THESE DE DOCTORAT DE
SORBONNE UNIVERSITE

Spécialité « Informatique »

Ecole doctorale EDITE de Paris (ED130)

Présentée par
Paul L. R. OLIVIER

Pour obtenir le grade de
DOCTEUR de SORBONNE UNIVERSITE

Sujet de la thèse :

Improving Hardware-in-the-loop
Dynamic Security Testing
For Linux-based Embedded Devices

Soutenue publiquement à Biot, le 22/03/2023

Devant le jury composé de:

Directeur de thèse	Prof. Aurélien FRANCILLON	EURECOM
Rapporteurs	Prof. Guillaume HIET	CentraleSupélec
	Dr. Vincent ROCA	Inria
Examinateurs	Prof. Davide BALZAROTTI	EURECOM
	Dr. Marius MUENCH	Vrije Universiteit Amsterdam
	Prof. Vincent NICOMETTE	INSA Toulouse
	Dr. Fabienne WAIDELICH	SIEMENS A.G.

Abstract

Dynamic analysis techniques have proven their effectiveness in security assessment. Nevertheless, it is necessary to be able to execute the code to be analyzed and this is often a challenge for firmware, most of which are deeply integrated into the hardware architecture of embedded systems. Emulation allows to execute a large part of the code but is quickly limited when it is necessary to interact with specialized components. For this, the partial emulation, or hardware-in-the-loop, approach offers several advantages: transferring access to hardware that is difficult to emulate properly and executing the firmware in turn on both entities. To date, this approach has been considered primarily for monolithic firmware, but less so for devices running advanced operating systems. In this thesis, we explore the challenges of security testing for processes running in an emulated environment where part of their execution must be transmitted to their original physical device.

Throughout this thesis, we first review the various techniques for intercepting system calls and their objectives. We highlight the fact that forwarding is not a very well explored technique in depth but is a promising approach for evaluating the security of embedded applications. We discuss the challenges of different ways of running a process in two different Linux kernels. We implement through a framework these transfers for a Linux process with its system calls and memory accesses between its emulated environment and its original environment on the physical device. To overcome the challenges of using physical devices for these security tests, we also present a new test platform to reproduce hardware-in-the-loop security experiments.

Résumé

Les techniques d'analyse dynamique ont fait la preuve de leur efficacité pour évaluer la sécurité. Il est en revanche nécessaire de pouvoir exécuter le code à analyser et c'est souvent un défi pour les firmware dont la plupart sont profondément intégrés à l'architecture matérielle du système embarqué. L'émulation permet d'exécuter une grande partie du code mais se retrouve rapidement limitée lorsqu'il est nécessaire d'interagir avec des composants spécialisés. Pour cela, l'approche d'émulation partielle, ou hardware-in-the-loop, offre plusieurs avantages: transférer les accès aux matériels qui sont difficiles à émuler correctement et exécuter le firmware à tour de rôle sur les deux entités. Jusqu'à présent, cette approche a été principalement considérée pour les firmware monolithiques, mais moins pour les dispositifs utilisant des systèmes d'exploitation avancés. Dans cette thèse, nous explorons les défis des tests de sécurité pour les processus exécutés dans un environnement émulé où une partie de leur exécution doit être transmise à leur dispositif physique d'origine.

Au travers de cette thèse, nous passons d'abord en revue les différentes techniques d'interception des appels système ainsi que leurs objectifs. Nous soulignons le fait que le transfert est une technique peu explorée en profondeur mais une approche prometteuse pour évaluer la sécurité des applications embarquées. Nous discutons des défis des différentes manières d'exécuter un processus dans deux noyaux Linux différents. Nous implémentons au travers d'un framework ces transferts pour un processus Linux avec ses appels système et accès mémoire entre son environnement émulé et son environnement d'origine sur le dispositif physique. Afin de surmonter les défis liés à l'utilisation de dispositifs physiques pour ces tests de sécurité, nous présentons également une nouvelle plateforme pour les tests de sécurité hardware-in-the-loop.

Acknowledgements

I would like to take this opportunity to share my sincerest thankfulness to those who have supported me throughout my PhD journey.

First and foremost, I want to express my deepest gratitude to my *esteemed* advisor Aurélien, for his invaluable guidance, constructive feedback, and unwavering support. As a young scientist and researcher, his mentorship has been a constant source of inspiration and motivation.

I would also like to thank my colleagues and friends from the S3 group at Eurecom for providing a stimulating and positive research environment. I hope that future generations of students will also have the privilege of experiencing such conditions. Additionally, many thanks to the Eurecom administration for their assistance in navigating the bureaucracy processes that come with pursuing a PhD.

I would also like to express my sincere appreciation to the members of the committee for their valuable comments. In particular, I am grateful to the reviewers, Prof. Guillaume HIET and Dr. Vincent ROCA for their time and insightful feedbacks, which helped me to improve my research work. Furthermore, I am glad to extend my gratitude to Siemens for their fundings towards my research. I am especially grateful fot the opportunity I had to intern with their team in Munich for two months.

Then, I would like to thank my family for their support and encouragement throughout my studies.

Last but not least, I would like to express my heartfelt gratitude to my fiancée, Isabelle, for her patience and for having always been by my side during all these years.

Contents

Contents

Chapter 1

Introduction

In their arms race and space conquest, both the Soviet Union and the United States needed more sophisticated ways to control the movements of their rockets. With the increase in computing power in the 1960s, it became possible to replace existing analog and mechanical approaches with digital computers. This led to the development of the Autonetics D-17 guidance system [2] and the Apollo Guidance Computer (AGC) [18], two of the earliest digital embedded systems designed. Their missions were critical and required high reliability. These systems were responsible for processing measured data and providing accurate information in a specific time frame. Their weight was considered a crucial factor, as additional weight meant the need for larger and more expensive rockets. Many of their characteristics are shared with modern counterparts.

Today, embedded systems can be found in a wide range of sectors, including industry, consumer, military, health, home automation, and agriculture. It is difficult to provide an exact number of deployed embedded systems. But according to some reports [5, 6], their global market in 2022 is estimated to represent 102.82 billion USD and to reach 130.5 billion by 2027. They are at the heart of critical systems that can sometimes directly affect the lives of human beings. Critical incidents such as the one involving the Therac-25 radiotherapy computer, which resulted in patients receiving overdoses of radiation because of a race condition, demonstrate the importance of asserting error-free software [15].

Therefore, their prevalence in our society and everyday life make them a key issue regarding security and privacy. And it is likely to continue in the future. For instance, the automotive sector is undergoing a paradigm shift with the gradual abandonment of combustion engines in favor of electric ones. New vehicles incorporate a lot of recent embedded systems for their advanced driver-assistance systems (ADAS). They help the driver by providing alerts (e.g., drowsiness detection), driving assistance (e.g., anti-

lock braking system (ABS), cruise control, collision avoidance system) and environment monitoring (e.g., traffic sign recognition, vehicular communication systems). Compared to rocket guidance systems, these systems need to accommodate other moving objects such as cars, bicycles and pedestrians. Moreover, the importance of embedded systems in various industries, such as the automotive sector, is highlighted by the ongoing global chip shortage.

As a result, it is also important to accurately assert the risk inherent in these systems. This led to the adoption of *secure software development life cycle* (SDLC). This methodology defines best practices that integrate security measures at every stage of the product development process. Such measures consist in identifying security concerns, building a threat model, keeping it up to date during the product's lifespan, documenting the procedures for addressing security vulnerabilities, and monitoring for known vulnerabilities. Additionally, it covers security testing such as static code review, test automation with continuous integration, product integrity during deployment and penetration testing on the final product.

However, firmware penetration testing presents additional challenges when compared to traditional software. A wide range of technologies have been developed to address the specialized requirements of embedded systems. This diversity has created a heterogeneous ecosystem in terms of solutions, designs, and architectures. Moreover, the rise of Internet-of-Things (IoT) has led to an increase in the number of connected devices that can communicate with each other and exchange data over the Internet. In addition to increasing their attack surface, these online services make the task even more complex for security analysis, as part of the environment is beyond control and may change continuously over time. Furthermore, the firmware is often closely integrated with the hardware making it challenging to effectively test the firmware without its hardware.

Modern software testing techniques are fruitful in finding bugs. In particular, dynamic analysis techniques such as fuzzing and symbolic execution perform very well and are automatable. These were at the core of the automatic defensive systems used to identify the vulnerabilities during the DARPA Cyber Grand Challenge (CGC) [57]. The best competitors employed a combination of fuzzing and symbolic execution to balance their weaknesses [29, 83, 127, 159]. This event sparked dramatic progress in published research in bug-finding techniques for the following years [111]. However, it is impractical to run these methods directly on the physical device for several reasons. Unlike more traditional computers, embedded systems have more limited resources in terms of computing power, memory size, network bandwidth and energy consumption. Additionally, they also provide less introspection into the firmware execution. Symbolic execution requires identifying branch statements and solving computationally intensive equations.

To select the next input to run, fuzzing relies on metrics derived from the feedback of a program execution trace. Furthermore, running the analysis on the device make automation and scaling more arduous. To overcome these challenges, emulation is often used to enhance the observability of system states and improve control over their execution. This process of moving the firmware from its original host to an emulated environment is called *rehosting* [71]. It comes with new challenges regarding the inference of an environment suitable for the proper execution of the firmware. Although efforts have been made to automatically generate these virtualized environments, the need for accuracy leads to including the original device in the simulation in a *hardware-in-the-loop* (HIL) fashion.

Pairing the hardware with the simulation has been a proven technique in various fields for several decades. It dates back to the early days of rocket testing [30, 68, 164] and was later employed in the development of fly-by-wire (FBW)[1] systems in aircraft [70]. The hardware-in-the-loop approach is particularly useful for scenarios involving complex environments that are difficult to model precisely. In the context of firmware rehosting, using hardware-in-the-loop offers several advantages. It allows focusing the rehosting process on the area of interest while still preserving the interactions with its original environment. In practice, it reduces the reverse engineering scope to the understanding of a suitable interface on the device. Furthermore, the hardware-in-the-loop approach leads to feeding more accurate inputs in the emulation. Additionally, the scalability of the analysis can be enhanced through caching interactions [96] or connecting multiple emulation instances to the same device [105].

Hardware-in-the-loop security testing has been considered primarily for monolithic firmware, but less so for devices running advanced operating systems. Because of their versatility and low cost, Linux-based systems are a popular choice among embedded systems. Although the kernel security model is robust, many flaws are discovered every day. This has an impact on billions of devices around the world and on people relying on them. Therefore, it is crucial to develop appropriate security testing techniques for them.

---

[1]It replaces the traditional mechanical flight controls with electronic signals. It allows for more precise and responsive control of the aircraft.

This thesis aims to explore and address the challenges of security testing with hardware-in-the-loop. In particular, the focus is on Linux-based operating systems with processes running in an emulated environment where part of their execution must be transmitted to their original physical device.

In summary, the main contributions of this thesis consist in:

- a state of the art on publications related to embedded device security testing using hardware-in-the-loop and Linux rehosting firmware approaches.

- a novel approach to filter and forward system calls between an emulator and a physical device which address the shortcomings of existing methods.

- a prototype, *Chestburster*, implementing this previous technique.

- an infrastructure, *BEERR*, aiming to ease the access to physical devices used in system security publications and the reproducibility of their artifacts.

The work of this thesis led to an academic publication "*BEERR: Bench of Embedded system Experiments for Reproducible Research*" [132], and a second on the system call forwarding approach which is going to be submitted soon.

Throughout my graduate studies, I had the pleasure to maintain and extend the research project *avatar²* [123], which was the outcome of the previous thesis [122, 192].

This thesis is organized into 7 chapters. After having introduced the context and the general challenges, Chapter 2 provides the necessary background for the rest of this thesis. Next in Chapter 3, we cover the state of the art by surveying hardware-in-the-loop security testing publications and existing approaches for rehosting Linux firmware. Furthermore, we highlight their current limitations and investigate how system call interception mechanisms can improve them. With Chapter 4, we delve into the challenges of forwarding system calls and propose a novel approach. In Chapter 5, we detail the implementation of our new technique in a prototype: *Chestburster*. Afterward, we present a solution to facilitate access to physical devices and promote artifacts reproducibility in Chapter 6. Finally, we discuss future work and conclude the thesis with Chapter 7.

Chapter 2

Background

This chapter discusses background information relevant to the rest of the thesis.

2.1 Embedded systems

Embedded systems are devices designed to perform a specific task within a larger system. They are optimized and tailored to meet specific requirements and minimize development and production costs. As a result, these systems typically have limited resources in terms of computing power, memory size, network bandwidth and energy consumption. Moreover, these devices often interact with the physical world and are subject to real-time computing constraints. This has led to the creation of a wide variety of technologies to meet different design choices. For example, it is illustrated by the large number of different Instruction Set Architectures (ISA) present in embedded devices (e.g., x86, MIPS, ARM, PowerPC, m68k, AVR, MSP430, RISC-V) where in addition each can enable extra features via diverse architecture extensions.

2.1.1 Firmware

The software driving an embedded system is commonly called a *firmware.* Besides this lexical difference, it presents other distinctions related to its intrinsic nature. A firmware is rarely portable across systems because it is often specific to a particular device and hardware platform. Although some embedded systems rely on safe languages such as Ada, and other languages are becoming more popular such as Rust, most are often written in low-level languages (e.g., C or assembly). It is usually designed to run with minimal intervention from the user, whereas traditional software often works with user inputs. Firmware is typically stored in read-only memory (ROM) or

non-volatile memory (flash memory, EEPROM). This makes its update process more difficult to perform than traditional software updates. All of these particularities contribute to the complexity of their security analysis.

In [124], the authors present a classification of firmware based on the type of operating system they used. This classification is relevant to us because it takes into account the number of *abstraction layers* present in the firmware.

Type 0   The *Non-embedded devices* category is used to represent traditional systems used by smartphones, desktops, workstations and servers. Examples include Unix-like operating systems (e.g., Debian, Android), BSD variants (e.g., OpenBSD, macOS) or others (e.g., Windows).

Type I.   *General-Purpose OS-based devices* are type 0 systems tailored for an embedded environment. They only include features needed by the system with lightweight user-mode applications. Their proximity to traditional operating systems simplifies the use of common analysis techniques. Examples include Minix and Linux-based systems paired with BusyBox.

Type II.   *Embedded OS-based devices* aims to address the resource constraint some systems face. They are typically characterized by their small footprint, high performance, and ability to support real-time scheduling requirements. Despite the absence of advanced processor features such as a Memory Management Unit (MMU), there still exists, usually, a logical separation between the kernel and the application code. Examples include real-time OS such as FreeRTOS, ZephyrOS, VxWorks and QNX which frequently runs on modem devices.

Type III.   *Devices without an OS-abstraction* represent monolithic firmware where all components are linked together into one executable which runs directly on the hardware. This firmware does not have precise OS abstraction but instead relies on library calls.

2.1.2   Peripherals

Peripherals play an essential role in the composition of a System-on-Chip (SoC). They provide the majority of the input/output (I/O) for the processors and connect them to the external environment. Examples include timers, hardware accelerators (cryptographic primitives, network packet processing, graphics rendering), communication interfaces, memory controllers and power management units. Peripherals may be located either internally or externally to the SoC. In the latter case, a method of communication between

the peripheral and the processor must be established. Some common mechanisms include:

- **Memory-mapped I/O (MMIO):** This method of communication involves directly mapping hardware registers of the peripheral in the firmware address space. This way, the processor can read and write the peripheral's status, configuration or any other data.

