kernel modules to load into the kernel). For this experiment we use a smaller dataset of 150 firmware images (100 images without `KALLSYMS` and 50 images with `KALLSYMS`). We bias our selection towards images without `KALLSYMS` since `KA` benefits these images the most. We present the results of this study in Table 3 in the third group column. Based on our results the contribution of `KA` is apparent when taking into account all

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

Table 2: Statistics about the fuzzing experiments with TriforceAFL. The `O` and `P` in column two represent kernel modules that are *open-source* and *proprietary*, respectively. Column six provides the variance and standard deviation for the paths found by the fuzzer over the ten runs.

of our metrics. Specifically, `Pandawan` with `KA` executes 6% more user level programs and executes 11% more TBs than `Pandawan` without `KA` (i.e., FirmSolo), while also loading 3% more kernel modules and executing 17% more kernel module TBs. Finally, our experiments show that the addition of only the `user-code-required` functionality by the `Oracle` is more beneficial than simply adding the kernel symbol stubs, since the majority of the metrics are the closest to `Pandawan` with `KA`. Specifically, `Pandawan` with only the `Oracle` enabled manages to surpass `Pandawan` with `KA` in kernel module loading by 1%. Upon further inspection, two images whose the kernel modules crashed and deleted from the F_{fs} file-system (see Section 4.2.3) during the emulation with `Pandawan` with `KA`, skew the results in favor of `Pandawan` with only the `Oracle` enabled.

Plugin Overhead. To measure the performance overhead of each PyPANDA plugin, we pinpoint the I_{fin} points using every possible combination of the plugins enabled in `Pandawan`'s analysis (i.e., `syscalls_logger`, `coverage` and `SyscallToKmodTracer`) excluding `FICD` which is responsible for determining the I_{fin} points and thus is always enabled. For this study we use again a smaller dataset of 150 images (95 with `KALLSYMS` and 55 without `KALLSYMS`) that better represents the distribution of the images in our dataset. We provide the results in Table 1. When all the plugins are enabled we notice a 22% slowdown compared to enabling only `FICD`. This is expected since each plugin contributes an additional level of analysis which impacts the overall emulation speed. Specifically in our case, the most computational heavy plugins are `coverage` and `SyscallToKmodTracer` since they constantly track the execution of QEMU TBs during the emulation, incurring a slowdown of 14% and 12%, when enabled individually and a slowdown of 22% when both are enabled.

5.4 Holistic Analysis

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

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

Table 3: Comparison results between the Pandawan (P), Firmadyne (FD), FirmAE (FAE) and FirmSolo (FS) re-hosting frameworks. The table depicts the average number of user programs and QEMU TBs executed, the number of kernel modules loaded and kernel module TBs executed, for each system respectively. The green cells represent the best results for each metric. The bold values represent the cases with statistical significance, where p-value $p < 0.05$ (using the Wilcoxon signed-rank test).

Seed Generation. The `SyscallToKmodTracer` plugin monitors the interactions between the *user* and *kernel* code during the firmware emulation run in component ②. `Pandawan` processes the information collected by the `SyscallToKmodTracer` plugin (i.e., system calls that lead to kernel module code execution) and generates seeds that are used by TriforceAFL, as explained in Section 4.2.3.

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

Unlike FirmSolo which targets only the IOCTL system call, `Pandawan` can trace and generate seeds for all types of system calls that interact with kernel modules. However, we opt to create seeds only for cases involving popular system calls that interact with kernel modules through a file descriptor created either by a `sys_socket` or `sys_open` system call, due to their compatibility with TriforceAFL's fuzzing agent. We also provide details about the system calls we did not fuzz in Table 7 in Appendix C. In the end, `Pandawan` creates seeds for 479 images whose *user* code invokes system calls (see Table 6 in Appendix C) which lead to the execution of kernel module code. Out of these images, 466 (97%) have serial console connectivity, thus can be fuzzed by TriforceAFL.

Kernel Module Fuzzing. After the seed creation, `Pandawan` uses TriforceAFL to fuzz the IoT modules within the firmware images. We run all experiments ten times for 12 hours.

