Informatics

Firmware Re-Hosting

An Evaluation and Verification of FirmAE

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Sebastian Dietz
Registration Number 11816257

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc
Christian Kudera, BSc MSc
Michael Pucher, BSc MSc

Vienna, 25th October, 2022

Sebastian Dietz   Edgar Weippl

Erklärung zur Verfassung der Arbeit

Sebastian Dietz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Oktober 2022

Sebastian Dietz

Abstract

Firmware re-hosting has been getting more attention as its use cases in developing embedded systems and security analysis are invaluable. In this thesis, we compare five state-of-the-art tools based on the properties ideal firmware re-hosting solutions must have and verify the results of the firmware emulation framework FirmAE. FirmAE is a fully automated dynamic analysis framework for Linux-based systems and extends the Firmadyne framework by implementing heuristics based on failure case analysis. We validated the published results using the publicized dataset and constructed a new set consisting of images from the top vendors on home networks. The firmware collection was then used to evaluate the overall emulation success rate. In addition, the impact of each arbitration technique was assessed. Our results show that FirmAE increases the emulation success rate of Firmadyne from 3.08% to 32.3%. Regarding the impact of each arbitration, the categories network and boot seem to have the most influence, reducing the emulation success rate by an average of 24% and 20% when disabled. NVRAM arbitration seems to be the least important, reducing the rate by about 4% across the board.

Contents

Abstract **v**

Contents **vii**

1 Introduction **1**

2 Background **3**
- 2.1 Embedded Systems 3
- 2.2 Firmware 4
- 2.3 Peripherals 4
- 2.4 Device Classification 5
- 2.5 Emulation 6
- 2.6 Analysis Techniques 7
- 2.7 Re-hosting 7

3 State of the Art **9**
- 3.1 Surveyed Frameworks 9
- 3.2 Challenges 12
- 3.3 Comparison 20

4 FirmAE Verification **23**
- 4.1 Problem Statement 23
- 4.2 Experimental Setup 24
- 4.3 Evaluation 25

5 Conclusion **33**

List of Figures **35**

List of Tables **37**

Bibliography **39**

CHAPTER 1

Introduction

The attack surface also rises with the ever-increasing number of connected embedded devices. As a result, new attacks and malware are being discovered frequently[1]. As many of these devices store sensitive information, protecting embedded systems from malicious actors is of utter importance. In the past, emulation was used during the development of embedded devices to reduce the need for physical hardware. This gave hardware developers a cost-efficient way to test the behavior of their systems. Just as in development, emulation can be used in security research to gain a better understanding of a given system. In addition, it enables new ways in which security researchers can interact with a target, which would have been impossible on a physical machine. Being able to interact with the board on various abstraction layers dramatically accelerates the discovery and mitigation of vulnerabilities in Internet of Things (IoT) devices [2]. Furthermore, re-hosting can be used by IoT honeypots [3] in order to learn more about upcoming cyber attack trends.

However, previous studies have shown that there are still some unsolved problems related to firmware re-hosting, and a one-size-fits-all solution is hard to achieve [2]. Therefore, we looked at five different state-of-the-art firmware re-hosting frameworks and compared them by discussing how each tool tries to overcome these challenges. In addition, we evaluated them over the properties an ideal firmware re-hosting solution must have and discussed the different approaches each of the tools takes. One of those tools is FirmAE. FirmAE is an extension of Firmadyne and claims to increase the emulation success rate from 16.28% to 79.36% by implementing five different arbitration techniques. Kim et al. [4] states that their heuristics were developed to handle failure cases empirically and may not apply to new devices and configurations. Still, the sample set used in the research seems deliberately chosen to reflect their research results. For example, firmware images from the manufacturer NETGEAR have an emulation success rate of 93.80% while being part of 24.3% of all evaluated images. Therefore, we constructed a new sample set to try to reproduce the results of their study. For this, we first verified the research results of Kim et al. [4] by using their published firmware samples. Next, we constructed a new collection of images containing the top manufacturers used in home networks using the results of Kumar et al. [5]. The newly created dataset is then used to evaluate FirmAE and to gain more insight into the success rate as well as the impact each arbitration has.

We start in chapter 2 by briefly introducing the terminology that comes with the topic and discussing different re-hosting approaches. In the next chapter, we continue by introducing the surveyed frameworks and discussing some issues with firmware re-hosting and emulation. We compare them in section 3.3 based on the properties introduced by Gustafson et al. [6], as well as how they handle each challenge. We then proceed with the verification of FirmAE, where we begin by describing our experimental setup and the arrangement of the firmware samples. We finish by discussing each of our results and a summary in section 5.

CHAPTER 2

Background

Before getting into the current problems of firmware re-hosting, we would like to give an overview of the targets, methods, and technologies this research field has to work with. Mainly, we provide the necessary background knowledge for the remainder of this thesis.

2.1 Embedded Systems

Embedded Systems can be found in anything the general public calls 'electronics'. From various commercial off-the-shelf (COTS) devices such as printers, home appliances, peripherals, and smartphones, to less consumer-oriented systems such as video surveillance systems, supervisory control, medical implants, military systems, and programmable logic controllers (PLCs) [7].

Because of their broad application, a general and precise description of an embedded system is hard to define. In addition, embedded systems vary wildly in terms of hardware, computing power, purpose, and costs. Making a concrete definition even harder. In the context of this thesis, we adopt the idea of Muench et al.[7] that embedded systems differ from modern general-purpose computers in two distinct points:

- Embedded systems are designed to fulfill a special purpose
- Embedded systems often interact with the physical world through a number of peripherals connected to sensors and actuators.

It is important to note that an embedded system can either be self-contained or consist of several embedded devices. For reasons of simplification, we will use the term embedded systems interchangeably with embedded devices, as the proposed solutions focus on single device re-hosting rather than interconnected systems.

2.2 Firmware

Muench [7] defined the term by stating that firmware is the software needed by embedded systems in order to carry out a specific task. Because the terms software and firmware are so similar, Muench distinguished them in three distinct points:

- First, firmware directly interacts with the underlying hardware, while traditional software uses application programming interfaces (APIs), libraries, or abstractions of the target operating system.
- Second, firmware is mostly stored in read-only memory (ROM) or non-volatile memory chips. Therefore, firmware is normally installed by the manufacturer rather than the user.
- Third, Muench stated that well-defined executable formats, for example, as seen on desktop systems, are the exception. Firmware mostly comes in one complete binary or 'blob' file, which contains all the information to ensure the device's functioning. Data, code, and metadata are often interleaved, and the entry point address used at the execution's start may be hardcoded inside the firmware such that the system's processor can directly access it.

Another approach would be to see firmware as *the art and the technique of transforming hardware (logic systems) into software (programs) and vice versa* [8]. While this artistic definition of Daniel Mange certainly holds, we will use the interpretation of Muench for this thesis instead.

2.3 Peripherals

Peripherals are responsible for input/output as well as the general interaction with the physical world [7]. Peripherals can be distinguished between on-chip and off-chip peripherals. If manufacturers combine the central processing unit with several peripherals, the device's peripherals are called on-chip. Such devices are also named System-on-Chip (SoC). The distinction between on-chip and off-chip is essential, as systems with off-chip components need special on-chip peripherals, such as Universal Synchronous/Asynchronous Receiver/-Transmitter (USART), to communicate with the off-chip hardware. According to Muench [7], even though the interaction between the CPU and on-chip peripherals is dependent on the CPU itself, they usually fall into one of the following categories:

Memory-Mapped Input/Output (MMIO). The hardware registers of a peripheral are directly mapped into the system's memory space. That means that the data, configuration, and status, can be accessed by reading and writing to special memory locations

Port-Mapped Input/Output (PMIO). Here, the peripherals are mapped onto ports on the embedded system. With special instructions, such as `in` and `out`, the Instruction Set Architecture (ISA) enables communication with those ports.

Interrupt Requests (IRQs). Interrupts are being issued by the peripheral and notify the CPU that an event needs its attention. When an IRQ arrives, the processor saves its current state and changes execution to the interrupt handler (ISR). When completed, the CPU restores its previous state and continues. Nowadays, CPUs often implement a more complex interrupt service routine, which allows nesting of interrupts, assigning priorities to them, and selectively disabling and enabling them [7].

Direct Memory Access (DMA). Allows the exchange of data while the CPU can execute other tasks. This method requires the presence of a dedicated DMA controller. The DMA controller is a specialized hardware component and peripheral on its own, capable of transferring data between other peripherals and main memory independent of the CPU. In most cases, the DMA controller notifies the CPU about completed data transfers by issuing an interrupt. [7]

2.4 Device Classification

As we have stated multiple times in previous sections, the sheer diversity found in embedded systems is enormous. While we have already defined the term embedded systems, the introduced definition still cover various devices. These systems could be classified by several aspects, such as cost, computing power, or the extent of environmental communication. However, the challenges and techniques in emulating or re-hosting these embedded devices vary and may not translate well from one system to another. Hence, we will categorize them by the type of firmware they execute. These classifications are based on Wright et al. [2] and Muench [7]:

General Purpose Embedded Systems (GPES). These systems are often retrofitted to the embedded system space. These systems will often run on a Linux OS Kernel with a modified lightweight userspace environment, such as busybox or uClibc. Custom peripherals or external hardware is often carried out via device drivers. Because of the abstractions provided by the operating systems, this class makes a great target for firmware re-hosting frameworks. Such systems include real-time Linux, embedded Windows, and Raspberry Pi [2].