- **Polling:** The processor periodically checks the status of a peripheral to see if the desired condition is satisfied.

- **Interrupt requests (IRQ):** To avoid wasting the processor's time in polling, peripherals can implement interrupts to notify events, such as when a task is completed. When it happens, the processor stops its execution, saves its current context and switches to the interrupt handler corresponding to the raised interrupt.

- **Direct memory access (DMA):** To continue improving the processor usage efficiency, a DMA peripheral accesses memory directly to read or write large amounts of data. On completion, the processor is notified by issuing an interrupt.

2.2 The Linux kernel

2.2.1 Importance of Linux in embedded devices

The Linux kernel has become so popular that it is now considered an essential component in the field of embedded systems. It is written in a highly portable language (C), has been designed to be easily adaptable to a wide range of hardware architectures and provides a large selection of software tools and libraries. Its modular design offers developers the ability to tailor the kernel to the specific needs of an embedded system and its application. In addition, the kernel has a robust security model to protect the integrity of the system such as memory protection and access controls. Finally, it has a large, active and open-source community that ensures continuous support.

2.2.2 The process abstraction

The process abstraction is a fundamental concept that refers to a program in execution. It is used by the operating system to manage concurrent tasks. Each process has unique identifiers such as the user ID (UID) and the group ID (GID). They can communicate with other processes using inter-process communication (IPC) methods, and in particular with their children. The

process table organizes processes in a hierarchical structure including parent and child relationships. Additionally, a process has its own address space which allows isolating processes from each other.

2.2.3   The memory model

The Linux kernel uses virtual memory management to enable processes to use more memory than is physically available on the system. This is accomplished through the use of paging. The kernel divides the process's virtual address space into small chunks, called *pages*, and swaps them in and out of physical memory as needed. Furthermore, the kernel maintains a *page table* for each process containing the mapping of their virtual addresses to the physical addresses. When a process accesses a virtual address, the hardware looks into the page table for the corresponding physical address and retrieves the data at this location. If the page is not present in physical memory, a page fault is raised. The kernel will then answer this error by loading the page in memory. The kernel also supports various memory-related features. Memory permissions, such as read-write-execute (`rwx`) permissions, help to ensure that processes only access memory in an authorized manner. Memory mapping allows mapping files or devices into processes' virtual address space.

However, it is important to note that different computer architectures may implement different memory models and management techniques [26]. The Most common types are radix tree for Intel x86, ARM and RISC-V, inverted page tables for PowerPC, and software-defined management for MIPS. It is the job of the kernel to provide a unified interface for processes, regardless of the underlying memory model.

2.2.4   Filesystem data structures

The filesystem is responsible for organizing and storing files and directories. The Linux kernel uses various data structures to manage the filesystem and enable file operations. A *file descriptor* is an abstraction that represents the connection between a process and a file or a device. *Inodes* are data structures that store information about files (e.g., size, permissions, location on disk). The *open file tables* and the *inodes tables* are used to keep track of the kernel's connections with open files. In contrast, the *file descriptor tables* are used to keep track of the connections between processes and files. In addition, the kernel uses a virtual filesystem (VFS) to abstract the underlying physical storage and provide a uniform interface for accesses.

Pipes allow processes to communicate with each other by passing data through a buffer. They can be used to redirect the output of one process as

the input of another, creating a chain of processes working together.

Special files, such as block devices and character devices, are used to abstract input/output (I/O) operations and provide a uniform interface for access to hardware devices.

2.2.5 Character devices

Character devices create an interface for user applications to interact with kernel and hardware components. The specificity of character devices is the way they handle data: the exchange is done through a continuous stream of characters. Character devices are accessed from user mode through the filesystem with a defined set of system calls. It is up to the character device to implement the supported system calls and their operations on the device.

Beyond the typical file operations, such as `open`, `read`, `write`, `mmap`, `llseek` or `lock`, the character devices support the `ioctl` system call. Its signature is: `int ioctl(int fildes, int request, ... /* arg */)`. The request code specifies the operation to be performed, and the additional arguments provide its inputs. This design choice grants flexibility to the device on the type of operations it can handle. However, it brings issues regarding consistency and portability across the different platforms. As a result, different device drivers may use different codes to represent the same operation, which can lead to confusion and difficulty when working with multiple devices.

2.2.6 Linux asynchronous I/O

The Linux asynchronous I/O mechanism emables concurrent processing of multiple I/O requests in the kernel without blocking the calling thread. It consists of two main primitives: submitting a message into the pending queue and consuming an event notifying the completion of the request. the *Submission* of a message into the pending queue, and the *event* notifying the completion.

The traditional Linux AIO subsystem has five system calls for managing the I/O contexts: `io_setup`, `io_destroy`, `io_submit`, `io_cancel`, `io_getevents` and `io_pgetevents`. It relies on a ring buffer internally to manage I/O request submissions while completion events are stored in an array. However, this implementation has several limitations such as not working with buffered I/O.

To address these limitations, a new interface called `io_uring` was introduced in Linux 5.1. To reduce the number of context switches, it uses two ring buffers in user space: the *submission queue* for I/O requests and the *completion queue* for events completion. The user space application puts

new I/O requests at the tail of the submission queue, and the kernel consumes them from the head. In the opposite way, the kernel puts completion events at the tail of the completion queue while the application consumes them from the head. As a result, managing diverse operations related to the queues requires only three system calls: `io_uring_setup`, `io_uring_enter` and `io_uring_register`.

2.2.7 Process migration

Process *snapshotting* is a technique used to capture the state of a process, including its context and memory, at a particular point in time. This allows the process to be inspected statically, cloned, restarted later or in a different location.

One use for process snapshotting is *cold migration*, where the process is stopped and its state is transferred to another system in order to continue its execution there. Tools like CRIU [48] (Checkpoint/Restore In Userspace) are commonly used to migrate Linux containers. Another technique for migrating virtual machines without downtime is *hot migration*. QEMU uses the `userfaultfd` [16] kernel feature to register a special file descriptor to handle the process's page faults in user mode. This way, when the migrating process tries to access a memory location not yet migrated, a page fault is raised and QEMU fetches the missing pages before restarting it. Instead, *checkpoint-restart* involves periodically saving the state of a process during its execution.

Distributed operating systems make use of these techniques to improve availability, scalability and load-balancing of applications [107]. For instance, MOSIX [33, 34] divides the migrating process in two parts. While the user context composed of the program code, data, stack, memory-maps and registers is allowed to migrate many times, the system context residing in the kernel stays at the initial nodes. All interactions between these contexts are intercepted and forwarded across the network. Additionally, various resource sharing algorithms are used to manage the load-balancing of processes between nodes.

2.3 Dynamic binary analysis

Static analysis is a method for analyzing a program without executing it. Dynamic analysis, on the other hand, involves executing the program and instrumenting its behavior while it is running. Both of these two classes of analysis offer different advantages and drawbacks which depend on the context of the analysis and its objectives [59, 161, 179]. In general, static analysis

can achieve larger coverage and produce sound results by analyzing all possible execution paths of a program. However, with the lack of runtime context, it has to make approximations that are often arbitrary and lead to false positives. In contrast, dynamic analysis is performed in a given environment and for a given input. It is more precise in what it observes (instructions, content of registers and memory), but offers a smaller coverage because one program path is executed at a time. Examples are techniques such as runtime testing, debugging, profiling, and fuzzing. Static analysis examples include code review, data and control flow analysis, and model checking. For some analysis techniques, the distinction between the two can be unclear. For instance, symbolic execution may be considered either static or dynamic analysis depending on how it is implemented [52, 160].

Source code instrumentation leverages high-level semantics (e.g., the variable types) to better reason about the program's behavior [52, 58, 170]. As a result, it is easier to discover vulnerabilities due to the increased information found in the context available. However, source code instrumentation may not be always feasible because it requires access to the source code and the ability to recompile the program. In addition, it does not test exactly all of what is executed by the processor. From its writing to its execution, the program source code goes into multiple transformations (e.g., compilation, linking, loading) where each stage has a chance to introduce new bugs [175]. This is particularly relevant in the current context with the increasing prevalence of supply chain attacks [167, 172]. Therefore, it is important to also test the program in its binary form, even when the source code is available.

Both approaches are therefore complementary. Testing at the source code level is more reasonable during product development because developers have access to the source code and may require feedback. In contrast, binary testing can happen before product release or for external audits where a threat model may not be initially thought of by the developer team.

In light of this, this thesis focuses on dynamic analysis targeting binary-only programs.

2.3.1 Hardware-in-the-loop

Hardware-in-the-loop (HIL) testing consists of the integration of physical components within a simulation [52, 97, 101, 105, 118, 123, 133, 153, 170]. In this way, the physical devices are fed with inputs from the simulation while their outputs are monitored. This technique brings the benefit of only requiring access to the interface with the physical component. This interface is often standardized (JTAG, I2C, MMIO), which removed the burden to know the internal structure of the device: it is considered a black box.

For instance, cars are increasingly filled with embedded systems with

complex and sensitive architectures. The automotive sector has strong incentives from regulations to extensively test their systems. HIL helps to build realistic testbeds to establish the reliability of the system [75, 133].

In the context of system security, HIL gained a lot of interest recently with *rehosting* [71, 188]. A significant advantage of HIL lies in its high fidelity of outputs returned to the simulation. In contrast, the approach presents several drawbacks. The devices need a debugging access to control their state. This interface often requires reverse engineering efforts to identify it, and may be intentionally disabled by the manufacturer. Moreover, the execution overhead introduced by forwarding accesses can significantly impact the analysis feasibility because of speed and latency issues [170]. Despite being limited by the number of physical devices, various optimization techniques such as caching [96] and concurrent execution [105] can be used to improve the scalability of the analysis.

2.3.2 Emulation

Emulation is the process of mimicking the behavior of a system by using another system. The emulator acts as a translation layer between the software to execute and the current hardware, allowing the software to be executed like it was on its original hardware [19, 36].

Emulation is particularly useful for security analysis because the original environment may not be suitable for deep analysis [64, 87]. It enhances system observability and introspection by being able to inspect and modify the sequence of state the software is passing through. However, the sheer variety of hardware makes it difficult to create a versatile emulator that would support all existing devices. This is even more true in the context of embedded system security where peripherals are often custom and proprietary, removing all hope to access any public documentation. For this reason, recent research focuses on techniques that emulate an appropriate environment for the execution of a given firmware. This technique is called *rehosting* [71, 188].

2.3.3 The avatar2 framework

The avatar² framework [123] is the worthy successor of Avatar [193]. It is a tool developed to facilitate the integration and interoperability between various binary analysis tools such as debuggers, emulators, disassemblers, symbolic execution engines and fuzzers. The framework is particularly aimed at analyzing embedded systems and their firmware, as it allows for the combination of physical devices with emulators in a hardware-in-the-loop fashion. This allows the application of traditional software security testing techniques to complex firmware, which would not otherwise be possible. Addi-

tionally, avatar² provides fine-grained control over the program execution. It allows doing live migration of a program between analysis tools and forwarding special accesses, such as memory and I/O, to others analysis tools for hybrid execution. Avatar² has been used in several security research works [47, 86, 89, 109, 155, 165].

2.3.4 Rehosting

Firmware *rehosting* is the process of creating a virtual environment in which a firmware can be run as if it was in its original physical environment. This allows the application of general dynamic analysis techniques for security testing such as debugging, tracing, fuzz testing and symbolic execution.

While there has been significant progress in this area in recent years, many challenges around obtaining the firmware image and its execution remain [71, 188]. The firmware image acquisition process is not consistent across devices. It may require intercepting updates, exploiting vulnerabilities, de-soldering and dumping memory chips, connecting to a debug interface or being confronted with protections such as an encrypted image or hardware memory read-protection that require invasive attacks. To proceed, it is necessary to identify the ISA utilized by the firmware (e.g., ARM, MIPS, PowerPC, AVR, MSP430). Additionally, the emulator should be able to determine and model the peripherals utilized by the firmware.

The chosen approach for rehosting a firmware depends on the individual case, as firmware, peripherals, and environments can vary significantly.

2.3.5 Fuzzing

Fuzz testing, or fuzzing, is a technique used to discover unexpected or erroneous behavior in a system by repeatedly feeding modified inputs. It involves executing a large number of inputs and collecting feedback in order to mutate them.

Early fuzzers [67, 117, 135] used a *blackbox* testing approach to execute a target as often as possible. However, their lack of introspection hindered their ability to thoroughly explore the target code due to conditional statements [80]. This led to the development of novel *whitebox* [42, 81, 82, 166] and *greybox* [156, 168, 194] fuzzers, which instrument the target to collect feedback and produce *better* inputs.

Originally developed to find security bugs in software, fuzzing has evolved into a more general approach that can be used to explore the different possible states of a system [39, 148].

Fuzzing has experienced considerable interest in software testing and vulnerability research because of its efficiency in bug finding [74]. Yet estab-

lishing good methodologies and metrics to compare fuzzing techniques and algorithms remain a challenge for researchers [73]. For this reason, Google proposes the FuzzBench [116] service to evaluate and compare fuzzers against a set of benchmarks.

Embedded systems offer additional challenges for fuzz testing [124, 154, 191]. Their limited resources make it difficult to run efficiently many inputs [40]. Furthermore, the lack of introspection into their internals can hinder the ability to collect data and provide meaningful feedback for selecting interesting inputs to mutate. Additionally, embedded systems often have complex interactions with their environment, making it difficult to correctly identify their input [72, 115, 150, 155, 165]. Testing on real hardware requires a method to properly reset the system between runs [53]. There is also the risk to brick the system and potentially posing a danger to the physical world and humans. Finally, as highlighted by Muench et al. [124], silent memory corruption can further complicate the detection of crashes.

2.3.6   Symbolic & concolic execution

Symbolic execution is a technique used to explore the possible states of a program by executing it *symbolically*. This method involves replacing certain inputs and variables during execution with symbolic expressions. As the program is executed, constraints are placed on these symbolic expressions. Solvers are then used to determine whether all the constraints are satisfiable and if so, to generate an input that can reach those states in the program.

For each conditional statement in the program, the symbolic engine typically forks execution to follow both paths. The number of paths to explore therefore grows exponentially over time and may make the analysis impractical for larger programs. This problem is referred to as *path explosion*. To address this issue, different approaches try to combine symbolic execution with concrete execution. *Concolic execution* [81, 157] uses a concrete input to guide the symbolic execution. This approach was later improved with the help of fuzz testing [166]. Differently, *selective symbolic execution* [46] limits the symbolic execution to a specific part of a program. This is especially useful when the program is composed of elements not relevant to the analysis.

Symbolic execution has been used to analyze firmware. Studies such as [58, 88, 160, 193] have demonstrated its effectiveness in identifying vulnerabilities in firmware. In particular, Inception [52] discovered a vulnerability in a bootloader before being written on the Mask ROM. Moreover, the framework hightlights the challenges to apply symbolic execution to firmware that contains both low-level and high-level code, resulting in different levels of semantic information.

More recently, work has been pushed to improve the execution speed of the symbolic engine. QSYM [190] brings the idea to replace the symbolic interpreter by instrumenting the execution with native machine code. Following approaches [50, 144, 145] include the concolic execution in the binary code with the help of compilers to drastically improve the execution speed.