While `Pandawan` creates seeds for 479 firmware images, we chose to fuzz only the kernel modules in a subset of 20 randomly selected firmware images (with serial console connectivity to invoke the fuzzing harness) due to computational resource constraints. We provide the information about our fuzzing campaigns in Table 2. The table provides the average number of paths found by TriforceAFL, the path variance and standard deviation over the ten runs and the number of bugs we confirmed, for each image. Based on the path variance and standard deviation measured (see column six in Table 2), the coverage found in all the fuzzing campaigns is consistent (the variances and standard deviations are insignificant).

Specifically, TriforceAFL triggers 16 bugs in 12 kernel modules (8 proprietary and 4 open-source – see Table 2). The bugs fall into the stack corruption (6), arbitrary memory reads and writes (7), and large virtual memory allocation (3) categories. Three of these bugs (`gpio` (1) and `ipv6_spi` (2)) were also detected by FirmSolo. In addition, the seven bugs triggered for the open source kernel modules (`arp_tables` (2), `ip6_tables` (2), `ip_tables` (2), and `x_tables` (1)) are related to known bugs (`CVE-2016-4998` and `CVE-2016-3135`), which is why we did not report these bugs to the Linux kernel developers. Finally, the six remaining bugs on the proprietary kernel modules `led` (1) `ipt_STAT` (2), `gpio_module` (1), `statistics` (2) are previously unknown bugs. We have disclosed these bugs to the respective vendors, and two vendors acknowledged our findings; `TP-Link` and `DLink`.

5.5 Comparative Results

The previous evaluation based on the coarse-grained "number-of-bugs-found" metric shows that `Pandawan` finds more diverse bugs than FirmSolo. Even when it comes to bug finding speed, `Pandawan` is four times faster on average than FirmSolo (see Table 8 in Appendix E). However, `FICD` helps to assess re-hosting process on a finer granularity than just bugs found. Thus, in this Section, we evaluate `FICD`'s utility in comparing full-system re-hosting approaches based on the `emulation-based` metrics. Specifically, we compare the `Pandawan` , FirmAE, Firmadyne and FirmSolo re-hosting frameworks to answer RQ3.

As mentioned previously, the `FICD` technique requires two parameters; 1) the task similarity threshold h for the *Levenshtein* edit distance (`ed`) which `Pandawan` uses to identify previously unseen tasks, and 2) the time frame tf which indicates how long can the emulation continue without the firmware executing a previously unseen task. For our experiments, we set the threshold h equal to 0.5 (for all re-hosting frameworks) and tf equal to 300 seconds for `Pandawan` and FirmSolo and 220 seconds for FirmAE and Firmadyne, respectively. To obtain these values we initiate a ten minute experimental run with FirmAE and `Pandawan` for all the images in the dataset, without the `FICD` plugin enabled. We set the h and tf of Firmadyne and FirmSolo the same as FirmAE's and `Pandawan`'s due to the re-hosting similarities shared between the frameworks (i.e., Firmadyne with FirmAE and FirmSolo with `Pandawan`). To calculate the threshold h, we first sort all tasks based on their creation time. Then, for each task in the sorted list we use `ed` to calculate its similarity against all the tasks with an earlier creation time, as a ratio between 0 and 1. According to

Figures 4a and 4b in Appendix D, almost 90% of the tasks have a similarity less than 0.5 with any other task created earlier, for both FirmAE and Pandawan, thus we choose $h = 0.5$. Then, to get the time frame tf, we measure the average $\bar{t}$ and standard deviation (σ) of the time that passes between the execution of previously unseen tasks. Based on the ten minute experiment runs, FirmAE's $\bar{t} = 52sec$ and $\sigma = 81sec$, while Pandawan's $\bar{t} = 69sec$ and $\sigma = 115sec$. To ensure statistical relevance we choose a time frame $tf > \bar{t} + 2\sigma$, thus we set $tf = 220sec$ as our optimal time frame for FirmAE and Firmadyne and $tf = 300sec$ for Pandawan and FirmSolo, respectively.