Special Purpose Embedded System (SPES). are systems with custom OS specifically developed for the embedded world. As these devices are often commercial products, the operating system is usually closed-source. While advanced processor features such as a Memory Management Unit (MMU) may not be present, a logical separation between the kernel and userspace still exists. However, this line is often blurred in reality, which can add difficulty when trying to emulate such kind of system. In addition, many emulation techniques from the personal computer space do not work, and emulation must start from scratch. Single-purpose user electronics, such as LTE modems or DVD players, are examples of these systems and usually run on operating systems such as uClinux, ZephyrOS, or VxWorks.

Bare-Metal Embedded Systems (BMES). These devices do not have a typical separation of the OS and the firmware, resulting in a so-called "monolithic firmware". The code running on such targets can be completely custom and is often based on a single main loop and interrupts from the peripherals. These approaches can often be found in microcomputers (e.x, Arduino, STM32), wifi cards, or GPS dongles.

2.5 Emulation

Emulation is the process of imitating a system on another device. Often emulation gets confused with simulation. Simulation can sometimes be encountered in scientific modeling, but in this context, simulation is another technique used to model the internals of a system. To be specific, simulation is the process of modeling a system by implementing parts of the internals in software. In contrast, emulation is the process of modeling the system by replacing the internals. Nowadays, the distinction is mostly unimportant, and the surveyed frameworks either use the terms interchangeably or refer to their approach as emulation based. When it comes to emulators, one of the first popular ones was Simics [9]. In the early 2000s, the tool was created to emulate multiple architectures, provide scripting and configuration management, and automated debugging. In addition, Simics is designed around a high instruction level fidelity, which allows for interrupt testing between any pair of instructions. On the contrary, tools like QEMU [10] give up some of their accuracies in order to improve emulation speed. QEMU emulates by translating entire blocks of instructions to the host systems instruction set. This allows for a better performance as QEMU can cache these basic blocks and does not need to check for interrupts after each instruction. Because of its open-source license and support for many architectures and peripherals, QEMU has become one of the staples in academia and for industry professionals. This is also the reason why QEMU and Simics are two of the most widely used emulators. [2]

Fidelity. In the context of this thesis, the expression *emulation fidelity* will sometimes occur. The term got introduced by Wright et al. [2] and describes how closely execution in the emulator can match that of the physical system. Emulation fidelity can be categorized into the following levels from 1 to 7: *Blackbox*, *Module*, *Function*, *Basic Block*, *Instruction*, *Cycle* and *Perfect*. When observing a system with *Blackbox* fidelity, the same external behavior can be observed as the real system, while internally, it may not execute the same instructions. At the *Module* level, some parts of the firmware are executed unmodified, whereas others could be completely replaced. At *Function*, it can happen that whole functions need to be replaced for emulation success. Similar, *Basic*

Block and *Instruction* accurately emulate at the basic block or instruction layers. The *Cycle* level faithfully emulates the instruction cycle. *Perfect* fidelity would mean that the emulation behaves exactly like the actual hardware.

2.6 Analysis Techniques

Recently analysis techniques like fuzzing or symbolic execution have been adopted by re-hosting solutions to infer hardware peripherals [11][12]. Because these methods are integrated into some frameworks, it requires some familiarity to discuss these topics. Therefore, we do not go into depth in these areas but rather give a brief overview and explanation of these techniques.

Fuzzing. The goal of fuzzing is to stress the target with random input such that unexpected behavior like crashes or resource leaks occur. Because of that, fuzzing is often used as a bug and vulnerability-finding technique. Furthermore, in the context of emulation, fuzzing can be used to improve the code exploration of the firmware or even to learn about unknown peripheral access [2].

Symbolic Execution. Instead of supplying the program the normal inputs, one supplies symbols (i.e., variables) representing arbitrary values. The execution proceeds as normal except that values may be symbolic formulas over the input [13]. When solving these symbolic formulas, a set of constraints defining the value the input must have to reach a specific part of the program is returned. This means that these symbols can be used to describe all program execution paths than can be executed. [2]

Concolic Execution. Concolic Execution is when the execution engine switches between using symbolic and concrete values during execution. This technique reduces the usual huge, or often infinite, search space of traditional symbolic execution [14]. In terms of firmware re-hosting and emulation, code found on embedded systems is sometimes dependent on the return value of peripheral access. By solving these hardware dependencies in symbolic space, researchers are able to reach code paths that would not be reachable otherwise. [2]

2.7 Re-hosting

Re-hosting is described by Wright et al. as the act of executing a binary on a host system using system emulation, which would otherwise need to be run on specific hardware [2]. Re-hosting can be achieved in various forms. Therefore, we present three options that may be available to the practitioner and discuss each method's advantages, disadvantages, and liabilities.

Full Firmware Re-hosting. Full re-hosting tries to build a fully-featured emulator from firmware and the metadata information from either the embedded system or firmware sample. Therefore, complete system emulation can be achieved without the need for physical hardware. While this approach might look promising, Wei Zhou et al. [12] showed that these emulation types frequently fail to execute complex firmware samples properly. For example, they stated that P^{2}IM's [15] heuristic of guessing the correct response to a peripheral read request is impractical when considering the large search space. Further, some frameworks might come with dependencies that only a handful of firmware might supply. For instance, firmadyne [16] can only emulate firmware that runs on GPES Systems with a Linux OS Kernel.

Partial Firmware Re-hosting. This approach attempts to construct an emulator from the firmware only. That means that because no additional auxiliary information about the peripherals is being used, the derivated emulator is not guaranteed to be complete, and only a handful of peripherals are available [11].

Physical Re-Hosting. Here, the binary code from a device is relocated to a new, more test-friendly device. If the new device is cheaper and more readily available than the original one, this may improve scalability. However, transferring the program to a completely different system has significant difficulties. On top of that, it may be difficult to reproduce bugs found on the original device due to the different architecture or changes the binary transfer may have introduced [7]. Hence, bugs that were present (or not) on the original target may not (or are) present on the new target. It should be pointed out that physical re-hosting does not fall into our definition of re-hosting. Still, for the sake of completeness, we wanted to include it.

Hardware-in-the-loop Re-hosting (HIDL). With the help of HIDL, it is possible to attach a software emulator to a physical machine directly. HIDL allows forwarding the I/O between the hardware and emulator, resulting in the highest fidelity emulation since we can directly communicate with the peripherals [11]. However, as the embedded system needs to be available and obtainable, these hardware-in-the-loop approaches cannot be used for large-scale re-hosting [12].

It should be noted that according to Muench [7], there is no distinction between hardware-in-the-loop and partial firmware re-hosting. However, since the proposal of Jetset [11], separating these two methods is necessary and valuable, as seen later in the framework comparison section.

CHAPTER 3

State of the Art

In this section, we go into the details of the current problems re-hosting has to face, as well as compare different state-of-the-art frameworks that are currently released. We start by introducing the five frameworks and give an overview of the different approaches and techniques these tools developed. Next, we dive into the problems firmware re-hosting has to work with and divide them by the time they occur during the emulation process. Here, we also discuss how each surveyed framework tries to overcome these challenges. In the end, we compare the frameworks by evaluating them over properties that ideal firmware re-hosting solutions must have and discuss each tool individually.

3.1 Surveyed Frameworks

Pretender. Pretender [6] records the interactions between the hardware and the firmware. Next, their machine learning engine creates a stateful behavior model of the peripherals and builds an emulator for this model without being dependent on the hardware afterward [12]. To do this, Pretender [6] goes through five phases. First, traces of MMIO region accesses get recorded. After that, the memory space boundaries of each peripheral get located to help divide the recording into sub-recordings. Next, based on the interleaving of interrupts with MMIO, Pretender [6] assigns each numbered interrupt event to a peripheral group. This is used to create timing patterns for later emulation. In the last two steps, a memory model for each memory location is created, and the way of system input is getting specified. If an MMIO region gets accessed during emulation, the pre-recorded traces get searched for accesses. The model behaves exactly like the trace if feasible access is found, resulting in either a returned value (`read`) or a written register (`write`).

Firmadyne. Firmadyne [16] is a fully automated dynamic analysis framework for Linux-based General-Purpose Embedded Systems. While the platform comes with its own firmware web scraper, extractor, and dynamic analysis engine, we will only evaluate the emulation solution. Firmadyne [16] uses *QEMU* [10] for full system emulation. To boot the firmware, firmadyne [16] uses its modified kernel image and user-space libraries to satisfy the required conditions. For example, about 52.6% of all extracted firmware images access a hardware non-volatile memory (NVRAM) location using a shared library [16]. As these memory locations often contain device-specific configurations, access to such values must be possible. To solve this issue, firmadyne [16] hooks NVRAM-related functions, such as *nvram_get* and *nvram_set*, to allow reimplementing this interface in userspace without emulating hardware-specific peripherals. Aside from NVRAM, hardware-specific peripherals such as watchdog timers or additional flash storage devices may be expected by the firmware image [16]. Normally, such functionality would be implemented using device drivers in kernel space and could therefore be intercepted by firmadyne's [16] custom kernel image. However, as some device manufacturers do not follow good software engineering practices, Firmadyne [16] modified sixteen bytes in *QEMU's* [10] source code to respond to user-space implementations. In addition, a 60-second initial learning phase is used to learn about the required networking configuration. In particular, firmadyne [16] keeps track of assigned IP addresses, IEEE 802.1d bridges, and IEEE 802.1Q VLAN tagging. The learned information is then used to build a more accurate *QEMU* [10] instance for the system.