2.3.7 Multi-variant execution environment

Multi-Variant eXecution (MVX) systems are used to prevent exploits by executing multiple variants of the same program with the same inputs [37, 55, 90, 100, 182, 183]. The goal is to detect any divergence, discrepancy, or difference in the execution of the variants, which would indicate that the program has been exploited. To do this, variants run on the same machine and are synchronized at the system call interface. The MVX monitor, responsible for managing variants, may be implemented as a loadable kernel module (LKM) [55, 182] or run in user mode [37, 90] to balance instrumentation context and execution overhead. The main challenges in multi-variant execution are the methods of synchronization and the strategy used to handle variant discrepancy.

Chapter 3

State of the art

In this chapter, we present a comprehensive overview of the current state-of-the-art techniques in hardware-in-the-loop (HIL) security testing and rehosting. We begin by examining the various publications that include physical devices in their security analysis. We then delve into the challenges of rehosting Linux-based firmware and provide an in-depth examination of the existing approaches and their shortcomings. Starting from the observation that current methods are limited in reproducing the original firmware environment, we also explore the interception mechanisms for the system call interface. Finally, we conclude with the promising facts this interface represents for rehosting processes.

3.1 Survey on embedded device security testing using HIL

In this Section, we survey papers related to embedded device security testing using hardware-in-the-loop (HIL) methods. Furthermore, we inspect their artifacts to estimate their reproducibility. Table 3.1 reports all experiments from the surveyed papers while table 3.2 presents a summary of their artifacts' status.

3.1.1 Publications

Muench et al. [124] address the state of memory corruptions in embedded devices and the lack of mechanisms to mitigate silent memory corruptions. For this purpose, they insert multiple vulnerabilities with independent trigger conditions on different classes of embedded systems. They observe different behavior ranging from crashing, rebooting, hanging, and malfunctioning to

Table 3.1: Experiments description in surveyed papers.

Publication	Experiment	Hardware	Artifact availability hardware[1]	source code
Wycinwyc [124]	- study effects of memory corruption on different class of embedded systems	Beaglebone Black	✓	✗
		Linksys EA6300v1	✗	
		Foscam FI8918W	✗	
		STM32 L152RE	✓	
	- measure mitigation execution overheads with fuzzing	STM32 L152RE	✓	✓
Avatar² [123]	- reproduce existing study	PLC Allen Bradley 1769-L16ER-BB1B	✓	✓
	- state transfer between concrete and symbolic execution modes			
	- record firmware execution	STM32 L152RE		
Avatar² examples	- forward memory accesses between emulators		✓	✓
	- state transfer between different targets	STM32 L152RE		
	- state transfer & peripheral modeling	nRF51-DK		
Avatar [193]	- backdoor detection in a masked ROM bootloader	Seagate ST3320413AS HDD	✗	✓
	- vulnerability research in a commercial Zigbee device	Redwire Econotag	✗	
	- helping reverse engineering the GSM stack of a phone	Motorola C118	✓	
Charm [170]	- feasibility (how long it takes to port a new driver)	Nexus 5X	✓	✓
	- performance (driver fuzzing with Syzkaller, driver initialization)			
	- record-and-replay (record bug PoC, measure execution overhead)			
	- bug finding (fuzzing with Syzkaller, sanitizing with KASAN)			
	- analyzing vulnerabilities with GDB (CVE-2016-3903, CVE-2016-2501, CVE-2016-2061)			
	- build driver exploit using GDB			
Prospect [97]	- performance impact on forwarding driver accesses using strace	324MHz embedded Linux MIPS with 16 MiB RAM	✗	✗
	- case study on proprietary fire alarm system (network fuzzing)	not disclosed		
Surrogates [101]	- measure performance impact of MMIO forwarding	Pico Computing E17FX70T	✗	✗
		custom JTAG adapter board		
		custom JTAG breakout / debug board		
		FriendlyARM Mini2440		
Inception [52]	- measure vulnerability detection via synthetic tests (Klocwork Test Suite)		✓	✓
	- validation tests (53200 tests)	LPC1850-DB1 & STM32 L152RE		
	- comparison with binary-only approaches	Xilinx ZedBoard FPGA		
	- measure timing overhead (Dhrystone benchmark and real-world applications)	STM32 L152RE		
	- compare recovering semantic from a binary to the source code with libopencm3			
	- security flaw detection with Juliet Test Suite 1.3 on FreeRTOS			
	- analysis of products during development phase (bootloader, chip SDK, payment terminal)	not disclosed	✗	✗
Mousse [105]	- performance evaluation	Pixel 3	✓	partially
	- measure coverage	Nexus 5X		
	- bugs and vulnerabilities research	Nexus 5		
Pretender [86]	- generate models for hardware peripherals (record, build and emulate)	STM32 L152RE	✓	✓
		STM32 F072RB		
		Maxim MAX32600MBED		
Conware [165]	- generate models for hardware peripherals (record, build and emulate)	Arduino Due (Atmel SMART SAM3X/A)	✓	✓
Frankenstein [153]	- heap overflow in device inquiry (CVE-2019-11516)	CYW20735	✓	✓
	- heap overflow in the reception of BLE PDUs (CVE-2019-13916)	CYW20819		
	- heap overflow on ACL packets buffer (CVE-2019-18614)			
FirmCorn [85]	- accuracy between virtual execution approaches	DLink (DIR-816, DIR-629, DIR-859, DIR-823G)	partially	partially
	- efficiency (benchmark nbench)	TPLink (WR940N, WR941N)		
	- stability	Ezviz C6CDahua (HFW5238M, HFW3236M)		
	- effectiveness			
Incision [174]	- correctness (control flow extraction, region inference, database improvement and error correction)	Huawei LTE R216h (ARMv7)	✓	✗
	- real-world usability (emulate Renault BCM, analysis the cryptography of Huawei R216h)	Renault BCM (Renesas V850ES)		
	- human effort (qualitative measure of complexity of manual intervention in database correction)			

1. The availability of the devices has been checked on google.com, amazon.com, digikey.com and ebay.com at the date 2021/12/15.

no effect. They propose mitigations against these vulnerabilities and measure their performance costs on fuzz testing.

Emulation facilitates the use of generic dynamic analysis techniques on firmware but suffers from limited device support and peripheral emulation. Therefore, many publications explore the idea of dynamically forwarding I/O operations to the physical device to improve emulation.

Avatar² [123] is a framework written in Python that aims to facilitate the interoperability between different dynamic binary analysis tools. In particular, it offers the power to use HIL techniques to plug devices into an emulator with the help of a debugger. Three use cases are presented within the publication. The first experiment reproduces the analysis of the HARVEY rootkit, while the second shows the ability to move the firmware execution state between concrete and symbolic execution modes. The third experiment demonstrates the capabilities of avatar² to forward peripherals accesses on the physical device from an emulator. This helps to record traces to replay and analyze them later without the device.

As an ancestor of avatar², Avatar [193] shares similar objectives and characteristics. Experiments gather around three case studies: backdoor detection in a masked ROM bootloader from a hard drive, vulnerability research in a commercial Zigbee device, the Econotag and helping reverse engineering the GSM stack of a Motorola C118 phone.

Prospect [97] targets embedded Linux systems by intercepting accesses to character devices in the filesystem and forwarding them to the physical device. The performance of the system is evaluated against an unknown 324 MHz embedded Linux MIPS system with 16MiB RAM using `strace`. In addition, an undisclosed proprietary fire alarm system is fuzzed as a security audit. The source code of the Prospect has not been made public.

Charm [170] focuses on device drivers for smartphones. It claims to support four different device drivers on different smartphones: camera and audio for LG Nexus 5X, GPU for Huawei Nexus 6P and IMU sensors for Samsung Galaxy S7. Experiments try to answer several questions on its feasibility, performance and capability to perform dynamic analysis techniques such as interactive debugging, fuzzing and record-and-replay on a Nexus 5X smartphone.

Surrogates [101] leverages specialized hardware to enable low-latency communication between the emulator and the system under test. It uses a custom FPGA to bridge the device's JTAG interface to the host's PCI Express bus. The implementation uses a Pico Computing E17FX70T with Xilinx Virtex5 FX70T FPGA because of its included ready-to-use PCI Express card. Unfortunately, the tool was never released and the FPGA board is not anymore commercially available. The experiments measure Surrogates' performance impact on MMIO forwarding and its ease to port it to two new target devices.

Inception [52] introduces symbolic execution to embedded systems. Similarly to Surrogates, it includes an FPGA-based debugger to provide high-speed and low-latency access to peripherals. But it differs in interfacing itself with the host via USB 3 instead of PCI Express. Experiments focus on validation of the design, benchmarking the performance, vulnerability detection and several use cases on proprietary systems.

Mousse [105] brings selective symbolic execution to environments that are too difficult to emulate because of specific hardware. The proposed system is evaluated around three aspects: performance, code coverage and vulnerability discovery; and against three smartphones: Pixel 3, Nexus 5X and Nexus 5.

Pretender [86] and Conware [165] focus on the challenges of automatically modeling hardware peripherals to enable better firmware emulation. Both follow a similar logic: first record traces of peripheral interactions, then use these traces to generate a model and finally plug the model into an emulator to allow the firmware to execute. Their contributions differ in the way of modeling the peripheral behavior from a recorded trace. Pretender uses machine learning while Conware employs automata representations. Both firmware datasets focus on the 32-bit ARM Cortex-M processor with a wide range of peripherals (e.g., timer, button, GPIO, I2C, USART, radio).

Frankenstein [153] takes memory snapshots of the wireless firmware on the device, mainly for Bluetooth and Wi-Fi. These captures are then patched to ease the emulation and fuzz them efficiently. The framework is used to discover three heap overflow vulnerabilities in implementations of the Bluetooth standard using the CYW20735 evaluation board.

FirmCorn [85] is a framework to fuzz IoT firmware. A collection of different firmware contexts are captured from the physical devices to be used as starting point for the fuzzing phase. Experiments target seven routers and three cameras. We evaluate multiple aspects of the proposed system such as accuracy, efficiency, stability and effectiveness.

Incision [174] tackles the challenge of combining static with dynamic analysis to help with the task of reverse engineering complex embedded systems. Execution traces are recorded in order to improve the static firmware analysis. The evaluation targets two physical devices, an LTE baseband unit and an automotive Body Control Module. The artifacts are not available. But at the time of writing, the authors plan to re-implement the source code with similar functionalities in a new open source framework called *Fugue*[1].

---

[1] https://github.com/UoBAutoSec/INCISION

3.1.2 Published artifacts' status

We observe that most publications release the source code of their proposed system and collected dataset. However, packaging their artifacts within a container or virtual machine images may improve their usability. In addition, scripts used to process generated data and plot figures for papers are rarely shared within the artifacts.

It is worth noting an initiative has been created to collect monolithic firmware used in publications:
https://github.com/ucsb-seclab/monolithic-firmware-collection

Table 3.2: Artifacts status in hardware in the loop papers surveyed.
⬠: source code   ◈: container   🐧: virtual machine

| Publication | Artifacts Packaging | | Hardware |
	Tool	Dataset	
Wycinwyc [124] [1]	⬠ 🐧	⬠ 🐧	STM32 L152RE
Avatar² [123] [2]	⬠ ◈	⬠ 🐧	PLC Allen Bradley 1769-L16ER-BB1B STM32 L152RE
Avatar² examples [3]	N/A	⬠	STM32 L152RE nRF51-DK
Avatar [193] [4]	⬠ ◈ 🐧	⬠	Seagate ST3320413AS HDD Econotag development board Motorola C118
Charm [170] [5]	⬠	⬠	Nexus 5X
Inception [52] [6]	⬠ ◈	⬠	Xilinx ZedBoard FPGA STM32 L152RE LPC1850-DB1
Mousse [105] [7]	⬠	⬠	Pixel 3
Pretender [86] [8]	⬠	⬠	STM32 L152RE STM32 F072RB Maxim MAX32600MBED
Conware [165] [9]	⬠	⬠	Arduino Due
Frankenstein [153] [10]	⬠	⬠	CYW20735 partially CYW20819
FirmCorn [85] [11]	⬠	⬠	DLink (DIR-816, DIR-629, DIR-859, DIR-823G) TPLink (WR940N, WR941N) Ezviz C6CDahua (HFW5238M, HFW3236M)
Incision [174] [12]			Huawei LTE R216h (ARMv7) Renault BCM (Renesas V850ES)

1. https://github.com/avatartwo/ndss18_muench2018you
2. https://github.com/avatartwo/bar18_avatar2
3. https://github.com/avatartwo/avatar2-examples
4. https://github.com/avatarone
5. https://trusslab.github.io/charm
6. https://github.com/Inception-framework
7. https://github.com/trusslab/mousse
8. https://github.com/ucsb-seclab/pretender
9. https://github.com/ucsb-seclab/conware
10. https://github.com/seemoo-lab/frankenstein
11. https://github.com/FIRMCORN-Fuzzing/FIRMCORN
12. https://github.com/UoBAutoSec/INCISION

3.2　Linux firmware rehosting approaches

As highlighted in the introduction, testing firmware poses a significant challenge: the systems are heterogeneous and operate in constrained environments. A recent trend attempts to *rehost* firmware to gain better control and deeper introspection on its execution. However, rehosting an entire system may not always be practical. Linux-based firmware is composed of various elements such as the kernel, drivers implemented as loadable kernel modules (LKM), libraries and applications. Previous research has focused on evaluating each of these elements as shown in Table 3.3.

Publication	Analysis focus					Technique		
	Hardware	Kernel	Driver	User program	Function	Emulation	HIL	Symbolic execution
RevNIC [45]	○	○	●	○	○	●	○	●
SymDrive [151]	○	○	●	○	○	●	○	●
Prospect [97]	○	○	●	●	○	●	●	○
Surrogates [101]	●	●	○	○	○	●	●	○
Firmalice [160]	●	●	○	○	●	○	○	●
Costin'16 [54]	○	○	○	●	○	●	○	○
Firmadyne [44]	○	○	○	●	○	●	○	○
Charm [170]	●	○	●	○	○	●	●	○
Firm-AFL [198]	○	○	○	●	○	●	○	○
FirmAE [98]	○	○	○	●	●	●	○	○
Mousse [105]	○	○	●	○	●	●	●	●
FirmCorn [85]	○	○	○	○	●	●	●	○
ECMO [94]	●	●	●	○	○	●	○	○
Jetset [95]	●	●	●	○	○	●	○	●
FirmGuide [104]	●	●	○	○	○	●	○	○
EQUAFL [199]	○	○	○	●	○	●	○	○
FirmSolo [28]	○	●	●	○	○	●	○	○

Table 3.3: Publications addressing Linux-based firmware rehosting.

3.2.1   Rehosting user mode applications

User mode processes are however a cornerstone for global system security. They implement most of the application logic and are often on the front line to parse external inputs. Although defense in-depth mechanisms exist to mitigate the impact of vulnerabilities in user mode processes, they offer a large attack surface. Moreover, their code is often proprietary contrary to what is written in the kernel and drivers, which makes it arduous to assert their security. This implies the importance of being able to do security testing and understanding what these applications achieve.

In this regard, we systematically review the existing approaches in literature focusing on rehosting and testing Linux processes, as well as exposing their limitations. In the following Figures 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6, the elements originating from the embedded system are represented in red while components participating in the emulation are highlighted in yellow and the parts located on the analysis workstation are shown in blue.

Costin et al. [54]   developed in 2016 an automated framework to analyze at-scale firmware images, and web service in particular. While their framework describes a pipeline for dynamic analysis, they discuss different approaches to rehost web services. Specifically, the approach they chose to implement executes the relevant programs in a `chroot` environment of the firmware filesystem. For architectural reasons, everything runs on top of a full-system emulation using a generic operating system. Those generic operating systems are pre-compiled Debian images composed of the kernel and its filesystem.