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

Pandawan executes up to 3% more user level programs on average than Firmadyne, FirmAE and FirmSolo (see column group 1 in Table 3). Similarly, Pandawan outperforms both Firmadyne and FirmSolo on the average number of TBs executed by up to 5%. However, FirmAE executes 3% more TBs on average than Pandawan. The predominant reasons behind this outcome are twofold; 1) the emulation speed of Pandawan is slower than FirmAE's due to the SyscallToKmodTracer affecting only Pandawan (and FirmSolo) which loads and analyzes the IoT kernel modules, and 2) KA impacts negatively the re-hosting progress. Since FirmAE and Firmadyne focus solely on user level re-hosting and do not load any firmware kernel modules, the plugin's module TB tracing functionality is ineffective for these frameworks. As illustrated in Table 1 both Firmadyne and FirmAE are 13% and 4% faster than Pandawan on average. In addition, for 36 out of the 1,328 images used in the comparison experiments, the PW_k kernels produce a crash (i.e., Oops), while the FS_k kernels do not suffer from the same issue. The addition of the user-code-required functionality and the kernel symbol "stubs" by KA hinders the re-hosting progress of the images in these cases. This small regression can be attributed to the heuristic nature of KA.

When it comes to the kernel module loading we compare only Pandawan and FirmSolo, since neither Firmadyne nor FirmAE load IoT kernel modules. Specifically, Pandawan loads 6% more kernel modules and executes 5% more kernel module TB's on average than FirmSolo. To be fair towards FirmSolo we addressed the crashes during its emulation runs using Pandawan's crash solving methods (see Section 4.2.3). In this case, we address 225 crashes in total.

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

Table 4: Coverage information of popular IoT webservers.

Since Pandawan only includes the user-code-required functionality in images without a KALLSYMS entry (see Section 3.2.2), the improvements of Pandawan over FirmSolo can be better observed on these images (see column group 2 in Table 3). In particular, Pandawan executes 6% more user programs and 21% more TBs on average than FirmSolo. Pandawan also outperforms FirmSolo on the number of kernel modules loaded and kernel module TB's executed by 9% and 26% on average, respectively. Both the user-code-required functionality and the symbol "stubs" added by Pandawan in the PW_k kernels for these images, greatly benefit the progress of holistic (*user* and *kernel* code) re-hosting as well as the loading of additional kernel modules.

We also test the statistical significance of our results using the Wilcoxon signed-rank test [45]. Specifically, for each metric we compare Pandawan's data with Firmadyne's, FirmAE's and FirmSolo's data (28 comparisons). We exclude the kernel module related metrics (loaded and TBs executed) for Firmadyne and FAE since they do not load the IoT kernel modules. We note that the majority of our measurements (24/28) are statistically significant with a p-value $p < 0.05$ (see Table 1).

Webserver Re-hosting. To further showcase the efficiency of Pandawan in *user* code re-hosting, we conduct a study about the re-hosting of webservers used by firmware images. We provide the results in Table 4. Specifically, we collect coverage information for 6 popular webservers used in IoT [28]. Our findings show that FirmAE and Firmadyne which are the state-of-the-art in *user* code re-hosting marginally outperform Pandawan in re-hosting webservers by executing up to 8% more QEMU TBs across all the servers in our study. Given the fact that Pandawan aims to holistically re-host and analyze Linux-based IoT firmware, Pandawan's slight deficiency in webserver re-hosting is an acceptable tradeoff. We leave the qualitative analysis of the differences in webserver re-hosting between the compared frameworks as future work.

6 Discussion

In this Section we discuss future applications of FICD and Pandawan. With the introduction of FICD, developers can efficiently improve upon the existing works by comparing their implementations with the state-of-the-art. Furthermore, Pandawan could be leveraged to develop future frameworks whose re-hosting capabilities closely approximate physical IoT devices. For instance, the re-hosting techniques imple-

mented in Honware [48] could be integrated into Pandawan to further improve its holistic re-hosting capabilities. In addition, the FICD and setup of Pandawan (i.e., using PyPANDA for firmware emulation) are generic and OS agnostic. Thus, any firmware that can be emulated by QEMU and uses a concept of process/task (e.g., FreeRTOS [22]) can be adapted to leverage FICD. Next, we explore the categories of metrics that could be used to quantify re-hosting progress and lastly specify certain limitations of both FICD and Pandawan.