FirmAE. FirmAE [4] extended the firmadyne [16] emulation framework by implementing additional heuristics and so-called *arbitrated emulation*. The goal of *arbitrated emulation* is to ensure sufficient high-level behavior to allow dynamic analysis on user-space programs [4]. Kim et al. [4] implemented this by analyzing the high failure rate of firmadyne [16] and coming up with different heuristics that each increase the emulation rate of the dataset. Firstly, they improved the success rate by implementing boot arbitrations. They discovered that because some firmware has unusual init paths, the kernel could not find the initialization program and panicked. However, as the init path is often found in the kernel command line strings, extracting the path was easily manageable by a custom-written script [4]. Other boot failure cases occurred due to missing files or directories. To solve this issue, FirmAE [4] extracts all path-like strings from executable binaries and prepares the file system based on these paths. Secondly, to allow dynamic analysis, the network needs to be configured properly such that the analyst can communicate with the re-hosted system. By running the guest system twice, FirmAE [4] is capable of acquiring the required network configuration from the system's initial startup logs and configuring VLAN, TAP interfaces, and iptables accordingly [4]. However, not all firmware images contain such information. Therefore, FirmAE [4] forcefully configures the Ethernet interface, *eth0*, of such images to a default config. Thirdly, FirmAE [4] extends Firmadyne's [16] already present approach of initializing present NVRAM files by extending the list of known values per manufacturer. Lastly, small changes, for example, executing the webserver manually, increased the emulation success rate as well.

Jetset. The motivation behind Jetset [11] is that for analysis, most of the time, only a small part of the system needs to be emulatable. Further, full re-hosting often comes with considerable challenges. Therefore, Johnson et al. [11] idea were to use symbolic execution to infer the correct behavior the firmware expects from peripherals. To do this, Jetset [11] requires the executable code, the memory layout of the system, the entry point address, and a so-called *goal address*. The memory layout and the entry point address can often be found in the datasheet of a system. The executable code is found in the firmware image, and the goal address specifies the instruction the analyst wants the emulator to reach. Jetset's operation follows a two-stage principle; First, Jetset uses symbolic execution to learn about the system's expected behavior. Next, the output of the first stage is used to build an appropriate *QEMU* [10] device. These two stages are called *peripheral inference* and *periperhal synthesis*. During the inference stage, Jetset [11] executes the code in a custom symbolic execution environment [11]. As long as the code is executed in this environment, all reads from MMIO (see section 2.3) addresses are handled as symbolic. This means that when the *goal address* is reached, a set of constraints leading to this address is available [11]. To keep the number of explorable paths small, Jetset [11] uses *guided symbolic execution*. Here, a distance is calculated from each basic block of the control flow graph to the goal. This helps branch decisions, as Jetset [11] will always use the next basic block with the lowest distance to the goal address. Because some embedded systems might need interrupts to reach the goal address, Jetset [11] periodically executes all Interrupt Service Routines (ISR). When finished, Z3 [17], the default SMT solver used by angr [18], is used to generate appropriate values for each read that, in the end, allows building a synthetic device. This device answers each MMIO read with a concrete value that guides the execution to the goal address [11].

µEmu. µEmu [12] tries to find bugs in code related to improper input handling of Input/Output Interfaces. µEmu [12] tries to emulate those interfaces by automatically generating appropriate responses. A response is considered appropriate when the response passes the firmware's internal checks so that the execution reaches a usable state for practical security analysis [12]. For each firmware, a knowledge extraction phase is run. During this phase, a knowledge base about how to respond to peripheral register access is built. With the help of symbolic execution, unknown peripheral register accesses get logged and, when influencing a branch decision, calculated via a constraint solver. When solved, the appropriate value gets cached in the knowledge base (KB). The cached values are used to help the symbolic execution engine in deciding the next branch target. To make this more efficient, µEmu [12] adopts a tiered caching strategy. These tiers get accessed via matching rules. When an unknown peripheral gets read, a matching rule finding as many similar peripheral accesses as possible is executed. If the cached values are wrong, µEmu [12] rejects them and upgrades the matching rule for the corresponding peripheral register [12]. After exploring all paths, µEmu [12] is ready for emulation. Should a register of a custom-made peripheral be accessed, the knowledge base gets referred, and an appropriate response value is returned to the emulation [12].

3.2 Challenges

As firmware often comes in the form of a raw binary blob, the way this blob is handled lies entirely in the hands of the processor's hardware. This way of handling will consequentially vary widely from system to system [6]. Without this abstraction, even unpacking or extracting the firmware might bring challenges. The consequence of this missing abstraction is that the execution environment for firmware is the hardware itself [6]. That means that for re-hosting firmware, we need to break down the hardware we want to emulate. For that, two unique categories can be pointed out. We base our approach on the work of Gustafson et al. [6] while summarizing on-chip and off-chip peripherals into one group:

- **CPU core** For this, the instruction set and the interrupt controller need to be emulated.
- **Peripherals** These include on-chip and external or off-chip peripherals. Here we find timers, bus controllers, serial ports, GPIOs, Inter-Integrated Circuit (I2C), or Serial Peripheral Interfaces (SPI). As explained in section 2.3 we have multiple options for accessing these peripherals. The problem with peripherals might not be evident as, from a programmer's perspective, using these peripherals is an easy task thanks to software libraries. For example, when wanting to communicate with an SPI connected via MMIO, the code is not much different from sending and receiving a message [6]. However, the compiled firmware goes through a complex series of MMIO accesses. Therefore, it is challenging to observe the correct behavior of data flow [6]. On top of this, using the same peripherals on two different CPUs from the same manufacturer vary wildly in their memory layout and implementation, which makes it even harder to find any indication of their layout in memory [6].

Because firmware re-hosting is a multi-stage process, categorizing the difficulties only by hardware is insufficient. Hence, why we additionally propose the idea of Wright et al. [2] of splitting these challenges by the time they occur in the emulation pipeline:

- Pre-Emulation
- Emulation
- Post-Emulation

Pre-Emulation consists of problems that are the requirement for emulation execution. Therefore, overcoming these challenges enables the execution of the first instruction in the emulator. These problems range from understanding how to configure the emulator for re-hosting to obtaining and unpacking the firmware. Once the first instruction got executed, problems now occur in the *Emulation* stage. This phase consists of two sub-stages, namely *Emulation Setup* and *Emulation Execution.* In Setup, challenges that allow further emulator refinement can be found, whereas Execution consists of fundamental challenges to emulation itself. Last is the *Post-Emulation* stage, where the re-hosting instance gets validated and analyzed depending on the use case of the digital twin. This thesis will mostly discuss the Pre-Emulation and Emulation stages, as these two phases include the most important considerations when re-hosting firmware.

3.2.1 Pre-Emulation

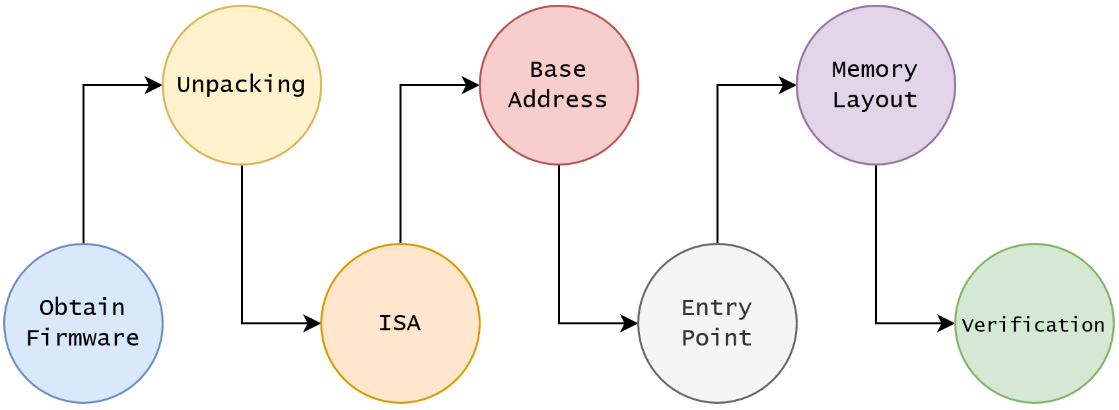

Figure 3.1: Categorization and flow of the steps required during Pre-Emulation. Based on Wright et al. [2]

In Figure 3.1, the flow of the Pre-Emulation stage can be observed. Each step indicates problems that need to be addressed before going on to the next stage. At the start of the re-hosting process, the analyst will usually have a system in mind he wants to emulate or re-host. In some cases, the firmware must be obtained (e.g., when working on a bug bounty or commercial closed-source hardware). Even if the firmware is available, key information required for emulation must be identified. Therefore, all these steps before configuring the emulator are what Wright et al. [2] defines as the Pre-Emulation stage. The required information to begin emulation varies wildly by the emulation technique. However, it includes obtaining the firmware, unpacking it, determining the memory layout and the instruction set architecture (ISA), identifying the processor, analyzing the binary, and an initial firmware analysis. [2].