The downside of the approach is an excessive execution overhead generated by the combination of full-system emulation with a generic operating system, which is not directly relevant to the analysis. In addition, only the web interfaces are subject to testing and any interactions with lower abstraction layers, such as the kernel or the hardware peripherals, do not work as on the original device.

Firmadyne [44]   and later on **FirmAE** [98] propose to reduce the execution overhead by booting the firmware filesystem on top of a custom kernel. They argue that a high-level behavior from the web services is sufficient to perform its dynamic analysis properly. Their approach described a pipeline for dynamically analyzing firmware and their web services at scale. After extracting the firmware filesystem, an initial emulation phase records its booting interactions with the network and other system hardware interfaces. These records are then used to infer the expected environment. The firmware is then started in the inferred environment for deeper analysis. The
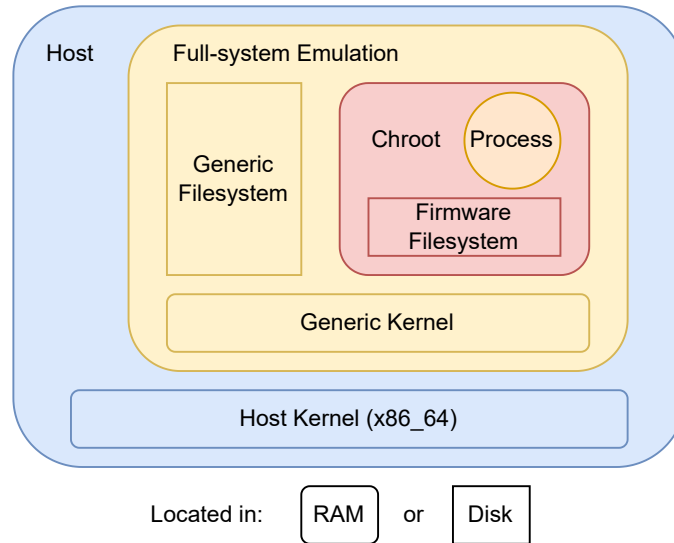
Figure 3.1: Costin'16 rehosting approach for web services.

main challenge in this approach is the setup of the environment for the kernel to boot correctly. In particular, it is related to improper booting sequence, network interfaces expected by the firmware, interactions with non-volatile storage memory (e.g., NVRAM) where configurations are often stored, and various other kernel issues.

This approach improves upon the state of the art but still suffers from high execution overhead because of full-system emulation. Furthermore, it may not always be possible to infer the correct environment for all firmware filesystems. Additionally, the interactions with hardware peripherals may also not be inferred or supported at all. Despite these limitations, FirmAE has demonstrated encouraging results by improving the original success rate of Firmadyne from 16.28% to 79.36% in their dataset. However, these results may be put into perspective when considering a different data set [61]. Nonetheless, both studies [61, 98] provide valuable insights into the key instrumentation that contribute the most to the success rate of rehosting, namely NVRAM, network and boot arbitration. Their success rate is measured by network reachability, which is tested by launching a `ping` command to the network services under test. While their evaluations show their approach is sufficient to find new vulnerabilities in web services, it is unlikely to be sufficient to run *accurately* more complex dynamic analysis, such as taint analysis, on the different applications. Furthermore, as demonstrated by Karonte [150], multi-binary services are widely used in applications.
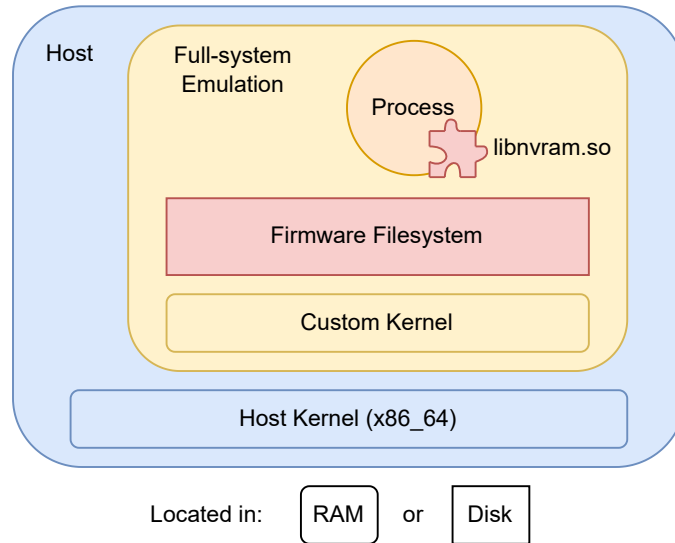
Figure 3.2: Firmadyne rehosting approach for web services.

FirmAFL [198]   aims to fuzz processes from a firmware. To improve execution speed, they have enhanced Firmadyne by providing the ability to switch the process between the user and the full-system mode emulation. They refer to their approach as *augmented process emulation.* The motivation behind the idea is to take advantage of the *faster* execution speed of process-level emulation while maintaining the *accuracy* of full-system mode when necessary. This approach addresses the challenges related to process synchronization and migration.
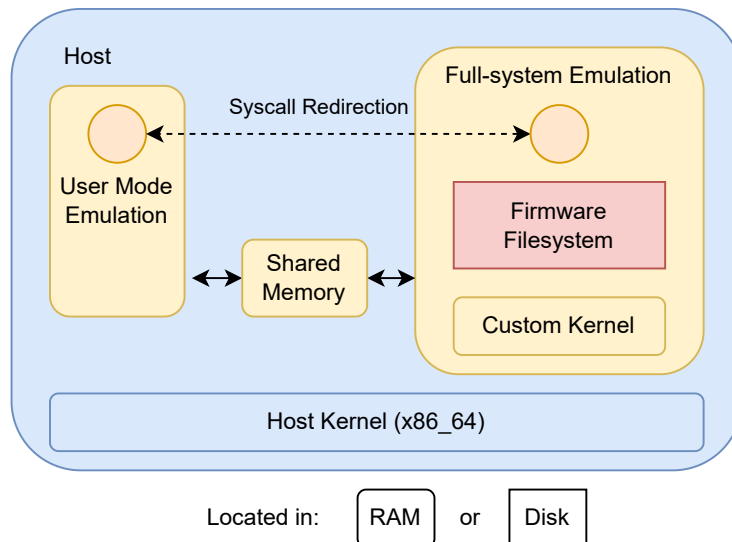
Figure 3.3: FirmAFL rehosting approach for web services.

The work addresses the execution speed problem. Therefore, most limitations regarding environment inferring and accuracy continue to be valid for FirmAFL.

EQUAFL [199]   seeks to further improve the execution speed of fuzzing by removing the need to switch to full-system emulation. The approach is twofold. Given a starting fuzzing point in a program, it first records the system environment setup in full-system using Firmadyne. The recorded information is then used to replay the same setup in a `chroot` environment on the host. Lastly, the target program execution can be resumed in user mode emulation. The recording process involves intercepting system calls of interest and collecting their information for the replay phase. The two most difficult behavior to reproduce are related to generating appropriate configuration files and handling network interactions.
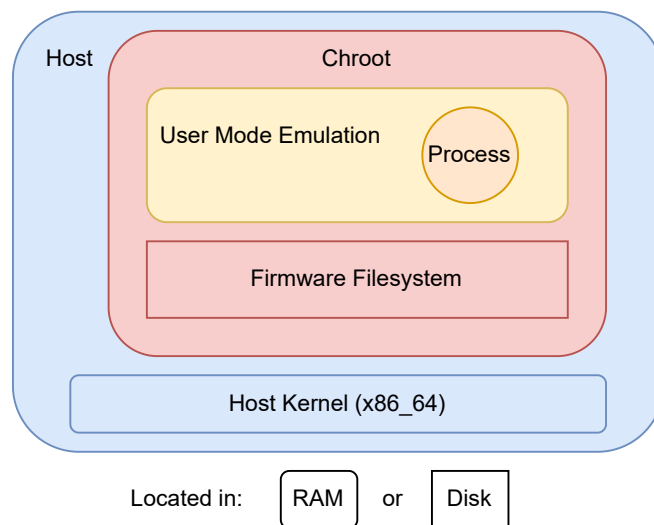
Figure 3.4: EQUAFL rehosting method.

Because the approach builds upon Firmadyne, it inherits similar limitations. In addition, it appears that while the execution speed is improved, the cost of analyzing at a large scale is incurred. The method requires manually unpacking the firmware and identifying environment variables using a decompiler.

FirmCorn [85] also focuses on increasing the efficiency of firmware fuzzing. First, the method conducts a static analysis of the firmware programs to identify potentially vulnerable code. This is achieved by measuring the cyclomatic complexity[2] of all functions and counting the number of times each function is called. Then, the program is started on the device to capture its context and migrated to the emulator. Three heuristics are used to further improve the emulation and fuzzing execution by replacing certain functions. *Unresolved functions* define functions that interact with newly dynamically allocated memory, they are emulated using the *uClibc* implementation. *Unnecessary functions* describe functions not interesting for fuzzing, such as printing and logging. *Hardware-specific functions* denote functions interacting with hardware components such as the NVRAM reads and writes.
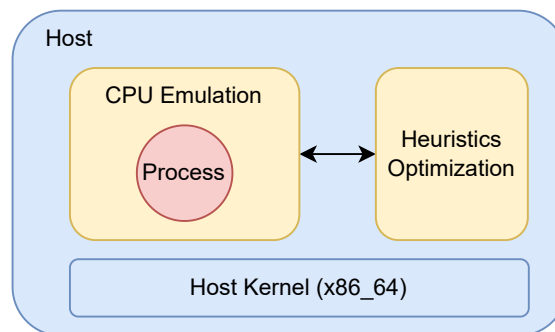
Figure 3.5: FirmCorn rehosting approach.

The main drawback of this approach lies in the fact that it prioritizes optimization for fuzzing at the expense of neglecting certain areas where vulnerabilities may reside. The algorithm for identifying vulnerable code is based on weak metrics. Additionally, the method only considers accesses in NVRAM but omits other types of hardware interactions such as character devices. Overall, the application of this rehosting technique is limited to the specific context of fuzzing and may not be easily adaptable to other scenarios.

PROSPECT [97] aims to address the limitation of using generic kernels for rehosting in the absence of hardware peripherals. For this reason, the paper describes a mechanism for intercepting system calls related to character devices and forwarding them to the physical device for re-execution. The reply is then returned to the process in the emulator. Later on, to reduce the execution overhead caused by forwarding system calls, the approach has been improved to incorporate caching of peripheral accesses [96].

---

[2]The cyclomatic complexity measures the complexity of a code by counting the number of linearly independent paths. It is calculated with the number of conditional statements such as if and loops.
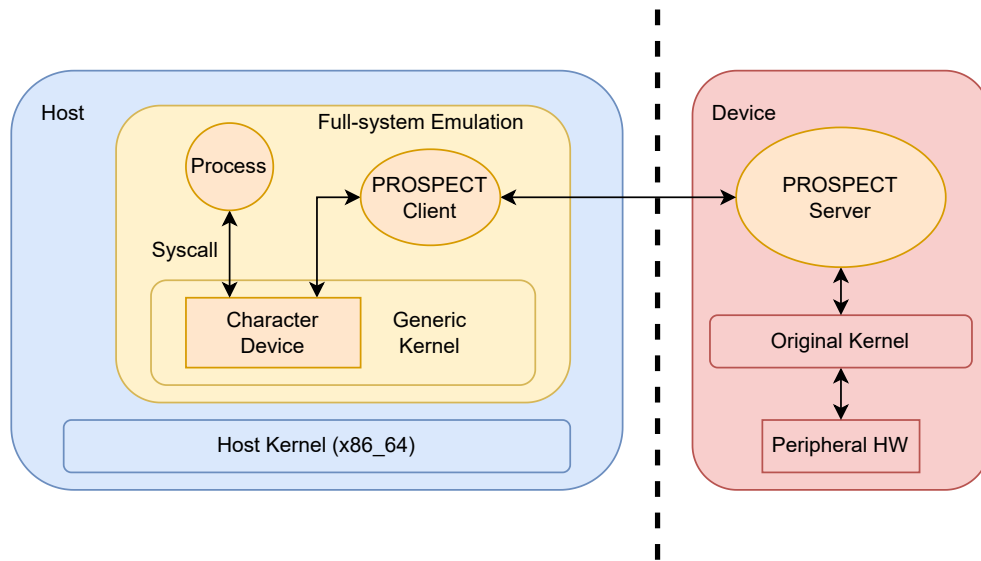
Figure 3.6: PROSPECT rehosting approach.

However, one of the major limitations of this approach lies in the way in which system calls are intercepted. The interception happens inside the kernel at the filesystem subsystem layer, and as acknowledged by its authors, it relies on the FUSE filesystem framework and its supported system calls. This implies that it is not capable of instrumenting other system calls not related to character devices. Although this is not the main objective of PROSPECT, it highlights its limitation in terms of interactions with the environment, such as those related to inter-process communication. Moreover, choosing to intercept system calls with a mechanism implemented inside the kernel prevents from using user mode emulation.

3.2.2   Summary

The choice to rehost certain components is characterized by the increase of complexity with the number of firmware parts chosen. First, application data alone presents limited interest. Usually, it is not located at a single point but spread across the system which makes its extraction more cumbersome. It can be stored in different memory types (e.g., NVRAM, ROM), files in the filesystem or even embedded in binaries. This also greatly limits the scope of analysis and vulnerability research. These are the reasons why current approaches prefer focusing on more comprehensive solutions.

Processes offer more interest for analysis, but migrating them from the device to the emulator can be challenging. This is because certain process resources such as TCP connections and devices are not part of the process

itself but a kernel abstraction.

Rehosting the firmware's filesystem provides a more accurate emulation of the original environment. This eliminates the need for emulating all processes interactions, such as loading dynamic libraries, opening configuration files or starting new programs.

During the boot process, the kernel is responsible for initializing the hardware components. Unlike the x86 architectures, which use the ACPI standard to dynamically discover and configure hardware components, many other architectures use a static device tree. This means that the kernel parses this data structure and, if errors are encountered, the kernel fails to boot. The vast variety of existing hardware components and peripherals makes it impossible for an emulator to support them all [71]. Therefore, some works have focused on inferring the expected hardware or guiding the kernel booting process [28, 94, 95, 104]. Instead, others [44, 54, 98, 198] have chosen to replace the firmware kernel with one that is compatible with the emulator. This improves the ability to easily modify the kernel to add instrumentation. However, this approach presents new challenges as programs expect a specific environment to execute properly, such as specific network interfaces.

The execution mode balances the trade-off between emulation speed and accuracy. By statically recompiling the application to the host ISA, the burden of emulating a different ISA is removed, resulting in increased execution speed as the program runs natively on the host. However, this requires either access to the source code of the application and a compatible toolchain or the ability to correctly decompile the binary. The latter is still an ongoing research problem because it can be difficult for certain ISA to properly identify the code from the data in the disassembly [27].

User mode emulation allows for the execution of foreign architecture binaries at the cost of slower execution. It is often used in conjunction with `chroot` to recreate the firmware filesystem environment. However, it lacks most of the hardware interactions.