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

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

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

Limitations. As far as FICD's limitations are concerned, the time frame tf is directly dependent on which (Py)PANDA plugins are enabled during a firmware emulation run. When many computationally heavy plugins are enabled, the emulation speed will decrease, thus tf requires an adjustment. In our re-hosting experiments we use three (Py)PANDA plugins (`coverage`, `syscalls_logger`, and `SyscallToKmodTracer`) along with the FICD plugin, which results in the optimal $tf = 220sec$ and $tf = 300sec$ for FirmAE and Pandawan, respectively. Furthermore, Pandawan, like other re-hosting frameworks, faces specific limitations. Since the *Oracle* targets only the data structures used by open-source kernel modules with a counterpart in UP_{kos}, it does not guarantee that the layout of data structures used by proprietary kernel modules will be unaffected. The options produced in S4 can potentially misalign the layout of data structures used by proprietary kernel modules causing the modules to crash during emulation. Furthermore, as shown in Section 5.3 only a fraction of the total modules in the F_{fs} file-systems are loaded during emulation by the firmwares' bootup scripts. Consequently, the SyscallToKmodTracer plugin only collects system call information for a subset of user level programs that interact with the loaded kernel modules during emulation. The kernel modules that are not loaded and user level programs that are not executed during the emulation do not contribute to the holistic firmware analysis.

7 Related Work

Benchmarking. These studies are important for establishing reliable metrics for evaluating the contributions of works and also setting the standards for future directions in a research area. Both [29, 31] introduce metrics to evaluate the performance and effectiveness (i.e., bug finding) of fuzzing tools as well as guidelines that should be followed by future fuzzing applications. AIR [52] proposes a metric to quantify the protection offered by Control Flow Integrity techniques on binary executables. Similar to these works, FICD uses specific metrics (i.e., the number of user level, user and kernel module code coverage and number of kernel modules loaded) to evaluate different approaches in the full-system re-hosting domain.

IoT Firmware Analysis. There are plenty of academic works focusing on firmware analysis. On the one hand, static analysis and symbolic execution [16, 25, 43, 44] are two of the most prominent techniques for firmware analysis. While some works rely solely on static analysis [13, 20] to analyze IoT firmware, it is mostly used to aid symbolic executors [5, 6, 10] to explore firmware code more efficiently.

On the other hand, dynamic analysis has also gained a lot of popularity in the IoT firmware analysis landscape in the past few years. Firmadyne [7] and FirmAE [28] are two examples of frameworks which use firmware re-hosting at their core to emulate IoT firmware and subject it to various dynamic analyses, such as vulnerability and bug testing. Similarly, Honware [48] relies on re-hosting techniques to deploy emulated IoT firmware as honeypots on the Internet to observe and study real world network attacks. FirmGuide [32] is a re-hosting framework that semi-automatically creates QEMU peripheral models to successfully re-host the Linux IoT kernels of open-source firmware images (e.g., OpenWRT). However, both the need for human intervention and source code availability renders FirmGuide unable to re-host proprietary firmware, unlike Pandawan. Finally, FirmSolo [1] is a framework that re-hosts and enables the dynamic analysis of IoT firmware binary kernel modules.

In contrast to Pandawan, which holistically re-hosts and analyzes IoT firmware, most of the works above focus exclusively on either the user or the kernel level aspect of IoT firmware.

Hardware-In-The-Loop. Frameworks in this category combine both a physical IoT device and re-hosting to dynamically analyze binary IoT firmware. Specifically, AVATAR [50] leverages the debugging interface (i.e., JTAG) of IoT devices and the QEMU emulator (driven by a symbolic execution engine) to forward I/O operations to the physical device while the firmware code is executed within the emulator. Similarly, SURROGATES [30] and Inception [12] follow a similar logic. While effective at analyzing IoT firmware, these systems are intrinsically limited in terms of scalability due to their

dependence on the IoT hardware.

Firmware Fuzzing. Both Greenhouse [46] and EQUAFL [55] rely on user mode QEMU to emulate user level firmware applications and analyze these applications through fuzzing. EASIER [38] is a framework that loads binary kernel modules on pre-configured Android kernels within an emulated environment and analyzes these modules with AFL [51]. FirmAFL [54] combines a hybrid firmware re-hosting technique (user and full system mode) with fuzzing to analyze user level firmware code. Fuzzware [41] is a system that relies on dynamic symbolic execution (DSE) to infer hardware created values (provided to the firmware through MMIO) which are used to drive a fuzzer while analyzing the embedded firmware code. On a similar fashion, Halucinator [11] uses Hardware Abstraction Layers (HALs) to model hardware peripherals along with a fuzzer to dynamically analyze embedded firmware.