Obtain Firmware. Before analyzing or re-host firmware, it is necessary to obtain the image. The easiest way would be to download it from the vendor's website, third-party archives, or FTP servers. This can be done manually or automatically via web crawling. If a physical device is present, the firmware can also be extracted via exposed UART-, debug- (e.g., JTAG), or USB- ports from the flash memory. Even though Vasile et al. [19] showed a high percentage of exposed interfaces, locked-down ports are still the norm. When the ports appear to be secured, removing the memories from the board and connecting it to a second system to dump the firmware could be a viable option. However, removing these parts from the circuit board greatly risks damaging or even destroying the hardware [2]. Regarding the surveyed frameworks, Firmadyne and FirmAE constructed their data set by downloading firmware from vendor sites, whereas Pretender obtained them from third-party sites. μEmu used the same data as P^2im [15] and Jetset expanded the set of P^2im [15] with new targets.

Unpacking. Depending on how the firmware got obtained, it may be necessary to extract or unpack the given file. Often signature matching tools such as binwalk [20] or Firmware-Mod-Kit [21] are used. As these tools only check the file headers for pre-defined signatures, encrypted or customized images can not be unpacked [4]. For example, images could contain the firmware for multiple architectures, which are only compiled on the hardware during boot-loading. Alternatively, as is the case for General Purpose Embedded Systems, the downloaded firmware only contains the user-level application [2]. This means that the system kernel must be obtained separately to perform full-system emulation. Frameworks tackling this challenge are Firmadyne [16] and FirmAE [4]: Chen et al. [16] determined through manual experimentation that the built-in recursive extraction mechanism ("Matryoshka") within binwalk was insufficient for their purpose. Specifically, this extraction was vulnerable to path explosion and not guaranteed to terminate, especially in the presence of a false positive signature match. Therefore, they developed a custom extraction tool using the binwalk API as a backend. The tool focuses on minimizing disk space and runtime by terminating when the root filesystem and (optionally) kernel are obtained.

Instruction Set Architecture. After obtaining and extracting the target firmware, it is required to determine which instruction set architecture (ISA) the firmware uses. The emulator uses the ISA to disassemble the firmware into the correct instructions. This includes determining the endianness (e.g., little- or big-endian) and word size. In addition to the ISA family (e.g., ARM, PowerPC, X86, MIPS, ARM64, AVR), the ISA version is sometimes required. Especially when working with ARM architectures, it is important to know if the target runs on ARM with Thumb support or floating point instructions. Most commonly, the ISA can be identified through the datasheet of the target processor. However, static analysis techniques can be used when the target is unknown or the datasheet cannot be obtained. An easy method would be to determine the file format by using the file utility (e.g., ELF, PE2, Mach-O), looking at other signatures in the firmware (e.g., encryption, compression), or extracting strings in the firmware to guess the ISA. A more sophisticated approach would be to use tools like binwalk [20]. Binwalk uses the capstone disassembler and tries to disassemble the firmware for various types of ISAs. If the instruction threshold of the given architecture (default is 500) is reached, then the heuristic gives a good guess for the ISA [2]. Of the compared tools, only Firmadyne and FirmAE try to determine the ISA with the help of binwalk, whereas all other tools expect the ISA to be known beforehand to work correctly.

Base Address. The base address is the address at which the firmware should be loaded. Similar to the instruction set architecture, the base address can sometimes be found in the target datasheet. The base address can be found in the linker scripts if the build tools are available. Because determining the base address is fundamental to many binary analysis techniques, a few papers have focused on automatically finding these addresses. Firmalice [22] exploits jump tables to find the correct loading position. First, the binary blob gets scanned for consecutive values that differ only in their two least significant bytes (LSB) to find jump tables. Next, all memory locations from which the indirect jumps are read get identified. Then, the binary gets relocated such that the maximum number of these accesses is associated with a jump table [22].
Firmadyne and FirmAE try to mitigate this problem by implementing a custom kernel. The kernel is passed to qemu, which bypasses the need to determine the base address. However, this reduces emulation fidelity and loses the original kernel. All other tools require the base address to be known.

Entry Point. After the base address has been found, specifying the entry point is the next step. For some binary formats, like ELF, the entry point information can be directly encoded into the file. When working with binary files without metadata or symbols, the root node of any weakly connected component in the call graph could be treated as an entry point. However, for this, the ISA needs to be already known. Because this call graph analysis often results in multiple entry points, each potential candidate must be verified by hand. [2]
When it comes to the surveyed works, Firmadyne and FirmAE both circumvent this problem by using a custom kernel. In the case of Jetset, μEmu, and Pretender, the entry point needs to be specified. In addition, Jetset requires a special "goal" address, which is used as a stopping condition for the emulation. [11]

Memory Layout. Determining the Memory Layout is of great importance. RAM, Flash, and MMIO might not be available without the layout. If the system is known, the datasheet can be consulted. Sometimes particular files like CMSIS-SVD files for ARM can be used to reverse engineer the memory layout. If all fails, a trial and error method is required. For this, the practitioner gives the target significantly more memory than the physical system and tries to determine where code and external peripherals in the memory lie by observing memory read and writes. These interactions are then mapped into an MMIO wrapper. For all compared frameworks, the memory layout needs to be specified. However, FirmAE and Firmadyne automatically perform the routine described above for mtd devices with the help of `nandsim`.

Verification. While this stage is not necessary, trying to verify the obtained information before moving on to emulation is certainly a good idea. The analyst can often use disassembly to ensure the correct ISA. In addition, a quick control flow graph can be generated to give weak affirmation that the base address and entry point address are correct. Because disassembly is an iterative process, the practitioner can modify the input between iterations to tweak the ISA, processor, base address, memory layout, and entry point. The practitioner may also analyze multiple firmware for the same system and aggregate the results to solidify the variables. [2]

3.2.2 Emulation

Ideally, after gathering all required information in Pre-Emulation, the base emulator already supports the target system. If not, a new specification has to be created. In the case of QEMU [10], documentation on how to add additional support is available. In order to reach a high emulation fidelity, additional challenges must be overcome. These challenges get split into two sub-categories - setup and execution. Emulation setup problems usually occur when the emulator is stopped or paused, whereas emulation execution challenges appear while the emulator is running. Other than in pre-emulation, the challenges faced in the emulation phase are generally not linear but rather occur in a different order depending on the re-hosted firmware. [2]

Setup

Here, the practitioner needs to handle the configuration, external interaction (peripherals), and memory. This stage is often iterative, as with each emulation execution, more knowledge about the firmware operation and its dependencies to peripherals and memory are gained. The knowledge can then be used to refine the configuration until the researcher reaches the emulation fidelity of his choice. The problems discussed in this section are what we believe are the biggest challenges in firmware re-hosting and should therefore be handled with utter importance.

Peripherals. Because of the sheer amount of different peripherals and vendors, it is doubtful that the base emulator implements the peripheral used by the firmware. In section 3.2.1, we already talked about determining the memory map such that we learn where the peripherals are located in the memory. However, the memory map only tells us where the peripherals lay in memory, not how or when they get accessed. Dynamic analysis can be used to gain some of this information depending on the target type. If the target system is a GPES running on a Linux kernel, debugging utilities such as `strace` can be used to understand peripheral interaction. This interaction can then be built into a peripheral model such that the target can interact with it. Depending on the fidelity of the peripheral model, i.e., the extent to which the peripherals are modeled, the amount of code executed may increase, resulting in even more peripheral accesses that can be observed and modeled.
In addition to modeling, peripheral interactions can be handled by forwarding them to the original physical target with the help of a hardware-in-the-loop approach or by patching the firmware and bypassing the interaction. Most of the time, performing a HITL technique will result in the highest emulation fidelity. However, the re-hosted solution will not be able to be parallelized as it is dependent on the physically connected device. Furthermore, HITL has significant problems synchronizing the states between

the emulator and the hardware. A simple example of such an issue would be if a timer generates an interrupt, resulting in the hardware being stuck in the timer's ISR. At the same time, the emulator does not have a timer and consequently keeps executing instructions and not processing any interrupts. Equally important are the current efforts of introducing machine learning to overcome this challenge. While the results of current machine learning approaches, such as Pretender [6], only show proof of concept on simple peripherals like UART ports, the idea of a high-fidelity automatic solution is very appealing. However, it is still unclear if this approach will work for arbitrary peripherals in the future [2]. When using a fuzzer instead, device-specific information is not required but may not achieve a high enough emulation fidelity for the researcher's use case. Still, there are papers published which use a fuzzer in combination with symbolic- and concolic-execution to find vulnerabilities and bugs in re-hosted firmware.
Of the surveyed tools, Jetset and μEmu use symbolic execution to build appropriate responses for hardware interactions. Pretender uses a hybrid approach of HIDL and machine learning to depict MMIO peripherals. Firmadyne and FirmAE assume that the peripherals are part of their core kernel, and the emulator crashes otherwise.

Memory. With the help of emulators, it is possible to taint memory access which allows notifying the user about data access or execution. Most of the time, the practitioner can also specify the memory regions and types of interactions such as RAM, Flash, and MMIO. If the used base emulator does not have memory configuration support, providing an abstraction in software is a feasible option. In addition, forwarding memory access to the actual hardware, such as in projects like PROSPECT [23], can also work. [2]
In Firmadyne and FirmAE, memory is managed by their kernel, and only a portion (mtd layout) of memory can be configured by hand. Because Pretender requires the basic memory layout and MMIO regions to be known, configuring and observing memory access is possible by design [6]. The same argument holds for Jetset [11]. For μEmu, it is essential to specify the memory via a configuration file [12].