Full-system emulation, in contrast, emulates the hardware behavior to enable execution for programs that interact directly with the hardware logic, such as kernels and drivers. But it is entirely dependent on the emulator support. For example, QEMU does not support all CPU architectures and does not implement all existing peripherals and platforms. In addition, peripheral hardware is often custom designed, and the documentation is rarely publicly available, making emulation more difficult without significant reverse-engineering efforts [71]. A workaround consists in combining the emulation with the physical device to forward hardware accesses.

Contrary to rehosting approaches focusing on the kernel, drivers and peripherals, as presented in Table 3.3, to the best of our knowledge, no firm-

ware rehosting methods that focus on user programs make use of symbolic models.

Each of the approaches makes different trade-offs to enable rehosting of various parts of the embedded system. They are summarized in Table 3.4.

Approach	Rehosted element				Execution mode					Other characteristics
	Process	Filesystem	Generic kernel	Custom kernel	CPU emulation	User mode emulation	Chroot	Full-system emulation	System call forwarding	
Costin'16 [54]	○	●	●	○	○	○	●	●	○	
Firmadyne [44]	○	●	○	●	○	○	○	●	○	Environment inferring
FirmAE [98]	○	●	○	●	○	○	○	●	○	Environment inferring
FirmAFL [198]	○	●	○	●	○	●	○	●	●	Shared memory
EQUAFL [199]	○	●	○	◐	○	●	●	○	○	
FirmCorn [85]	●	○	○	○	●	○	○	○	○	Process migration
PROSPECT [97]	○	●	○	●	○	○	○	●	●	System call caching [96]

Table 3.4: Summary of user mode rehosting approaches.

3.3   System call interception

Earlier, we examined various techniques for rehosting programs in user mode. The primary interface that separates processes from the kernel is system calls. In this Section, we provide background information and explore motivations for system call interception. In particular, we shed light on the inner workings of Linux system calls and present reasons for their interception (i.e., tracing, filtering, instrumentation, and forwarding). Next, we present in more details the common interception techniques used in Linux-based systems. Table 3.6 summarizes the main characteristics.

3.3.1   Motivation

A modern computing system typically consists of a kernel and a user space. The privileged operating system is running in kernel mode, while user programs reside in user space. Generally, user-mode applications interact with

the kernel via the system call interface. Hence, whenever a user program requires a service provided by the kernel, it initiates a system call.

The Linux kernel follows the "everything is a file" philosophy, and therefore many system calls are dedicated to files and filesystem management. Other examples of services exposed via system calls are address space, process management, signal handling, and ISA-specific services. Table 3.5 depicts examples of these system calls. User mode applications rarely need to issue the system calls directly, they are abstracted away by the standard system libraries (e.g., *libc*, *libstdc++*, *libm*, *libpthread*).

OS subcomponent	Example system calls
Files	open, read, write, close
Filesystem	mkdir, rmdir, rename
Process management	execve, fork, prctl
Address space management	mmap, brk
Signal handling	rt_sigaction, rt_sigprocmask, rt_sigreturn
Peripheral control	ioctl
Architecture specifics	arm_set_tls

Table 3.5: Example system calls for various Linux components.

When an application issues a system call, the execution is usually[3] transferred to the kernel via architecture-dependent instructions. For instance, Linux on x86 architectures uses software interrupts for 32-bit programs (int 0x80), and traps for 64-bit programs (sysenter), while resorting to exceptions for the ARM (svc) and MIPS (syscall) architectures. The execution context of the user process is saved. The kernel then determines the number of the requested system call, usually stored in a pre-defined register. This system call number is then used as an index to the kernel internal *system call table*, which stores a pointer to the individual system call handlers. Once the handler finishes, the process context is restored and a single return value is passed on success. On the opposite, a failure is typically indicated by a return value of -1 and sets errno, a special variable indicating the type of error.

Tracing

The least invasive form of system call interception is tracing. It is a technique that logs information about the executed system calls: the arguments, the return value, the potential errors and sometimes also about the process context such as the stack trace. Essentially, the resulting traces are recordings of the interaction between these programs and the kernel and give deep

---

[3]A notorious exception to this are system calls like gettimeofday or clock_gettime, which are dispatched in user space via the ELF virtual Dynamic Shared Object (vdso).

insights into the programs' behavior. Its applications are manifold and range from simple debugging to the analysis of complex malware and profiling.

While naïve tracers are just logging the executed system calls and their arguments, the need for more advanced traces is indisputable. A variety of system calls have pointers to locations in memory in user space as arguments, and without logging those memory contents, information is lost. Furthermore, additional information about the state of the process, such as the content of the stack, when issuing the system call may be relevant for the analysis. However, tracing can lead to considerable overhead in performance, as dereferencing and copying memory is slow, and system calls are frequently issued for most software. As a result, developing system call tracing engines requires trade-offs between performance and granularity of the retrieved information.

On Linux, the most established system call tracer is probably *strace* [14], which uses `ptrace`, a Unix system call for enabling debugging and tracing of processes. While *strace* provides thorough information about the executed system calls, a process can identify whether it is ptraced by querying the Linux kernel, and adversarial software, such as malware, typically changes its behavior if this is the case. Hence, other solutions for system call tracing are usually used in this context. For instance, the malware analysis framework Ninja [128] uses the Embedded Trace Macrocell, a hardware feature for ARM CPUs, while Cozzi et al. [56] used a modified version of System-Tap[4] [93] to trace system calls for analysis of Linux malware.

Instrumentation

There are several use cases for the instrumentation of system calls and their arguments or return values. For instance, modifying the arguments passed to the kernel can help during debugging and analysis of a program, as it allows the modification of the program's behavior. More common, however, is the modification of system call return values. In particular, changing the return value can exercise different paths in the program, which is useful during dynamic testing. For instance, the purposeful injection of error return codes helps for identifying bugs and vulnerabilities in programs [186,195]. Furthermore, instrumentation of system calls is often used by malicious software. Rootkits are frequently instrumenting system call arguments and return values, for instance, to modify the behavior of other userland processes, or for evading malware detection systems.

---

[4]SystemTap is a tool to write scripts which collect information of a running Linux system.

Filtering & Sandboxing

System calls are the main interface for processes to modify the state of the system. Hence, filtering out malicious or unwanted system calls is a central goal in any sandboxing approach. Moreover, filtering system calls can also be a component in the principle of the least privileges: not all userland processes require having access to all implemented system calls. Therefore, by restricting a process to the system calls it is supposed to need, the attack surface of the kernel is limited.

The *seccomp* subsystem is a popular mechanism for system call filtering on Linux. It can be accessed through the `prctl` system call and allows a process to be placed in "secure computing mode" which only allows a limited subset of system calls. The *seccomp-bpf* extension offers more precise control over system calls by specifying policies through Berkeley Packet Filter (BPF) rules. In addition to actions such as killing a process, this extension allows for the handling of system call events in user space, through methods such as sending a signal or via a ptrace-based tracer.

Forwarding

An emerging trend for system call interception is the action of forwarding system calls from one system to another. As a result, user space applications do not interact with their native kernel, but instead with a foreign one. While this may appear non-intuitive at first, it is the underlying concept for popular user-mode emulators, such as *qemu-user* [36]. Furthermore, an emerging use case for system call forwarding is the analysis of Linux-based embedded devices as demonstrated with PROSPECT [96, 97] and Firm-AFL [198]. The key idea is that resource-intensive analysis of user space applications is carried out on an analysis host, while system calls interacting with the hardware of the device are forwarded.

Another notable example of system call forwarding, although not targeting Linux system calls, is a framework written by Martignoni et al. [112]. It forwards and executes a subset of Windows system calls from a user host to an analysis host in the cloud for facilitating advanced runtime malware analysis.

Distributed operating systems rely on forwarding mechanisms to seamlessly continue the execution of migrated processes across different nodes within a cluster [34, 107, 180, 181].

3.3.2   Main system call interception techniques

The need for observing, instrumenting, filtering, and forwarding system calls leads to a variety of techniques, each with its advantages and drawbacks. Although system call interception techniques differ widely in the way they operate and the capabilities they provide, they all have in common that the execution of a hook comes alongside the execution of the actual system call. Hence, we will refer to different interception techniques as hooking techniques for the sake of readability, regardless of the goal and motivation of the interception.

Generally speaking, hooking techniques usually fall into one of the following three categories:

1. **User mode-based.** These techniques register and execute their hook in user space. While some of these techniques require assistance from the kernel, the core logic of the hook resides in user mode.

2. **Kernel-based.** In opposition to the user mode-based techniques, the code of the hook is executed in the kernel context.

3. **External.** Another option is the insertion and execution of hooks outside the operating system. This scenario includes for instance hypervisor-based hooking or the usage of special debug interfaces.

While we will use this categorization for structuring the remainder of this section, a variety of other properties can be used to characterize a given system call interception technique. For instance, one key property framing the performance, requirements and accessible information is the hooking *mechanism*, i.e., how execution is transferred from entering or exiting a system call to the actual hook. Typically, a hook is either reached via a *trampoline*, or a *trap*. The former is a direct transfer of execution to the hook via a jump instruction, whereas a trap uses some kind of interrupt mechanism during the execution, which eventually transfers the control flow to the hook.

Other characteristics include whether the hook is inserted statically, e.g., during compile time, dynamically during program execution, or whether the technique needs to be supported by a given kernel.

We will highlight the diversity of system call hooking techniques on Linux by discussing several different techniques. While the list of presented techniques is not exhaustive, it will give a good overview of the current state of the art. Table 3.6 provides a summary of them.

Instrumentation location		Name	Hook mechanism				Insertion method		Hook location		
			Trap	Trampoline	Substitution	Branch condition	Static	Dynamic	User	Kernel	External
User		Library injection	○	○	●	○	●	●	●	○	○
		Binary rewriting	●	●	○	○	●	●	●	○	○
		ptrace	●	○	○	○	○	●	○	●	○
		seccomp	○	○	○	●	○	●	○	●	○
Kernel		Linux tracepoint	○	○	○	●	●	○	○	●	○
		Kprobes	●	●	○	○	○	●	○	●	○
		Uprobes	●	○	○	○	○	●	○	●	○
		Character device	○	○	●	○	●	●	○	●	○
		Kernel patching	●	●	●	●	●	●	○	●	○
External		Hypervisor	●	○	●	○	○	●	○	○	●
		Hardware tracing	●	○	○	○	○	●	○	○	●
		Hardware debugging	●	○	○	○	○	●	○	○	●

Table 3.6: Summary of Linux hooking methods.

User mode-based

Approaches that implement instrumentation using user mode programs have the advantage of increased stability and portability across different platforms. However, this comes at the cost of reduced stealth, which can be a significant factor in the context of malware analysis.

Library Injection. The principle of this technique is to inject a shared library responsible for instrumentation into another process, either during startup or run-time. The most famous example for this is probably `LD_PRELOAD` based hooking. By exploiting the behavior of the dynamic linker/loader, this technique overrides the function references in the Global Offset Table (GOT) to redirect them to other locations. While this does not give full control and visibility over system calls, it allows the instrumentation of the according wrapper functions of the standard C library. Although the simplicity of this approach is remarkable, it is very prone to miss system calls not issued by the instrumented libraries, and it is not applicable for statically linked binaries. Nonetheless, `LD_PRELOAD` serves as the building block for other techniques,

such as *syscall_intercept* [20], which preloads a stub for disassembling the standard C library and replacing system call instructions with a trampoline.

A similar, but more sophisticated approach is implemented by Frida [149], which injects a JavaScript interpreter inside a running process to provide various dynamic instrumentation capabilities, including system call interception. The script has full access to memory, and process' functions and can instrument its execution. However, the overhead introduced in memory and CPU time is significant [106].

Binary Rewriting.   Essentially, binary rewriting consists of disassembling the binary code, adding instrumentation, and reassembling the modified code before execution. This is either performed *offline*, using static binary rewriting tools such as Ramblr [184], Retrowrite [63] and ZAFL [126], or *online* during program run-time, which can be achieved with frameworks such as DynamoRIO [41] and PIN [108]. While static rewriting techniques are somewhat limited, as they rely on correct disassembly and can't cope with obfuscated code and packed binaries, dynamic rewriting typically introduces a significant performance overhead during the process execution, as instrumentation code is typically added to every basic block. Hence, some systems aim to provide lighter solutions, such as *VARAN* [90], which uses *selective binary rewriting* where only system calls instructions are replaced by either a trap or a trampoline whenever a code page is loaded into memory. While this beats the performance of more traditional binary rewriting frameworks, system calls originating from self-modifying code as often seen in malware can be missed by this approach.

Process Trace.   One of the most versatile building blocks for user mode-based hooking on Linux is process trace, or, in short, *ptrace*, which is provided by the kernel via the ptrace system call. Despite its name, ptrace allows one process to introspect and modify the registers and memory contents of another process with the help of the kernel. Additionally, trap handlers for special events such as system calls can be registered. A variety of standard debugging tools are built on top of this facility, such as strace, GDB and ltrace, as well as advanced dynamic analysis tools [183, 197].

The main drawback of this approach lies in the multiple context switches between the tracer and the tracee, resulting in high execution overhead. It also suffers from TOCTOU attacks [146] which are difficult to resolve due to the lack of atomicity in this interception mechanism.

Seccomp.   Like ptrace, seccomp allows registering hooks for a system call with the help of the kernel. In particular, it allows a process to register Berke-

ley Packet Filter (BPF) [5] rules to match system calls and their arguments. The kernel will evaluate subsequent system calls against this filter and carry out an action encoded in the return value of the filter. Possible actions include termination of the process, returning an error for the system call, passing control to the program via signals, or notifying a `ptrace`-based tracer.

Although the interception mechanism resides in the kernel, this approach and the ptrace method are classified as user mode-based due to the execution of actual system call instrumentation.

Kernel-based

The Linux kernel offers various tracing, hooking, and probing facilities for all sorts of events, including the execution of system calls. A standard way to access these functions are Loadable Kernel Modules (LKM), which can be used for flexible instrumentation of user applications from kernel space. By running at higher privilege mode, an LKM can manipulate the target process address space while having access to all kernel data structures and functions, and perform system call interception transparently to the user program.

However, to build such modules, the toolchain and headers originally used to compile the kernel are required, which is frequently impossible or very difficult to obtain [28]. In the following, we describe various APIs and techniques accessible from LKM to enable system call hooking.

Tracepoints.   The Linux kernel allows inserting tracepoints in its source code. They can later be hooked with handler functions, called probes, at runtime. A tracepoint can be turned on or off by registering or unregistering the probe. Various tracepoints, including some for system calls, are already present in the Linux kernel and can be accessed via the API function `trace_tracepointname()` and the `TRACE_EVENT()` macro. Although this mechanism introduces only minimal overhead, it does not allow interception of the system call beyond simple tracing and can be inserted without recompiling the kernel code.

Kprobes.   Dynamic insertion of hooking points can be achieved with *Kprobes*, the trap-based dynamic tracing system of the kernel. Kprobes allows placing a software breakpoint *almost anywhere* in the kernel code [99] and allows registration of pre-, post- and fault handler routines. When the breakpoint is hit, the CPU traps into an interrupt context, saves its state and executes

---

[5]The replacement of the traditional BPF virtual machine with the extended BPF (eBPF) version has been a topic of debate within the community, due to potential security concerns [10]

the pre-handler instrumentation. Then, it executes the saved instruction and the post-handler function before returning after the breakpoint.

As trapping and switching to interrupt context usually comes with a significant performance penalty, *optimized Kprobes* replace the software interrupt instruction with a trampoline when a strict set of requirements is met. The trampoline handler simulates the breakpoint behavior by pushing CPU's registers onto the stack. Then, the Kprobe handlers and the copied instruction are executed, before the state is restored and execution continued at the original location.