Unlike Pandawan, these systems are unable able or only suitable of analyzing specific categories of firmware kernel modules. Pandawan can load and analyze a variety of binary Linux IoT kernel modules, due to its holistic analysis capabilities.

8 Conclusion

In this work, we present `FICD`, a technique that enables the objective comparison of full-system re-hosting approaches on their emulation capabilities. Also, we introduce `Pandawan`, a framework that enables holistic re-hosting and analysis of Linux-based IoT firmware. `Pandawan`, implements the `Kernel Augmentation` technique which produces kernels conducive to holistic re-hosting and analysis. We use `Pandawan` to holistically analyze IoT firmware and provide the information produced to the TriforceAFL fuzzer to analyze the firmware's kernel modules. Finally, we rely on `FICD` to showcase `Pandawan`'s progress in firmware re-hosting by comparing it with the Firmadyne FirmAE, and FirmSolo re-hosting frameworks.

9 Acknowledgements

We would like to thank our shepherd and anonymous reviewers for their valuable comments and feedback. This research was enabled by the National Science Foundation under grants CNS-1942793, CNS-1916393, CNS-2127232 and Red Hat grant 2024-01-RH06. The computational work reported in this paper was performed on the Shared Computing Cluster, administered by Boston University's Research Computing Services.

References

[1] 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.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai Botnet," in *Proceedings of the USENIX Security Symposium*, 2017.

[3] N. Bars, M. Schloegel, T. Scharnowski, and N. Schiller, "Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge," in *Proceedings of the USENIX Security Symposium*, 2023.

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

[5] C. Cadar, D. Dunbar, and D. 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.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2012.

[7] 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.

[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[9] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

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

[12] 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.

[13] 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.

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

[15] D. Ganesan, "Lack of Standardization can Halt the IoT Juggernaut," https://www.dqindia.com/lack-of-standardization-can-halt-the-iot-juggernaut/, 2017.

[16] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *Proceedings of the USENIX Security Symposium*, 2013.

[17] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable Reverse Engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)*, 2015.

[18] 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.

[19] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference," 2021.

[20] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, "Static Analysis for Discovering IoT Vulnerabilities," *International Journal on Software Tools for Technology Transfer (STTT)*, 2021.

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[22] FreeRTOS, "Freertos: Real-time operating system for microcontrollers," https://www.freertos.org/index.html, 2023.

[23] Google, "Syzkaller," https://github.com/google/syzkaller, 2023.

[24] M. Harris, "Where should companies start when it comes to device security?" https://www.helpnetsecurity.com/2022/03/31/devices-security/, 2022.

[25] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. 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.

[26] J. Howarth, "80+ amazing iot statistics," https://explodingtopics.com/blog/iot-stats, 2023.

[27] A. Junnila, "How iot works – part 4: User interface," https://trackinno.com/iot/how-iot-works-part-4-user-interface/, 2023.

[28] 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.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[30] 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.

[31] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers," in *Proceedings of the USENIX Security Symposium*, 2021.

[32] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren, "Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[33] Malwarebytes, "Trickbot," https://www.malwarebytes.com/trickbot, 2023.

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

[35] T. Newsham, "Triforcelinuxsyscallfuzzer," https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2017.

[36] Owasp, "Owasp internet of things project," https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project, 2018.

[37] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *Proceedings of the USENIX Security Symposium*, 2018.

[38] 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.

[39] Rapid7, "Metasploit framework," https://www.metasploit.com/, 2023.

[40] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "kneedle" in a haystack: Detecting knee points in system behavior," in *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2011.

[41] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *Proceedings of the USENIX Security Symposium*, 2022.

[42] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds," in *Proceedings of the USENIX Security Symposium*, 2022.

[43] 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.

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

[45] S. Solutions, "Understanding the wilcoxon signed rank test," https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/how-to-conduct-the-wilcox-sign-test/, 2024.

[46] 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 *Proccedings of the USENIX Security Symposium*, 2023.

[47] O. Y. V. Ganti, "A brief history of the meris botnet," https://blog.cloudflare.com/meris-botnet/, 2021.

[48] 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.

[49] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing," *KSII Transactions on Internet and Information Systems (TIIS)*, 2013.

[50] 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.

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

[52] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the USENIX Security Symposium*, 2013.