Hardware. Setting up the Hardware is most of the time only relevant when using a hardware-in-the-loop (HITL) for firmware re-hosting. This includes initializing the states of the emulator and the physical device and specifying how and when to forward interactions to the hardware. Sometimes this may also include the usage of delays or other specialized hardware to synchronize states or solve other hardware-related problems. [2]
As all surveyed frameworks do not use HITL, these problems are irrelevant. However, in its initialization phase, Pretender uses hardware forwarding to learn about the target's memory layout, but the paper did not state any timing or hardware-related problems [6].

Missing Code. When obtaining the firmware, sometimes the firmware can be incomplete. Reasons for that could be that the sample is a partial firmware update, unsuccessful unpacking, wrong specified entry point, or even patched out functionality

by the bootloader. In some cases, it may be possible to patch out the missing references to the code and still achieve sufficient emulation fidelity. However, often missing code can make re-hosting unfeasible. [2]
For all examined tools, firmware with missing code requires a manual solution.

Function Identification. When working with an emulation fidelity at *Function* level, identifying the functions is necessary. According to Wright et al. [2], this problem is not an easy task, and a plethora of research is still actively trying to overcome it. Still, some techniques could help determine certain functions, like using IDA FLIRT signatures or Hardware Abstraction Library (HAL) calls. [2]
Of the surveyed works, all tools do not need to overcome such problems as they all operate on a different emulation fidelity level than *Function*.

Execution

As already mentioned, Execution deals with fundamental problems related to emulation itself. Because of that, we will discuss the following points not concerning the examined frameworks but in conjunction with the base emulator they use. For example, Firmadyne, FirmAE, μEmu, Pretender, and Jetset use QEMU as base emulators. Therefore, this section will primarily be an in-depth look at QEMU [10].

Register allocation. Every architecture uses different registers and conventions. Emulators need to represent these registers somewhere in their process. On most base emulators, the target registers get mapped to arbitrary memory, and only a few temporary variables get stored in host registers. QEMU, on the other side, uses a fixed register allocation. This means that the target CPU registers get mapped on a fixed memory address or host register. The advantage of this method is simplicity and portability. [10]

Condition code optimizations. In order to achieve good performance, QEMU uses a special technique called "lazy condition code evaluation" to emulate CPU condition code. For example, the x86 architecture uses an eflags register to store the outcome of a condition. Instead of computing the condition codes after each instruction, QEMU stores one operand, the result, and the type of operation, which allows the recovery of all corresponding condition codes if the next instruction needs them. [10]

Direct block chaining. As mentioned in Section 2.5, QEMU translates the target code to host code up to the next jump or instruction, modifying the static CPU state in a way that cannot be deduced at translation time. These basic blocks are called Translated Block (TB). Because of that, determining where the next TB lies is essential. QEMU uses a simulated Program Counter (PC) and other information of the static CPU state to find the following TB using a hash table. If the next TB has already been translated, a jump is issued. Otherwise, a new translation is started. For common cases like an indirect jump, QEMU can patch the TB directly so that the block chaining does not produce overhead. [10]

Memory Management. QEMU supports a software MMU which allows the virtual to physical address translation to be done at every memory access. To optimize the performance, a translation cache is used. If the emulated target does not access an area reserved by the host system, QEMU can also be configured to use the mmap system call. However, this mode seems to be deprecated as it needs a patched target OS and the security risk of the target being able to access the host QEMU address space. In order to avoid discarding the TB each time the MMU mapping change, the translated block get indexed by their physical address. However, in terms of block chaining, the connection between TBs still needs to be reset as the physical address of the jump targets may change. [10]

Self-modifying code and translated code invalidation. Normally, in the case of self-modifying code, the CPU issues a special code invalidation instruction that notifies the cache that the code has been modified. However, on some architectures like x86, no cache invalidation instruction is specified. QEMU handles these cases by write-protecting the host pages corresponding to the TBs. If write access is made to these pages, QEMU invalidates all the translated code in a page and enables writing to it, allowing for the code to be rewritten. When using the software MMU mode, invalidating is made more efficient by using a bitmap to only change the relevant section of the page. This avoids invalidating the complete TB when only data is modified on the page. [10]

Hardware interrupts. In QEMU, hardware interrupts are not checked after each instruction. Instead, the user calls a specific function to notify that an interrupt is pending. The function call resets the current TB chain, which ensures that the execution will return to the main loop of the CPU emulator. Afterward, the main loop executes the corresponding ISR. [10]

Timing Constraints. QEMU may not be the best choice if the practitioner's research question includes timing-related topics. While it will be faster, the execution time is non-deterministic, which may make it unsuitable as a base emulator when timing guarantees are required. In many cases, timing is closely related to the interrupt handling of timers or watchdogs. Interrupts from these sources often need some special triggers or timing implementation in the base emulator. [2]

Multi-Threading. When multi-threading is enabled on a system, the application that uses multiple threads may use semaphores for inter-process communication, but this requires that the emulator allows multiple threads to run simultaneously. While QEMU can emulate such systems, because of its design, the threads will only be able to interact with each other at the end of each translated block. QEMU can overcome this problem by setting break-points at every instruction, which tricks QEMU into thinking that each instruction is a basic block. However, this will drastically reduce the performance of the tool. [2][10]

3.3 Comparison

To compare the state-of-the-art tools even further, we want to introduce the four required properties for an ideal analysis re-hosting system developed by Gustafson et al. [6]:

- **Virtual.** The system should not require physical hardware. As explained in section 2.7 hardware-in-the-loop approaches inherently limit the scale of firmware re-hosting by making the solution depend on the presence of hardware.
- **Interactive.** Because analysis is often the goal of re-hosting, the possibility for fuzzing or symbolic execution must be present. For that to happen, the system's emulated hardware should be responsive to new input.
- **Abstraction-less.** In an Ideal system, software abstractions must not be required. For instance, as explained in section 3.2 and 3.2.1, firmware is often obtained without any format. Therefore, constraints on the firmware format greatly limit the scope of emulatable targets.
- **Automation.** For each device, an ideal system should not require significant manual effort to emulate.

All tools except Pretender [6] fulfill the property virtual. This conflicts with the evaluation of Gustafson et al. [6]. However, this property is obviously violated as Pretender [6] requires the presence of hardware during initialization.
When it comes to interactivity Firmadyne [16], Pretender [6], FirmAE [4] and µEmu [12], fully implement input responsiveness. Because Jetset [11] only emulates to a given *goal address*, the correct response to new input can not be promised.
In terms of abstraction, the strict requirement of a running Linux OS System limits the scope of targeting systems enormously. Therefore Firmadyne [16] and FirmAE [4], which is an extension of the first, are not considered abstraction-less solutions.
Lastly, let us take a look at the possibility of automation. Jetset [11] has tight constraints regarding its emulation requirements. Because the analyst specifies the memory layout, entry point, and *goal address* need to be specified, Jetset is clearly not built for automation. Firmadyne [16], µEmu [12], and FirmAE [4], on the other hand, only use the available information firmware images have built-in, and therefore can easily automate the re-hosting of firmware on a very large scale. Pretender [6] can also be used for automatic emulation, given that the device the image should run on got already traced. A summary of this comparison can be found in Table 3.1.

Firmadyne [16], FirmAE [4] as well as Pretender [6] try to fully re-host a given firmware. Jetset [11] is a classic partial re-hosting tool, trying only to implement sufficient peripherals to emulate the target to a given instruction successfully. In the case of µEmu [12], things are not as clear. µEmu [12] builds models for peripheral accesses that directly influence branch decisions. These models do not get emulated but are only approximated by an SMT solved value, similar to Jetset [11]. Therefore we see µEmu [12] more of a partial re-hosting solution.

Each tool has its own focus, benefits, and disadvantages, so comparing them only by these properties is insufficient. For example, even though Firmadyne [16] and FirmAE [4] might depend on the abstraction Linux OS systems bring, no other tool can achieve such a high emulation success rate for these devices. Similar, when only a particular function is of interest, using Jetset [11] might be the tool of choice. Another notable fact is that not all tools currently have the support for all CPU architectures or peripherals types. For example, pretender [6] is a proof of concept built to showcase the re-hosting ability on ARM systems with MMIO peripherals. However, by providing a basic instruction set emulator, creating the short interrupt recording stub, and providing the needed physical memory access to the device to enable recording, Pretender [6] can be easily extended for other architectures. When it comes to peripherals, Jetset [11] does currently not support devices that perform direct memory access (DMA) to normal RAM [11]. According to Johnson et al. [11], this has to do with how DMA's are handled. Since the CPU does not assist in DMA, this behavior is not observable by firmware, resulting in additional manual assistance needed. As seen, we are currently far from a generic solution for firmware re-hosting. Therefore, the practitioner should always select his choice based on his intentions.