For hooking system calls with Kprobes, two strategies are viable: registering a Kprobe for every possible system call routine, or registering a single Kprobe in the system call dispatching function. The former approach is insufficient in the presence of rootkits, as they typically alter the system call table, while the latter has additional hurdles to overcome: As system call dispatching is a critical operation, interrupts may be disabled and the code locations may be excluded from the authorized locations for Kprobes.

Uprobes.   Uprobes allow kernel modules to instrument user space applications by replacing a given instruction with a software breakpoint. The instructions to instrument are identified by tuples composed of the inode of the binary and their offset from the start. When the breakpoint is hit, an additional filter, specified at uprobe creation, examines additional conditions to meet before transferring control to the hook or continuing execution in user space. While this mechanism is especially useful for user mode tracers such as *perf* [11] or *ftrace* [4], it has similar limitations as static binary rewriting as system call instructions need to be identified before the program starts.

Character device.   User space applications often interact with drivers and hardware using special character devices. Starting from this assumption, a kernel module can provide a custom character device to co-operating user mode programs to intercept file-related system calls [97, 125]. Although this method has its benefits when the ultimate goal is forwarding interaction with device drivers, it has severe limitations as a generic system call hooking strategy.

Kernel patching.   If all the aforementioned methods are not applicable, a module can always fall back to the most direct hooking technique: overwriting kernel code—or data—during runtime [43, 65, 121, 169]. For instance, single entries in the system call table could be changed, the complete system call table could be replaced, or instructions from the dispatching routines could be rewritten. However, this method offers the lowest level of porta-

bility and usually requires eluding various runtime protections aiming to prevent kernel memory tampering.

External

System call interception from outside the operating system usually has the advantage of being undetectable both by the operating system and user space. Furthermore, it can lead to better performance and full access to memory and any low-level machine state information. However, most of the time a semantic gap has to be bridged, as internal details of the kernel, such as data structures, are lost.

Hypervisor.   System call interception at the hypervisor level is deeply coupled with the virtualization approach used. Some hypervisors rely on CPU *hardware virtualization extension* to run a guest operating system. When a sensitive instruction (e.g., software interrupts for system calls) is issued, the control flow moves to the hypervisor which then decides what action to take. Indeed, system calls can be intercepted directly by the hypervisor before reaching the guest kernel. It is particularly useful when the instrumentation needs to be undetectable by the program being analyzed, like for malware analysis [62].

On the other hand, when the hypervisor relies on *dynamic binary translation* to provide a different execution environment, it is possible to rewrite the system calls to include the desired instrumentation. Dedicated hypervisor tools such as PANDA [64], provide hooks for various events on the guest, including the occurrence of system calls.

A third case is illustrated by approaches based on KVM. Dune [35] and later ELKM [139] use a library to convert system calls into hypercalls which are then instrumented by the hypervisor.

Hardware Tracing.   Modern CPUs are oftentimes equipped with specialized tracing capabilities such as Intel's *Processor Trace (PT)*, or ARM's *Embedded Trace Macrocell (ETM)*. Unfortunately, these capabilities do not allow for any kind of system call hooking outside the scope of tracing on its own, as the generation of the trace is typically completely decoupled from the execution of a binary. However, systems like Griffin [77] demonstrate that they indeed can be used to introduce blocking checks in the event of system calls. Moreover, tracing system calls with these hardware features is especially useful for advanced malware analysis, as illustrated by Ninja [128].

One limitation of hardware tracing is its lack of portability. The hardware and software used for hardware tracing are deeply coupled with the architecture. This makes it difficult to apply a generic solution or to compare the

behavior of systems that use different architectures. This is particularly true given the wide range of architectures employed by embedded systems. A second limitation concerns the limited number of events that can be traced at once. Most tracing solutions have a fixed amount of hardware resources that are dedicated to tracing. This means that it may not be possible to trace all the events of interest at a very high frequency without deleting old records.

Hardware Debugging.   In comparison to tracing, hardware debugging facilities can alter CPU state and memory content in a completely transparent way to the underlying operating system and user space. While this effectively allows large flexibility with system call interception, it requires low-level knowledge about the target operating system. Nonetheless, the viability of this approach for system call interception is shown by OpenST [196], which performs system introspection for Linux on ARM using JTAG and OpenOCD. However, retrieving and rebuilding information through a low-speed debug port can introduce high execution overhead. In addition, debugging ports are not always present, accessible, or simply enabled on devices.

3.4 Problem statement

The previously discussed methods for rehosting processes introduce promising concepts, but overall offer a best-effort approach to approximate the minimum requirements needed to run a program. Although these methods have been successful in identifying new vulnerabilities via fuzzing, they are not sufficient for performing more complex analyses on program execution such as taint tracking and symbolic execution. Furthermore, these methods do not fully reproduce the context of the embedded system and its comprehensive interactions with the environment. For instance, Firmadyne focuses on the booting process of the firmware image but does not verify the correctness of the rest of the execution, particularly concerning hardware accesses. Moreover, the rehosting method is designed to analyze firmware *at scale* and efforts are focused to improve the overall success over the dataset. Alternatively, PROSPECT is limited to character device accesses only.

Therefore, it is crucial to continue proposing new approaches to reproduce environments and interactions faithful to the original devices. In the next chapter, we will examine how different methods of system call instrumentation can be used to rehost processes.

Chapter 4

System call forwarding for Linux processes

Previous research has mainly focused on the ability to boot and run the firmware but with less emphasis placed on its execution accuracy. In this chapter, we start by highlighting the shortcomings of existing approaches through an example of an embedded system presenting a variety of interactions with a rich environment, including local wireless network and cloud services. We then examine the challenges the process abstraction poses for rehosting and investigate the solutions to handle them. Finally, we propose a novel approach for providing partial execution to a process by forwarding its system calls from an emulator to its original environment on a physical device.

4.1   Motivation of the approach

4.1.1   Motivational example

We observed that existing approaches primarily focus on extracting the programs from its filesystem and apply *best-effort* methods to approximate the minimum required for the program to run. While this is sufficient to identify some vulnerabilities, it fails to capture the complete context of the embedded system and lacks comprehensive interactions with the environment as we will see in the following example.

A program rarely runs alone in a Linux-based system. It is indeed the very point of an operating system to offer the ability to share the execution time between multiple programs. The rehosting technique implemented via Firmadyne and FirmAE illustrates this by trying to identify all the firmware services and start them properly. But that is not always sufficient to capture all firmware interactions with the rest of its environment. As highlighted before, the original environment is not reproduced in its accuracy: the kernel
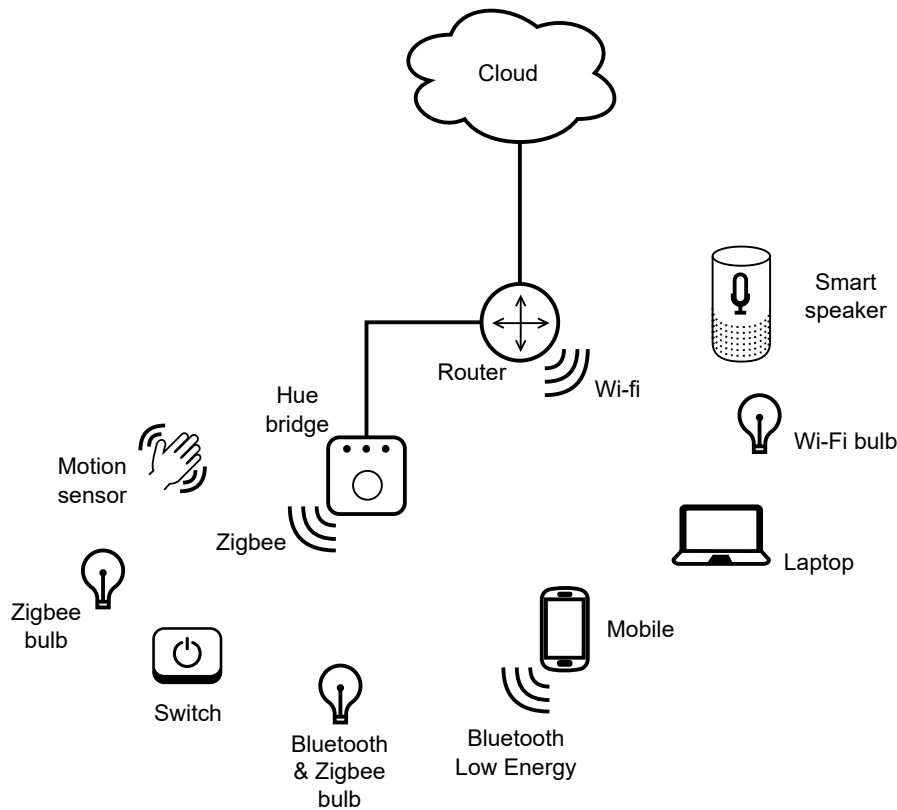
Figure 4.1: Philips Hue devices in the home network.

version is different, which makes it impossible to load kernel modules; some hardware components may not be supported by the emulator; not all firmware services could be started. Therefore, some key program inputs may be missing.

A representative example of firmware evolving in a complex environment is the firmware of the *Hue Bridge. Philips Hue* is a popular smart home automation system that offers to control lighting devices wirelessly. It is composed of different types of light devices: bulbs, light strips, as well as switches, motion sensors and a bridge. The bridge aims to provide interoperability between the local network and the wireless protocols used by the devices. To enhance user experience, the system integrates with different cloud services, for instance enabling users to control the lights from their smartphones when outside their home, or to launch voice commands via smart speakers. Figure 4.1 shows a typical Philips Hue home network environment.

To facilitate interactions, and support different use cases, multiple protocols are supported: Zigbee Light Link, Bluetooth Low Energy and Wi-Fi.
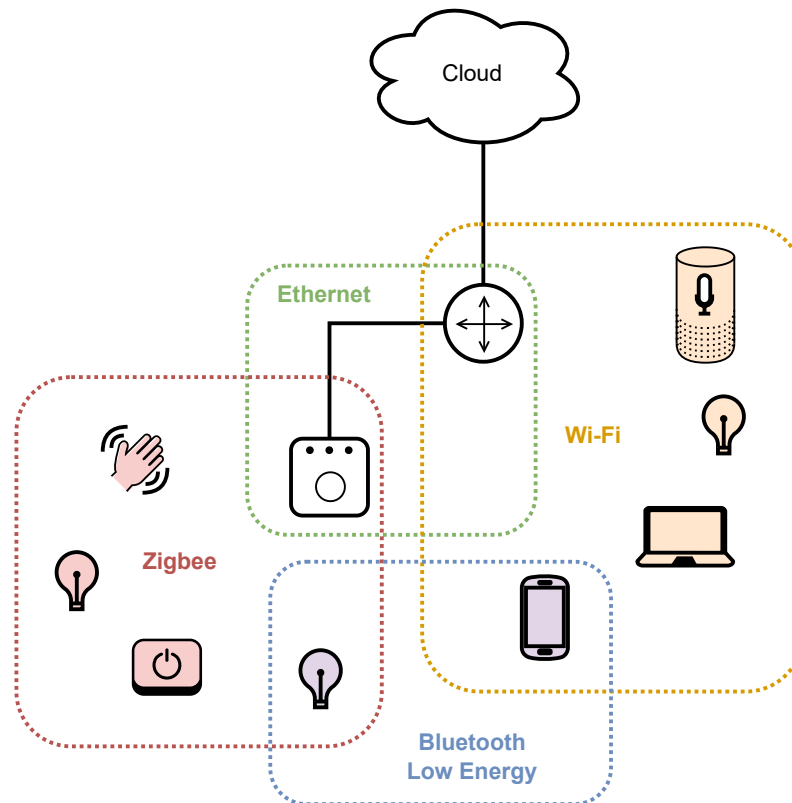
Figure 4.2: Set of protocols used by Philips Hue system.

The core of the system is the *Hue Bridge*. It connects to the router via Ethernet and the devices via the Zigbee mesh network. In addition to the Zigbee Light Link protocol, some devices support the Bluetooth Low Energy protocol. This allows sending commands directly to the device without the need for a bridge. Figure 4.2 illustrates the different protocols involved in the system.

In addition to its technological choices, the *Philips Hue* system presents security challenges [25, 119, 152]. Its privacy has also been studied. Thiery et al. [173] discussed how the state of lights and other sensors within the house help understand user habits. By cross-referencing these data with those of their partners, companies can infer broader behavior about their customers. It is even more true when *Philips Hue* system is advertised as being well integrated with other home automation systems, in particular *Google Assistant*, *Amazon Alexa*, *Apple HomeKit* and *Microsoft Cortana*. In their experiments with multiple devices, Thiery et al. [173] measures that 75.35% of data is sent to servers located in the US while the device resides in Europe. This raises questions about data sovereignty.

Thiery et al. [173] analyzed the traffic from the network. However, most

of the traffic was encrypted, and it was not always possible to set up a *man-in-the-middle* attack. The authors observed recurrent communication between the bridge and the cloud server (hosted by Google) but because communications are encrypted they were unable to precisely analyze them. Nonetheless, they supposed it is used either to notify the light status or as a keep-alive mechanism. This is why it is interesting to be able to have a deep insight into the device's internals. Because the *Hue Bridge* is at the heart of the whole system, rehosting and instrumenting its programs would let us better understand what kind of information is shared with devices and cloud services.

At the hardware level, the *Hue Bridge* can be divided into two parts: the Linux system and the Zigbee modem. Communication between both components is handled via a UART and appears as a `tty` device under the dev directory: `/dev/ttyZigbee`. At the software level, the application `ipbridge` plays a central role in forwarding the commands to the Zigbee modem, monitoring the status of the lamps and communicating with cloud services.
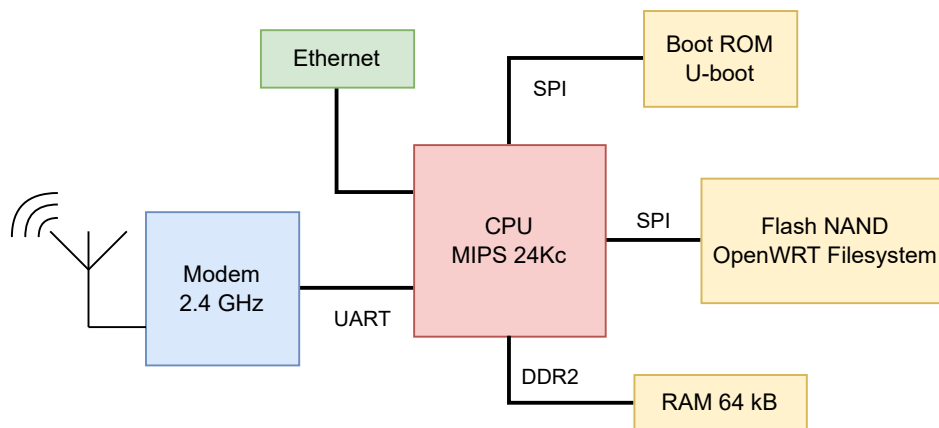
Figure 4.3: Hue Bridge block diagram.

This illustrates the fact that rehosting only a part of the system, for instance, the Linux one, is not sufficient for it to function properly. Therefore, it is crucial for the rehosted application to maintain its relations with its original environment.