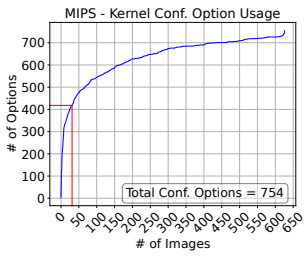
[53] W. Zhao, K. Lu, Q. Wu, and Y. Qi, "Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.

[54] 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.

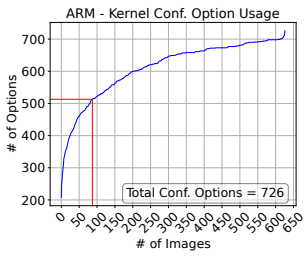
[55] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

A User-Code-Required Functionality

In Figures 3a and 3b we showcase the popularity of the configuration options in the O_{MIPS} and O_{ARM} pools, respectively.



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



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

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

B Console Connectivity Issues

Table 5 provides the breakdown of the reasons and the number of cases behind Pandawan's console connectivity issues

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

Table 5: Console connectivity issues during the Pandawan emulation experiments. Column one illustrates the reasons behind the console connectivity issues.

C System Calls Traced Statistics

In Table 6 we present the twelve system calls that lead to kernel module code execution and are explicitly fuzzed by Pandawan (using TriforceAFL). We opt to fuzz only these system calls due to their inherent compatibility with TriforceAFL's fuzzing agent (harness), TriforceLinuxSyscallFuzzer [35]. Even though the fuzzing agent is capable of invoking any system call, it primarily supports socket or file-descriptor based system calls (see Table 6).

Table 7 illustrates the top ten traced system calls, which lead to kernel module code execution, but not fuzzed during our fuzzing experiments. We opt out from fuzzing these system calls for two reasons. First these system calls do not operate on a file descriptor or socket (i.e., the most compatible system calls with TriforceAFL's fuzzing agent). Second, invoking these system calls does not lead to kernel module code being executed until certain conditions are met. For example, sys_select is used to monitor file descriptors until they are ready for I/O, which the fuzzer is not sophisticated enough to generate. Also fuzzing this specific system call would not produce any interesting results (i.e., crashes). The same premise is true for the rest of the system calls we did not fuzz.

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

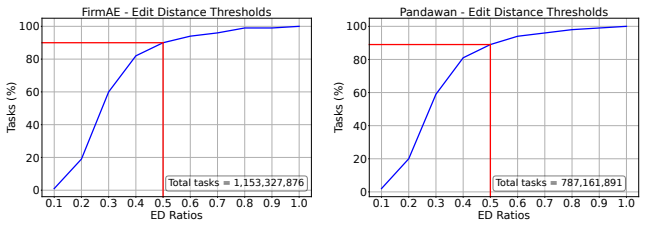
Table 6: The most popular system calls used in Pandawan's fuzzing experiments.

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

Table 7: The top 10 traced system calls that are not "compatible" with TriforceAFL.

D Edit Distance Threshold Experiments

Figures 4a and 4b depict the results of the ten minute edit distance threshold experiments for FirmAE and Pandawan.



(a) FirmAE edit distance threshold experiment.

(b) Pandawan edit distance threshold experiment.

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

E FirmSolo Bug Finding Speed

Table 8 depicts the bug finding speed measurements for Pandawan and FirmSolo. For Pandawan's experiments we use the images where TriforceAFL triggered at least a bug in one of their kernel modules. For FirmSolo' experiments, we choose images that load the same kernel modules that FirmSolo originally found bugs in (except rt_rdm.ko which did not load in any of our images). We run the experiments 10 times for 12 hours and the results are the averages over these runs. The measured Time-To-Crash (TTC) variances and standard deviations for both Pandawan and FirmSolo (see columns three and six in Table 8) indicate that TriforceAFL is inconsistent regarding the time it requires to arrive to the first crash/bug.

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

Table 8: Pandawan's and FirmSolo's bug finding speed measurements. TTC in column two and four stands for Time-To-Crash (i.e, the time it takes the fuzzer to reach the first crash). Columns three and six provide the measured variance and standard deviation for TTC. Regarding the led.ko kernel module, in line four, TriforceAFL registers the bug as a *hang* and not a *crash*, hence there is no TTC in this case. The art.ko kernel module in line two crashed while being loaded into FirmSolo's kernels and thus the measurement is invalid.