Tool	Virtual	Interactive	Abstraction-less	Automatic
Firmadyne	true	true	false	true
PRETENDER	-	true	true	true
FirmAE	true	true	false	true
Jetset	true	false	true	false
µEmu	true	true	true	true

Table 3.1: Excerpt of tools tackling the re-hosting problem

CHAPTER 4

FirmAE Verification

4.1 Problem Statement

During another research project concerning the emulation and re-hosting of router firmware and their corresponding web service, we required a framework that could take a Linux root file system and automatically host its content. As we were working with Linux-based operating systems, we were looking for a tool that could use the abstraction the Linux kernel offers to overcome some of the challenges emulations bring. In addition, our research included the usage of parallelization and, therefore, could not use hardware-in-the-loop approaches for re-hosting. Which is why we choose Firmadyne [16], and FirmAE [4] as our frameworks of choice. FirmAE is an extension of Firmadyne and claims that it increases the emulation success rate of Firmadyne significantly by implementing five arbitration techniques. To be specific, Kim et al. [4] aver that their implementation raises the success rate from 16.28% to 79.36%. However, while evaluating FirmAE for our research, our samples returned a relatively low emulation success rate. Kim et al. [4] states that their heuristics were developed to handle failure cases empirically and may not apply to new devices and configurations. Still, the sample set used in the research looks deliberately chosen to reflect their results. For example, firmware images from the manufacturer NETGEAR have an emulation success rate of 93.80% while being part of 24.3% of all evaluated images. Therefore, we propose the creation of a new sample set to try reproducing the stated success rate. In order to prove our hypothesis, we first evaluate FirmAE over their published *AnalysisSet*, *LatestSet* and *CamSet*. Afterward, we construct our own sample set based on the sophisticated research results from Kumar et al. [5]. Next, we use them as input on Firmadyne and FirmAE and compare both results in order to back or disprove their claimed ≈4.8x better emulation success rate. In addition, we verify the impact each arbitration technique has by disabling them manually for each set and comparing it with the published results from Kim et al.[4].

4.2 Experimental Setup

4.2.1 Environment

All experiments were conducted on a Hyper-V server with an AMD Ryzen 5 5600X 6-Core 3.70 GHz CPU, 32 GB DDR4 RAM, and 1 TB SSD. We installed Ubuntu `20.04` with PostgreSQL `12.11` and Docker `20.10.12`. For our FirmAE instance we used commit `65e528d76e83181e9f91c51bc59008d1fd9b085d` (29 Oct 2021). In order to compare the results to Firmadyne, we executed the samples in FirmAE without any arbitrations. This behaviour was configured by setting `FIRMAE_BOOT`, `FIRMAE_NETWORK`, `FIRMAE_NVRAM`, `FIRMAE_KERNEL`, and `FIRMAE_ETC` to `false` in the `firmae.config` file.

4.2.2 FirmAE Dataset

In order to test how well our setup performs, we used the published dataset from Kim et al. [24] and tried to reproduce their findings. The set comprises 1124 firmware samples from the top eight wireless home router and IP camera vendors. From those samples, 1079 are wireless router images, and 45 are IP camera images. The images are divided into three datasets: `AnalysisSet`, `LatestSet`, and `CamSet`. `AnalysisSet` consists of 526 outdated images from D-Link, TP-Link, and NETGEAR. `LatestSet` and `CamSet` were collected on December 2018. `CamSet` consists of 45 samples from D-Link, TP-Link, and TRENDnet, whereas `LatestSet` has 553 images of all eight vendors. According to Kim et al.[4] `AnalysisSet` may include multiple firmware versions per device, whereas the other datasets have only one image per device. In addition, there is no intersection among the datasets. To compare our results, we used the published online spreadsheet [25] of Kim et al. [4], which includes the results of all evaluated firmware samples of their experiment.

4.2.3 Hypothesis Dataset

The dataset used to prove our hypothesis is based on the excellent work of Kumar et al. [5]. Their research assessed user-initiated network scans of 83 million devices in 16 million households in order to provide a large-scale analysis of IoT devices in real-world homes. One of their results is a device landscape table that shows the five most popular vendors per device type across eleven regions [5]. We exported the data manually and implemented a python script that aggregates their results in order to return the highest distributed vendors grouped by device type (e.g., routers, automation, and surveillance devices) which is summarized in Table 4.1. The program calculates the sum of all vendor scores per region. Due to the different devices spread across the distinct regions, we additionally weighted the vendor score by multiplying it with its regional distribution (See Listing 4.1).

```python
    vendor_data_for_device = self.vendor_distribution_per_type_and_region[device_type]
    for region in vendor_data_for_device:
        region_multiplier = region_device_amount_percent[region] / 100
        for vendor, percent in vendor_data_for_device[region]:
            if vendor not in scores:
                scores[vendor] = 0
            score = percent * region_multiplier if weighted else percent
            scores[vendor] += percent * region_multiplier

    return sorted(scores.items(), key=lambda score: score[1], reverse=True)[:limit]
```

Listing 4.1: Excerpt of aggregation script

In order to gain the accumulated score of all types, we iterated over all top vendors per type and calculated the mean value of the summarized values (See Listing 4.2).

```python
    for device_type in device_types:
        for vendor, score in self.get_top_vendors_for_type(device_type, weighted, limit):
            if vendor not in scores:
                scores[vendor] = 0
            scores[vendor] += score

    for vendor in scores:
        scores[vendor] /= len(device_types)
```

Listing 4.2: Excerpt of top vendors overall

This data was then used to gather appropriate firmware samples. The first set we built, called `RandomSet`, consists of 105 randomly chosen images from D-Link, Huawei, TP-Link, AVR, Dahua, Hikvision, Mitrastar, Sagemcom, and Technicolor. As these images represent no particular class of embedded system and support for them is not guaranteed, we constructed a second set named `LinuxSet`, which only contains firmware samples of Linux-based systems running on either ARM or MIPS. `LinuxSet` consists of 35 images from Belkin, Cisco, D-Link, Huawei, Intelbras, TP-Link, and Wdigital. The firmware samples have been downloaded from either the manufacturer's site and third-party suppliers or through extensively traversing the web. As the availability of these images varies between vendors, the selection is not evenly distributed.

4.3 Evaluation

4.3.1 FirmAE Dataset

After extracting the dataset, we used the docker implementation of FirmAE to test the emulation success rate of the images. We used docker because it allows running FirmAE in parallel, allowing greater execution speed. Because our setup only had 1 TB available, we needed to split the dataset into three parts. The results of each part

Surveillance		**Routers**		**Automation**		**Overall**	
Hikvision	20.6243	TP-Link	16.6602	Philips	35.0981	Philips	11.6994
Dahua	15.6551	Huawei	7.8171	SMA	7.0101	Hikvision	6.87477
Free	6.678	Sagemcom	3.914	Belkin	5.1691	TP-Link	5.5534
Cisco	5.5442	Arris	3.4836	Nest	4.9062	Dahua	5.21837
Intelbras	1.7712	Free	2.898	Alertme.com	3.186	Free	3.192
D-Link	1.6469	ZTE	2.797	eQ-3	1.8212	Huawei	2.6057
ICP	1.4617	D-Link	2.6815	Phillips	1.5984	SMA	2.3367
Suga	0.6438	Technicolor	1.6664	Inspur	1.4921	Cisco	2.14777
Flir	0.5883	AVM	1.026	Enphase	1.3957	Belkin	1.7223
Topwell	0.3139	Mitrastar	0.9936	Hager	1.0898	Nest	1.6354
						D-Link	1.4428
						Sagemcom	1.30467
						Arris	1.1612
						Alertme.com	1.062
						ZTE	0.99803
						eQ-3	0.607067
						Intelbras	0.5904
						Technicolor	0.555467
						Phillips	0.5328
						Inspur	0.497367

Table 4.1: Vendors mean score grouped by device type

were merged to receive a single file. Emulated firmware images get saved under the `scratch/<id>` folder and the results of the network and web access respectively under `scratch/<id>/ping` and `scratch/<id>/web`. In addition to these two emulation fidelity parameters, additional information such as `name`, `ip` and `architecture` are also available. We used these files to create a CSV file of all images and their emulation results. To our luck, FirmAE [4] already implemented a script to fetch the required information called `util/collect_results.py`. Next, we downloaded all sheets from their published spreadsheet [25] and combined them. However, as we did not want to lose the information about their manufacturer, we introduced a new field with the help of `csvstack`, called vendor, which is used to identify the sample and its associated sheet.

```
[...]
$ csvstack -n vendor -g asus_latest asus_latest.csv
$ csvstack -n vendor -g belkin_latest  belkin_latest.csv
$ csvstack -n vendor -g linksys_latest  linksys_latest.csv
$ csvstack -n vendor -g netgear_latest  netgear_latest.csv
$ csvstack -n vendor -g tplink_latest  tplink_latest.csv
$ csvstack -n vendor -g trendnet_latest  trendnet_latest.csv
$ csvstack -n vendor -g zyxel_latest  zyxel_latest.csv
$ csvstack -n vendor -g dlink_ipcamera  dlink_ipcamera.csv
$ csvstack -n vendor -g tplink_ipcamera  tplink_ipcamera.csv
$ csvstack -n vendor -g trendnet_ipcamera  trendnet_ipcamera.csv
$ csvstack -n vendor -g dlink_latest  dlink_latest.csv
```