4.1.2   Our Approach

Our approach consists of intercepting system calls, deciding if their execution should be carried out locally or remotely, invoking them on the chosen platform, and returning them to the application. For certain system calls that modify the process structure, we perform additional operations to keep the

representations of the process in the two kernels synchronized. This is particularly the case for the `mmap` family which alters the structure of the address space (Figure 4.4).
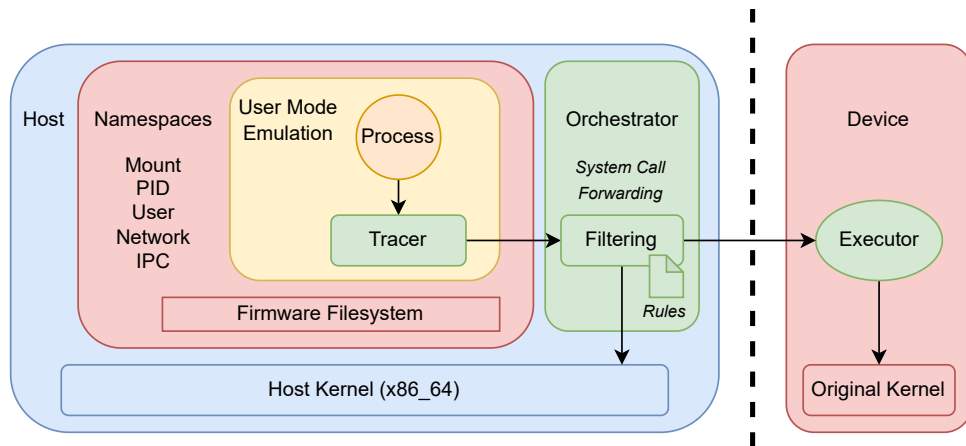
Figure 4.4: Proposed system call forwarding approach.

We started by observing the typical composition of Linux-based systems. We summarize those components in three[1] layers: the hardware, the kernel and the user space. For a user-mode application, the system call interface is the main point of interaction with the kernel and its hardware resources. It is a public and stable API and ABI. Unlike low-level hardware accesses happening between the kernel and the hardware such as I/O operations, the system call interactions are more general and suffer less from latency issues [170]. Furthermore, many techniques exist for instrumenting system calls as we saw in Section 3.3: tracing, sandboxing, filtering, forwarding, etc. As a result, all these characteristics make the system call interface a good candidate for rehosting programs.

We have chosen to rely on interception methods that are external to the executing program to avoid the need for modifications in the programs. Thus, the compatible mechanisms are user mode emulation, ptrace, seccomp and other kernel-based interceptions. In our implementation described in Chapter 5, we use QEMU user mode emulation with several *Linux namespaces* to reproduce the original firmware environment.

---

[1]We do not consider the hypervisor in our case. It would lay down between the hardware and the kernel.

4.2   Challenges

As seen in the motivation in 4.1, some existing work has been done to pro-
vide a *limited* way for forwarding system calls. FirmAFL [198] shares mem-
ory between the two emulators, but system calls are almost always executed
in full-emulation mode. PROSPECT [97] can forward system calls to the
physical device, but its interception mechanism lies in the FUSE filesystem,
limiting it to a certain category of system calls. In this Section we will look
at the different challenges faced when forwarding system calls between two
separate kernels.

4.2.1   The process duality: between user and kernel mode

A process is composed of elements in both user and kernel modes.

The execution is divided between user and kernel mode. In user mode,
application code is executed together with its linked libraries, while in the
kernel a thread is maintained for each user process. When a system call
is invoked, the control flow jumps to its associated thread where the user
context is saved and further operations are executed on behalf of the process.

The kernel maintains several structures for the process. The hierarchy
and relations between processes are stored in the kernel process table. The
kernel maintains multiple tables for file management: the file descriptor ta-
ble, the open file table and the inode table.

Inter-process communication (IPC) mechanisms are partly controlled by
the kernel and the user process. For instance, the process registers handlers
in the kernel to properly receive signals [13]. Shared memory lets processes
create a bridge between different address spaces.

To access user space memory safely, the kernel uses a dedicated API (de-
fined under the `usaccess.h` files). Table 4.1 shows the main functions of the
interface.

Therefore, from user mode, the process does not have full control over
its state and the kernel resources it uses.

Name	Function purpose
`clear_user`	Zero *n* bytes of memory in user space
`access_ok`	Check whether a pointer is valid for user space access
`copy_from_user`	Copy *size* bytes of memory from user to kernel space
`copy_to_user`	Copy *size* bytes of memory from kernel to user space
`get_user`	Macro to copy a variable from user space
`put_user`	Macro to copy a variable to user space
`strncpy_from_user`	Copy a string of *count* maximun length from user to kernel space
`strlen_user`	Return the size of a string from user space

Table 4.1: Linux user space memory access API

4.2.2   Filtering

Forwarding all system calls to the remote kernel is not always necessary, and provides optimization opportunities. In addition, as we will see in the rest of the chapter, certain system calls may need to be executed on both kernels to keep the process structures synchronized. To decide which system call to execute, and where to execute it, a method for *filtering* system calls and the data they carry in their arguments is necessary.

4.2.3   Classification

The two most crucial aspects of forwarding system calls are the information that needs to be transferred and the operations required to keep processes synchronized. Most existing classifications are based on the function of the system call, i.e, the type of operations the system call executes and the subsystem it interacts with. However, these groupings do not consider the interface in itself and the information transferred between user and kernel modes.

It exists mainly two types of arguments:

- *Direct variables*: integers where all the information is contained in its value (e.g., `int option`, `int pid`, `uid_t uid`, `size_t len`, `off_t offset`, `unsigned int fd`, `int flags`, `umode_t mode`),

- *Indirect variables*: pointers to memory blocks in user space such as buffers (`char *buf`, `void *buf`), strings (`char *filename`, `const char *pathname`), structures (`struct tms *tbuf`) and arrays (`int *fildes`). These are often associated with a *direction* for the transfer: is the memory copied from user space to kernel, or the opposite, or both? In the kernel, this is reflected by the usage of the user space memory access API (Table 4.1). In user mode, clues exist in the argument definitions via the usage of qualifiers such as `const` and `__user`.

However, this does not grasp the full range of corner cases offered by system calls with variable argument lengths such as `ioctl()` and `fcntl()`.

The `mmap` system call family is particularly noteworthy. The *vma* argument represents an address that is the start of a memory region and could be misinterpreted as a pointer. But it does not refer to any data in itself, the information is the address, thus the value of the argument, so it should belong to the integer category.

Another corner case concerns linked lists via `set_robust_list()`. The system call is used to set the robustness of a process's futexes (fast userspace mutexes), i.e., the behavior of the mutex when the process terminates. It is a mechanism used to avoid deadlocks when a process terminates accidentally.

The system call takes as an argument a pointer to a structure representing the head of the linked list: `struct robust_list_head *head`. However, the kernel only stores the value of the pointer. It is only when the process is terminated that user memory is accessed to traverse the linked list. Because the management of the futexes is delegated to the user process, most of the structures are stored in the user memory space. So the kernel keeps only pointers to these structures and therefore needs to do many reads to user memory to set the state of futexes. In our approach, we have set aside this particular case to handle all the futex management with the local kernel and avoid any forwarding. As we will further see in the Chapter 5, not all system calls need to be forwarded and the filtering mechanism helps us achieve that.

Taking these observations into account, we identified four categories of system calls:

- **Category A** modifies the structure of the process.

- **Category B** exchanges information with the kernel via pointers to user space buffers.

- **Category C** gathers simple functions where all the information is contained in their arguments.

- **Category D** is special because it is the vDSO library that invokes direct calls to the kernel without any context switches.

We will now, discuss each group with examples.

A. Modifying the address space

These system calls operate directly on the kernel structures of the process address space. They modify it in a way that is hard to control from user mode. However, their number is limited, and it is possible to replicate their effects to keep both process structures synchronized.

B. Exchange user space data

These system calls are characterized by their interactions with the process address space through memory reads and writes. As mentioned in Table 4.1, the Linux kernel uses a specific interface that takes the pointer and the size as arguments. The size of structures is part of the system call ABI. In order to preserve backward compatibility for user space applications and libraries [8] [9], the Linux kernel developers are making great efforts to not change the ABI. In contrast, buffers that are not strings are often followed by another argument defining their size.

```
1  void *mmap(void *addr, size_t length, int prot, int flags,
2           int fd, off_t offset);
3  int brk(void *addr);
4  int mlock(const void *addr, size_t len);
5  pid_t fork(void);
6  pid_t vfork(void);
7  int clone(unsigned int flags, void *stack, int *parent_tid,
8           unsigned long tls, int *child_tid);
9  int execve(const char *pathname, char *const argv[],
10          char *const envp[]);
```

Listing 4.1: Examples of system call prototypes that modify the process address space.

```
1  int open(const char *pathname, int flags, mode_t mode);
2  ssize_t read(int fd, void *buf, size_t count);
3  int recv(int s, void *buf, int len, unsigned int flags);
4  int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Listing 4.2: Examples of system call prototypes that exchange user space data.

C. Simple function

These are the simplest system calls to forward as all the information exchanged with the kernel and the resulting effects (from the kernel routine) they trigger are contained in their parameters and the return value. This means that only the registers must be copied on the system call entry and exit.

```
1  pid_t getpid(void);
2  int socket(int domain, int type, int protocol);
3  int flock(int fd, int operation);
4  void sync(void);
5  int kill(pid_t pid, int sig);
```

Listing 4.3: Examples of system call prototypes that exchange user space data.

D. vDSO - Virtual Dynamic Shared Object

For some system calls, the time spent in the kernel is negligible compared to the overhead of the context switching. For this reason, the kernel maps

a shared library in the address space of user space processes to avoid the interrupt-handling procedure. Hence, invoking those system calls results in a function call rather than a classic system call. These system calls are often not intercepted by tracing tools as they use a different interface. This is usually a minor problem because it mainly concerns time-related system calls. The implementation of these functions in the vDSO is architecture specific [17] For example, on ARM and MIPS architecture, only the following two functions are concerned.