Listing 4.3: csvstack adding field

After modifying our data layout, we implemented a small python script that compares the data to the published FirmAE dataset. The program iterates over the FirmAE set and tries to find a corresponding entry in the result set. It compares the `web` and `ping` results of each point. If the columns match, `is_same` gets set to true, and the next row gets processed.

```
[...]
with open('comparison_result.csv', 'w') as csvfile:
    writer = DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for row in dataset_reader:
        vendor = row['vendor']
        name = row['Name']
        data_ping = row['ping'].lower()
        data_web = row['web'].lower()
        result_ping = 'false'
        result_web = 'false'
        exists = False
        is_same = False

        #cases where the dataset included a wrong _ at the end
        #DIR825B1_FW201SS_KR_
        name = name.strip("_")

        with open(args.results, 'r') as resultobj:
            result_reader = DictReader(resultobj)
            for r_row in result_reader:
                if r_row['Name'] == name:
                    exists = True
                    result_ping = r_row['ping'].lower()
                    result_web = r_row['web'].lower()

                    if result_web == 'none':
                        result_web = 'false'
                    if result_ping == 'none':
                        result_ping = 'false'

                    if result_ping == data_ping and (result_web == data_web):
                        is_same = True
                    break

        writer.writerow({'vendor': vendor, 'name':name, 'is_same':is_same, '
    exists_in_result':exists, 'data_ping':data_ping, 'result_ping':
    result_ping, 'data_web':data_web, 'result_web':result_web})

dataobj.close()
```

Listing 4.4: data comparison script

Afterward, we imported the returned file into a spreadsheet in order to calculate the emulation success rate per manufacturer. The result of this evaluation can be found in Table 4.2. The experiment shows that our setup works equally well as the one used in the paper. We believe that the better emulation fidelity is the cause of the newer utilized FirmAE commit, which introduced various bug fixes over three years. The only set which performed worse is `Linksys` from `LatestSet`. In this subset an image named `FW_E2500_2.0.00.001_US_20140417` reported no web access. This contracts the information found in the published emulation report of FirmAE [25]. As this experiment was only done to prove our setup's functionality, we will not continue with failure analysis. We will therefore carry on with the evaluation of our hypothesis. However, we assume that the failed case of `FW_E2500_2.0.00.001_US_20140417` has something to do with the configuration in `docker-helper.py`. Each emulation test runs until it succeeds, fails, or runs into the configured timeout (default 2400s). During our evaluation, we observed execution times close to the timeout (sometimes around 30 minutes), which could mean that the `Linksys` image timed out.

			FirmAE		**Evaluation**	
Dataset	**Vendor**	**Images**	**Net**	**Web**	**Net**	**Web**
AnalysisSet	D-Link	179	177	167 (93.30%)	177	168 (93.85%)
	TP-Link	73	73	59 (80.82%)	73	62 (84.93%)
	NETGEAR	274	259	257 (93.80%)	259	257 (93.80%)
Sub Total		526	509	483 (91.83%)	509	487 (92.59%)
LatestSet	D-Link	58	54	48 (82.76%)	54	49 (84.48%)
	TP-Link	69	69	54 (78.26%)	68	56 (81.16%)
	NETGEAR	101	92	79 (78.22%)	95	83 (82.18%)
	TRENDnet	106	91	63 (59.43%)	90	66 (62.26%)
	ASUS	107	63	62 (57.94%)	64	62 (57.94%)
	Belkin	37	30	22 (59.46%)	31	22 (59.46%)
	Linksys	55	48	44 (80.00%)	50	43 (78.18%)
	Zyxel	20	18	10 (50.00%)	18	10 (50.00%)
Sub Total		553	465	382 (69.08%)	470	391 (70.70%)
CamSet	D-Link	26	19	17 (65.38%)	19	17 (65.38%)
	TP-Link	6	6	0 (00.00%)	6	6 (00.00%)
	TRENDnet	13	10	10 (76.92%)	13	10 (76.92%)
Sub Total		45	35	27 (60.00%)	35	27 (60.00%)
Total		**1124**	**1009**	**892 (79.36%)**	**1014**	**905 (80.51%)**

Table 4.2: Emulation rate comparison of the FirmAE dataset

4.3.2 Hypothesis Evaluation

In order to test our hypothesis, we first tried to evaluate `RandomSet`. The first round of analysis proved to show that a large number of samples could not be processed by FirmAE. The reason is that most of the used samples do not include a Linux-based system. Also, only about 30 images of `RandomSet` could be extracted and identified correctly by the framework. Because of this, a reduced data collection of `RandomSet` was constructed, which only consists of samples that Firmadyne and FirmAE support. The new set includes 30 images and represents the 28.5% success rate we observed during our initial analysis. After evaluating `LinuxSet`, we collected and imported the results into a spreadsheet for further processing. The result can be seen at Table 4.3. The table shows that out of 65 samples, Firmadyne could only manage to emulate and provide access to the web service for two of them. FirmAE, however, managed to emulate 32.3% of all images. Therefore, the results show that FirmAE has a 10.48 times better emulation success rate than Firmadyne, which means that our hypothesis got disproved.

			FirmAE		**Firmadyne**	
Dataset	**Vendor**	**Images**	**Net**	**Web**	**Net**	**Web**
RandomSet (reduced)	AVR	6	1	0 (0.00%)	0	0 (0.00%)
	Dahua	3	0	0 (0.00%)	0	0 (0.00%)
	D-Link	5	5	4 (13.3%)	0	0 (0.00%)
	Huawei	8	0	0 (0.00%)	0	0 (0.00%)
	Sagemcom	3	3	0 (0.00%)	0	0 (0.00%)
	TP-Link	5	5	2 (6.66%)	0	0 (0.00%)
Sub Total		30	14	6 (20.0%)	0	0 (0.00%)
LinuxSet	Belkin	3	2	2 (5.71%)	1	1 (2.85%)
	Cisco	7	4	2 (5.71%)	0	0 (0.00%)
	D-Link	9	7	5 (14.2%)	1	1 (2.85%)
	Huawei	1	0	0 (0.00%)	0	0 (0.00%)
	Intelbras	5	2	2 (5.71%)	0	0 (0.00%)
	TP-Link	6	6	4 (11.4%)	1	0 (0.00%)
	Wdigital	4	0	0 (0.00%)	0	0 (0.00%)
Sub Total		35	21	15 (42.9%)	3	2 (5.71%)
Total		**65**	**35**	**21 (32.3%)**	**3**	**2 (3.08%)**

Table 4.3: Emulation rate comparison of RandomSet and LinuxSet

As we also wanted to see how the different emulation tweaks of FirmAE impact the success rate, we also evaluated the test set with each arbitration disabled. We modified `firmae.config` for each of the five arbitrations. The result is shown in Figure 4.1. In our experiment, network arbitration seems to be the most important, decreasing the emulation success rate by 20% in RandomSet, 28.57% in LinuxSet, and 24,62% overall. Omitting the boot arbitration also seems to greatly lower the emulation rate by 20%. The other arbitration seemed to impact the rate by 10%, while NVRAM only reduced it by about 4% across the board.

Perhaps the most interesting find is the result of disabling kernel arbitration. It kept the emulation rate the same and increased it in the case of D-Link DWR-111 from RandomSet. This has to do with how kernel arbitration and disabling it is working. Kim et al. [4] implemented a kernel module that can be configured to create stubs in devfs, emulate system reboots, create stubs in procfs, and other quality-of-life features. This module is included in the pre-compiled Linux v4.1 and Linux v2.6 MIPS kernels that get downloaded when deploying FirmAE. The exciting part is that these modified kernels are the only kernel files (except for the ARM kernel) downloaded, meaning that disabling FIRMAE_KERNEL in firmae.config does not omit kernel arbitration but rather switches between the v4.1 and v2.6 kernel respectively (See Listing 4.5). This is important as turning off the kernel arbitration is therefore only possible by unloading the kernel module at run-time and not, as communicated on their GitHub page, by changing the configuration of FirmAE. However, when looking at the published results, the emulation success rate decreases when disabling kernel arbitration which would indicate that Kim et al. [4] maybe used a slightly different setup than the one publicized on GitHub.

```
FIRMAE_KERNEL=true
[...]
    case "${1}" in
        armel)
            echo "${BINARY_DIR}/zImage.${1}"
            ;;
        mipseb)
            if (${FIRMAE_KERNEL}); then
              echo "${BINARY_DIR}/vmlinux.${1}.4"
            else
              echo "${BINARY_DIR}/vmlinux.${1}.2"
            fi
            ;;
        mipsel)
            if (${FIRMAE_KERNEL}); then
              echo "${BINARY_DIR}/vmlinux.${1}.4"
            else
              echo "${BINARY_DIR}/vmlinux.${1}.2"
            fi
            ;;
        *)
            echo "Error: Invalid architecture!"
            exit 1
    esac
}
```

Listing 4.5: firmae.config

Another fascinating result can be seen when comparing the bar graph to the one published by Kim et al. [4]. Unlike in our result, NVRAM arbitration appears to decrease emulation success rate the most with an average of 35%. Turning off the boot and network arbitration decreases the emulation rate by 30%. The other arbitrations seem to impact the success rate by about 22.35%, and only 4.88% of images are affected by kernel arbitration. NVRAM and kernel show the most prominent differences, while boot, network, and other arbitration seem to have around the same value. This demonstrates that when re-hosting a randomly obtained firmware sample, using boot and network arbitration techniques most frequently increases the emulation success rate. NRAM arbitration appears not always to be relevant, but when firmware uses the NVRAM interface, it seems to help significantly in making the web service accessible. Therefore, improving boot and network tweaks should be the primary focus when aiming for a broader and more generic framework.

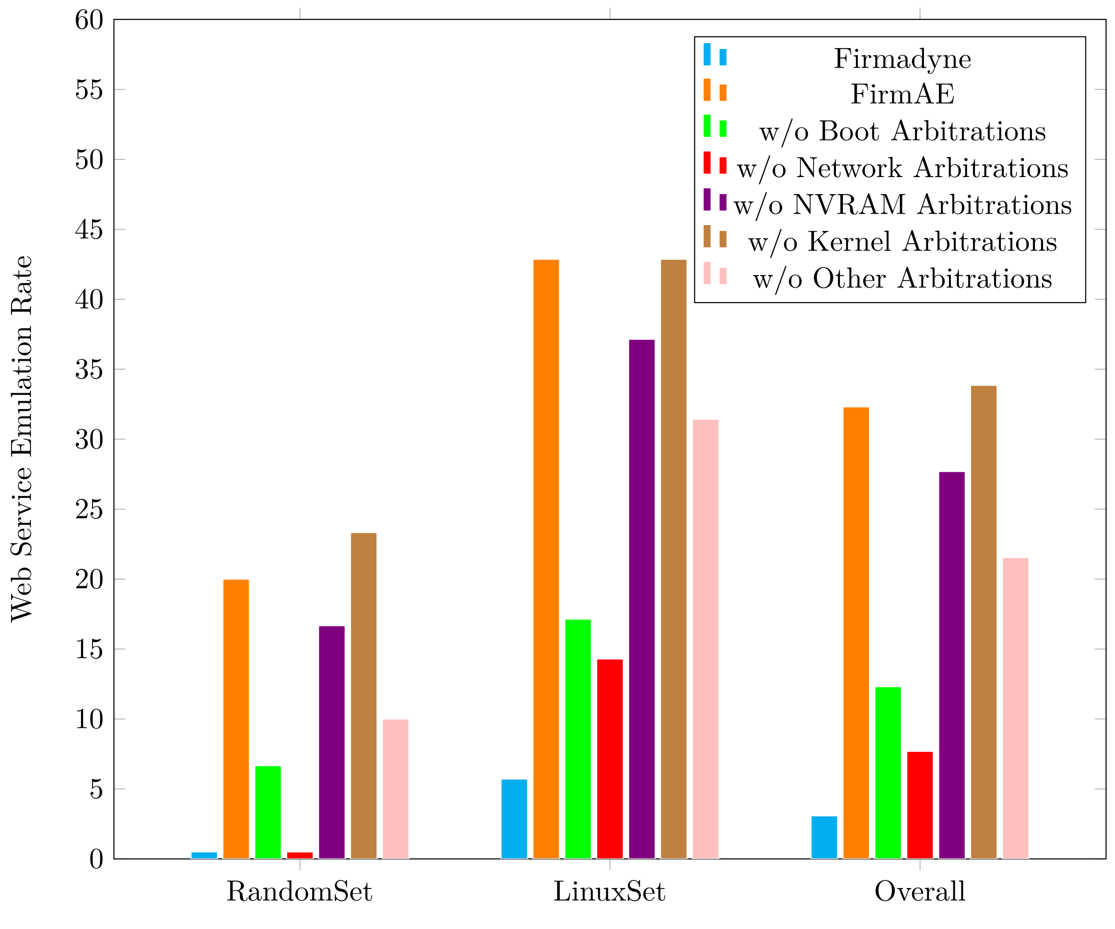

Figure 4.1: web-service emulation rate for each arbitration

CHAPTER 5

Conclusion

Firmware re-hosting has been getting more attention as its use case in developing embedded systems and security analysis is invaluable. In this work we first gave a brief introduction to the terminology that comes with embedded devices and firmware re-hosting. We talked about the different device types and clarified the term firmware. In addition, peripherals and their different access methods in the context of embedded systems got discussed. The difference between re-hosting approaches was another topic we tackled. We explained how binary analysis techniques like fuzzing or symbolic execution get used by firmware re-hosting solutions to infer hardware peripherals. Also, we adopted the idea of Wright et al. and defined the term emulation fidelity, which describes how closely the emulation execution can match that of the physical system. Next, we introduced five firmware re-hosting frameworks and discussed current problems with which this research field has to work. While the biggest challenges appear to be related to peripheral access, more minor difficulties (e.g., unpacking and extracting firmware samples) are still as important and can decide between emulation success and failure. Additionally, we discussed how each of the surveyed tools tries to overcome these complications and introduced properties that ideal analysis re-hosting systems must have to compare them even further. Although we have seen that there are many attempts to solve firmware re-hosting, we are currently far from a generic solution. Therefore, we concluded that the practitioner should always select the tool based on his intentions. The excerpt of tools showed that solutions trying to infer the system's behavior using symbolic execution are currently on the rise and show great potential when considering the information they have available. Still, the fidelity of hardware-in-the-loop approaches is not matched, and many open problems remain before this technique can be generally applicable. Until then, full firmware re-hosting frameworks that use abstractions to specialize on one type of system look the most promising.

FirmAE, the successor of firmadyne, is one of those tools. Kim et al. [4] stated that their developed techniques increase the emulation success rate of firmadyne by about 4.8 times. We used their published dataset to verify their results and proved that our experimental setup works equally well as theirs. Afterward, we used the research results of Kuma et al. [5] to construct our own dataset and demonstrated that FirmAE has a 10.48 times better emulation rate than firmadyne. Disabling each arbitration showed that the categories boot and network seem to have the most impact, while NVRAM only reduces the success rate by about 4%. This contracts with the information found in the published paper. While the categories boot, network, and other are roughly around the same values, NVRAM and kernel show significant differences. Therefore we propose that for a broader and more generic framework improving boot and network arbitration should be the main focus. We also found that disabling kernel arbitration increases the emulation success rate for one sample, and turning it off only switches between a modified v4.1 and v2.6 Linux kernel. Meaning that deactivating kernel arbitration can only be done by unloading the kernel module at runtime and not as stated by setting the corresponding option in `firmae.config`. Due to this reason, further studies should look at the available data to determine whether the images may depend on the newer functions that the v2.6 kernel does not offer and, thus, reduce the emulation rate.

List of Figures

3.1 Categorization and flow of the steps required during Pre-Emulation. Based on Wright et al. [2] 13

4.1 web-service emulation rate for each arbitration 31

List of Tables

3.1 Excerpt of tools tackling the re-hosting problem 21

4.1 Vendors mean score grouped by device type 26
4.2 Emulation rate comparison of the FirmAE dataset 28
4.3 Emulation rate comparison of RandomSet and LinuxSet 29

Bibliography

[1] J. Margolis, T. Oh, S. Jadhav, and Y. Kim, “An in-depth analysis of the mirai botnet,” pp. 6–12, 07 2017.

[2] C. Wright, W. Moeglein, S. Bagchi, M. Kulkarni, and A. Clements, “Challenges in firmware re-hosting, emulation, and analysis,” *ACM Computing Surveys*, vol. 54, pp. 1–36, 01 2021.

[3] A. Vetterl and R. Clayton, “Honware: A virtual honeypot framework for capturing cpe and iot zero days,” in *2019 APWG Symposium on Electronic Crime Research (eCrime)*, pp. 1–13, 2019.

[4] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, “Firmae: Towards large-scale emulation of iot firmware for dynamic analysis,” pp. 733–745, 12 2020.

[5] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, “All things considered: An analysis of IoT devices on home networks,” in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 1169–1185, USENIX Association, Aug. 2019.

[6] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, “Toward the analysis of embedded firmware through automated re-hosting,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, (Chaoyang District, Beijing), pp. 135–150, USENIX Association, Sept. 2019.

[7] M. Muench, *Dynamic binary firmware analysis : challenges solutions*. PhD thesis, 09 2019.

[8] D. Mange, “Teaching firmware as a bridge between hardware and software,” *Education, IEEE Transactions on*, vol. 36, pp. 152 – 157, 03 1993.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, p. 50–58, feb 2002.

[10] F. Bellard, “Qemu, a fast and portable dynamic translator.,” pp. 41–46, 01 2005.

[11] "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[12] "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[13] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, p. 385–394, jul 1976.

[14] G. Vidal, "Concolic execution and test case generation in prolog," pp. 167–181, 09 2014.

[15] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1237–1254, USENIX Association, Aug. 2020.

[16] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," 01 2016.

[17] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," vol. 4963, pp. 337–340, 04 2008.

[18] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, 2017.

[19] S. Vasile, D. Oswald, and T. Chothia, *Breaking All the Things—A Systematic Survey of Firmware Extraction Techniques for IoT Devices: Studies on Socio-Ecological Systems' Vulnerability, Resilience and Governance*, pp. 171–185. 01 2019.

[20] C. Heffner, "binwalk," 2010.

[21] J. C. Craig Heffner, "firmware mod kit," 2011.

[22] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[23] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect peripheral proxying supported embedded code testing," 06 2014.

[24] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae emulation dataset." `https://drive.google.com/file/d/1hdm75NVKBvs-eVH9rKb5xfgryNSnsg_8/view?usp=sharing`, 2020. Accessed: 2022-08-19.

[25] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, “Firmae emulation result.” `https://docs.google.com/spreadsheets/d/1dbKxr_WOZ7UmneOogug1Zykj1erpfk-GzRNni8DjroI/edit?usp=sharing`, 2020. Accessed: 2022-08-19.