```
1  int gettimeofday(struct timeval *restrict tv,
2                   struct timezone *restrict tz);
3  int clock_gettime(clockid_t clockid, struct timespec *tp);
```

Listing 4.4: Examples of system call prototypes that exchange user space data.

4.2.4 Memory forwarding

As previously discussed, certain system calls expect to access user space memory. Hence, a problem arises: which memory blocks must be forwarded together with the system call? In our research, we have considered seven approaches, each has drawbacks but remains relevant for particular situations, thus being complementary to each other. They can be grouped according to whether the instrumentation happens at *runtime* or is built into a *model* during an upstream analysis. Table 4.2 summarizes them.

	Name	Use case	Limitation
At runtime	Naive	Keep region synchronized	Migration time
	Ideal	Cat. B system calls	Identify accesses
	Dynamic memory tunneling	ioctl()	Heuristic
	Distributed shared memory	mmap regions	Trace accesses
Model-based	Record	Cat. B system calls	Record execution
	Selective symbolic execution	Cat. B system calls	Symbolize kernel
	Static analysis	Cat. B system calls	Disassemble binary

Table 4.2: Summary of memory forwarding approaches.

At runtime

I. The *naive* approach is to synchronize the complete process memory at the entry and exit of each system call. While this method does not require any knowledge of the type of system call, it is very inefficient as it introduces a significant delay. Indeed, the embedded system network throughput is

often limited to transferring all the process memory. In general, a system call does not interact with the whole process memory but only a region. It is often the case when a file is mapped in memory via `mmap()`.

II.   In contrast, the *ideal* approach would precisely identify and synchronize the memory regions modified by each system call. This is possible for many system calls because the size of the structure is known or the size of the buffer is located in another argument. However, the size of objects referred to by a parameter will not be known for custom system calls and `ioctl()` calls to custom character devices.

III.   To handle the limitations of the ideal method with `ioctl`, PROSPECT [97] proposed a novel mechanism called *Dynamic Memory Tunneling*. The idea consists of "always transferring a memory buffer to the target system if the IOCTL parameter is a possible pointer to a memory location". To identify if the argument is a pointer, the technique checks if the value corresponds to a valid address in the process address space and if the corresponding memory region has read and write permissions. The pages containing the address, as well as the surrounding ones, are synchronized. Upon the system call's return, the memory is compared to detect modifications, i.e., writes from the kernel. The number of bytes that were changed is used to decide if the pages need to be forwarded back. Heuristics are employed to identify the number of pages that need to be forwarded. The authors have determined empirically that three pages are usually sufficient in most cases, as structures are often located at the border of a page.

IV.   An alternative method consists of implementing a form of *distributed shared memory*. To accomplish this, a mechanism to intercept the memory writes and reads is needed. On Linux systems, it is possible to take advantage of the kernel page fault handling mechanism. This is what the `libsigsegv` library [7] and the `userfaultfd()` system call allows doing. However, page faults are not free: raising and catching them introduces overhead in the instrumentation execution.

Model-based

V.   The proposed method is carried out in two phases. Firstly, the process is traced under normal use during which system calls and user memory accesses are recorded. From these recordings, the data structure outlines are identified and associated with the system call arguments. Secondly, during emulation, the inferred rules are used to decide which blocks of memory need to be forwarded with the system call.

However, it should be noted that tracing system calls and recording their memory accesses on the original environment can be difficult. Additionally, the traces may not cover all the program execution paths of interest during emulation.

VI.   Alternatively, selective symbolic execution [46] may be used to identify the memory regions touched by a given system call. However, it requires the ability to symbolically execute the relevant kernel codes. For example, Mousse [105] shows promising results to apply selective symbolic execution on OS services and kernel routines.

VII.   As before, a model is built to determine which memory blocks should be forwarded during the system call. However, this method uses static analysis. The approach involves identifying the parts in the system call code that interact with user space memory and the associated data structures. As seen in table 4.1, a user space memory access API is often used. Rules can then be inferred from this information and used at runtime to synchronize the memory before and after the forwarding.

A similar approach has been demonstrated through DIFUZE [51]. Using source code, DIFUZE focuses on `ioctl` system calls to recover the device driver interface. In particular, it reconstructs the driver filename, the `ioctl` commands and the associated data structure exchanged with user space.

It should be possible to generalize this approach to binaries without source code access. In particular for Loadable Kernel Modules because they need to export their symbols to be properly linked with the kernel.

4.2.5   Process resources consistency and synchronization

It is essential to maintain a minimum level of synchronization between the emulator and the physical device while the program is executing in a distributed fashion. This is necessary to ensure that the program runs normally. This section discusses strategies to manage the process resources consistently. It is not a straightforward task because, as previously discussed in section 4.2.1, the resources are partially managed by the kernel. Although it is possible to instrument the kernel on the host [94, 104], we aim at not modifying the kernel running on the physical device. Moreover, this latter may not always be feasible in certain cases. Therefore, additional care must be taken in some cases when managing resources from user mode to ensure consistency in the program execution.

Process creation and identifiers

On the creation of a new process or thread using the `fork()` or `clone()` system call, the operation must be forwarded to the physical device. In addition, all identifiers reflecting the process hierarchy must be kept synchronized. This requires special attention as the choice of identifiers, in particular the PID, is rarely decided by the user process but imposed by the kernel. To address this, we chose to copy the identifiers chosen by the kernel on the remote device and rearrange them in the emulator. This approach is easier to implement as it provides more control over the emulator state.

On Linux, there are several methods available for setting the PID of a child process. One method is to repeatedly use the `fork()` system call until the desired PID is obtained. Another method, initially used by CRIU, utilizes the `/proc/sys/kernel/ns_last_pid` file to assign PIDs. It has the advantage of not requiring root privileges. From the kernel version 5.5, the `set_tid` array in the `clone3()` system call offers the ability to choose the PID of the child process. Moreover, these methods can be used within a PID namespace to prevent interferences from other processes on the host system.

File descriptors

When a file or device is opened, the created file descriptor is only valid within the kernel where it was generated. As a result, any actions taken on the file descriptor, such as reading, writing or other system calls (e.g., `mmap`, `lseek`, `ioctl`, `close`), must be performed on the same kernel. The filtering system allows us to modify these rules at runtime.

To reduce the number of context switches introduced by I/O operations, such as `read()` and `write()` system calls, a file can be mapped in the process memory via `mmap()`. Doing so allows the process to directly access the file content without asking the kernel. It is done transparently by the kernel. Similarly, `splice()` and `vmsplice()` system calls remap memory pages internally without a round trip to user space. They are used to transfer data to a pipe. Both present different challenges for memory synchronization between the process memory living in the emulator and the kernel managing the read and write accesses on the remote device.

For example, a file opened on the remote device is then mapped in the process memory. Besides allocating the same memory region locally in the emulator, the accesses must be forwarded otherwise the program will execute with wrong values. This is similar to a `splice()` call between a file and a pipe not living in the same kernel.

Inter-Process Communication

IPCs are managed by the kernel. Most of them use the system call interface: files, pipes, message queues, sockets and sending signals. Therefore, for most of them, no further special care needs to be taken. Because it is the kernel that handles the signal propagation and reception, they need to be intercepted on the device, returned, and injected into the emulator. Shared memory with other processes can be handled by intercepting the accesses and synchronizing them using a distributed shared memory mechanism between the emulator and the device. Environment variables are stored within the memory of the process. They can be synchronized at the initialization. Any environment variable change should be intercepted in the same way as any other memory region.

Remote events

So far, we have mainly discussed actions initiated by the user mode application. However, the kernel may also initiate operations on the process such as memory accesses or sending signals. In such cases, these actions are initiated by the remote kernel located on the device and need to be forwarded to the emulator.

As previously explained in the background in 2.2.6, asynchronous interfaces consist of two types of operations: a submission issued by the process and an event confirming the operation's status issued by the kernel to the process. The older asynchronous I/O interface handles both operations via system calls initiated by the process and can therefore be handled in a similar way to other system calls. Instead, the more recent io_uring interface uses two ring buffers shared between kernel and user space. They present a challenge because the kernel may read and write or modify pointers related to the ring buffers whenever it wants. Similarly, when a file located on the remote device is mapped in the process address space, accesses to this memory region have to be intercepted for proper synchronization.

Direct Memory Access (DMA) /citemera2021dice allows hardware devices to transfer data directly to and from the memory without involving the CPU. However, it may be difficult to intercept these accesses from user space because it is a more privileged operation.

In addition, kernel modules may also implement new mechanisms to access user space memory. In this case, it is highly dependent of what the developer intend to do and may even not follow the kernel guidance.

Finally, special architecture modes may also access the user space memory. This is typically the case for the System Management Mode (SMM) on Intel, ARM TrustZone or MIPS VZ (Virtualization Extensions). Althought

embedded systems are the main scope of this tool, these hardware extensions may not be present.

To monitor accesses to theses regions and propagate the event back to the tracer, one approach is to use a shared memory. However, it is necessary to identify in advance the correct memory regions. In practice, the kernel on the device may not support the correct capabilities to implement this shared memory, i.e., `userfaultfd` interface for intercepting the memory accesses. Furthermore, more privileged accesses may not be possible to intercept from the user space at all. In such cases, it may be necessary to use a shadow memory for these regions and a polling mechanism to monitor changes in their content, to later replicate them in the emulator.

4.3 System call forwarding

4.3.1 State machines

We propose a new approach able to trace and forward system calls from a running program. As we have seen in 3.3, each use case has different motivations for intercepting system calls at different abstraction layers. To avoid reinventing the wheel, we have decided to dissociate the *interception mechanisms* from the *tracing instrumentation* and focus on the latter. In Chapter 5, we describe how we have integrated our tracer in QEMU user mode emulation. Another characteristic that influenced our design choices is that the analysis has to be carried out by different components communicating: the emulator and the physical devices. To address this, we have created a distributed system composed of three main components: a tracer, an executor and an orchestrator. The approach is generic enough for POSIX systems, but we have focused on Linux in particular. Figure 4.4 describes our approach.

First, the tracer takes over at the next entry to or exit from a system call. On system call entry, the context is transferred to the orchestrator which filters according to rules set by the analyst. The decision made is sent back to the tracer which executes it. On system call exit, the saved decision is checked again before being carried out. The orchestrator operates over three different events: *SEND_SYSCALL_ENTRY*, *SEND_SYSCALL_EXIT* and *RETURN_SYSCALL_EXIT*. The first two are linked to the tracer while the last one is related to the executor. The orchestrator is responsible for instrumenting system calls and establishing connections between the tracer and the executor. Finally, the executor works in a loop, waiting for new system calls to execute and replying with their return values. The state machines for the three components are illustrated with Figures 4.5, 4.6 and 4.7.

A distributed system implies a mean for elements to communicate. For

this purpose, we have designed a packet-based protocol with a fixed-length header and a variable-length payload. More details on its implementation are described in Section 5.2.4.
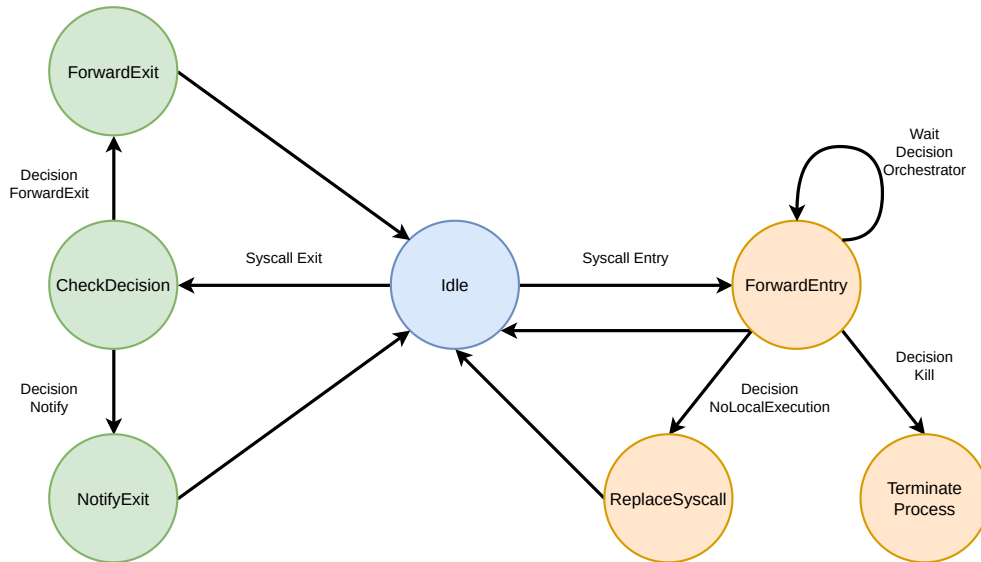

Figure 4.5: The *tracer* state machine.
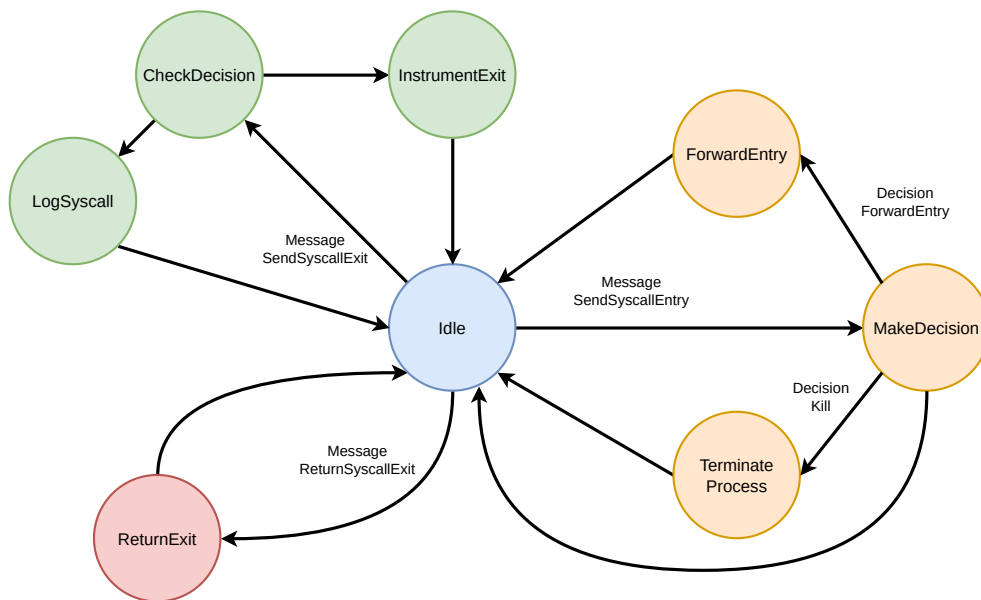
Figure 4.6: The *orchestrator* state machine.
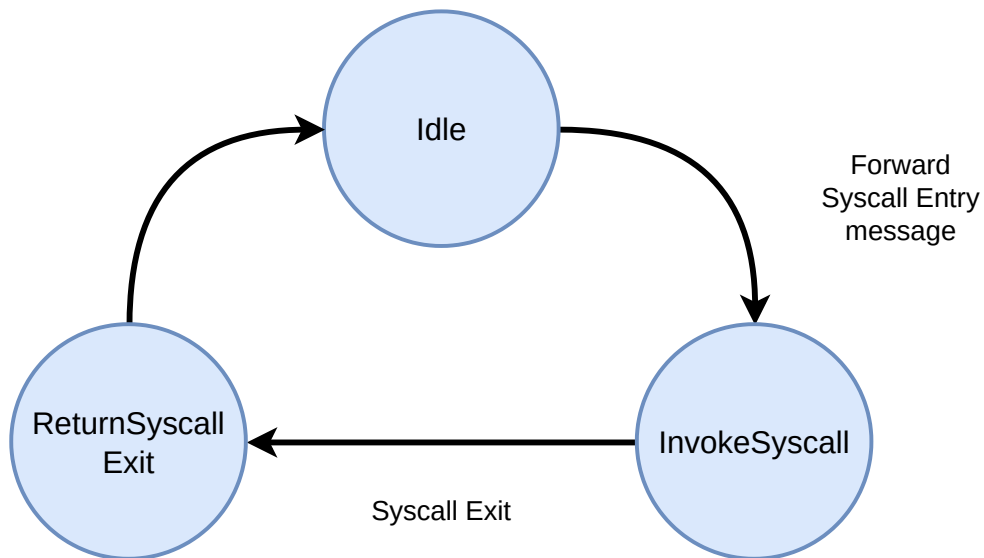
Figure 4.7: The *executor* state machine.

4.3.2 Filter & Rules

Because it is not efficient or desirable to forward all system calls to the remote kernel, we have implemented a new filtering mechanism. The filtering system allows for defining rules for system call interception in a relatively simple grammar, fully depicted in Listing 4.5.

Each filter consists of a set of rules and aims to return a decision based on the first matched rule. If no rule is matched, a default decision, which is often to execute the system call normally, is returned. A rule must at least define the system call to which it is applied and a decision. Optionally, a rule can be refined by adding a set of conditions that must be met by the system call's arguments before returning a match. For example on the MIPS architecture, the following rule forwards the `openat` system call when it opens any file with the name *filename* with read and write permissions:

```
4288: *a2=="filename" and a4==2 -> FWD_ENTRY|FWD_EXIT|NO_EXEC;
```

In addition, multiple conditions for a single rule can be concatenated with the "AND" keyword, leading to a logical conjunction of the individual conditions. While a single rule is not able to match multiple mutually exclusive statements, multiple rules for the same system call can be registered with different conditions to achieve the same goal. Effectively, this allows the creation of logical disjunct statements by combining multiple rules, while keeping the grammar and its parser simple.

```
1  digit     = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
2  digits    = digit , {digit}
3  argument  = "a1" | "a2" | "a3" | "a4" | "a5" | "a6";
4  parameter = ["*"], argument
5  operator  = "==" | "!=" | ">" | "<" | ">=" | "<=";
6  sys_no    = digits
7  decision  = "CONTINUE" | "FWD_ENTRY" | "FWD_EXIT" | "NOTIFY_EXIT" |
         "NO_EXEC" | "KILL" ;
8  decisions = decision , { "|" , decision };
9  condition = parameter , operator , ( digits | parameter );
10 conditions = "" | condition , { " and " , condition };
11 rule      = sys_no , ": " , conditions , " -> " decisions , ";" ;
12 filter    = {rule};
```

Listing 4.5: Filter grammar in eBNF.

4.3.3 Decisions

Once a rule of a filter is successfully matched, the corresponding decision has to be carried out. Six different decisions can be made, or a combination of them. We carefully selected these decisions to ensure it is expressive enough to cover all use cases. In detail, the decisions are:

1. CONTINUE: This is the default decision, it executes the system call without any further interception.

2. FWD_ENTRY: Execution will be transferred to a callback on system call entry and allows the modification of system call arguments. This can be used for system call instrumentation and forwarding.

3. FWD_EXIT: Similar to F_ENTRY, except that the callback is executed on system call exit and allows for modification of its return value.

4. NOTIFY_EXIT: Notify on system call exit of its return value without any callback.

5. NO_EXEC: Suppresses the execution of the system call, which results in an undefined return value being passed to the user space. This decision is typically selected in combination with F_EXIT to explicitly set a return value, for instance, to implement system call forwarding.

6. KILL: Terminate the process issuing the system call.

4.3.4 Execution order

When a program issues a system call, it may be executed on one side, the other side or both sides. It is crucial to consider the order of execution when executing the system call on both sides to keep both processes synchronized.

It is also necessary to determine whether the system call behavior needs to be replicated on the other side or if it can be executed independently.

To illustrate this concept, consider the example of allocating a memory region through the `mmap()` system call with the address argument set to `NULL`. On success, the kernel returns the starting address of the allocated pages. For instance, when tracing the `ls` program, the following system call is invoked:

`mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x77a78000`.

It informs the program that the kernel has allocated two pages of memory with read and write permissions located at the address `0x77a78000`. The question that arises is which kernel should decide the location of the new memory block. There may exist some dissimilarities between the kernel versions and configurations resulting in a different allocation algorithm. The Linux kernel implements different allocators such as SLOB, SLAB and SLUB. Additionally, other running programs may also influence the allocation via the allocator caches. Moreover, it is easier to instrument the kernel running on the host with the emulator than the device kernel. For these reasons, we chose to first invoke an anonymous `mmap()` on the device kernel, so that a second `mmap()` call is issued in the emulator with the address argument sets to the former's return value.

Other system calls do not always need to implement this ordering mechanism resulting from the instrumentation. Typically, system calls from category C (i.e., simple routine) can be replaced on the local kernel by a dummy call without having any impact on the execution flow. Both system calls can be performed in parallel with a synchronized mechanism on their termination to substitute the return value.

Chapter 5

Improving Linux-based firmware emulation with process snapshot and syscall forwarding

In this Chapter, we first describe our implementation of Chestburster before looking at its performances.

5.1 Design concept

Our approach is based on binary analysis rather than source code instrumentation. We have chosen this approach for several practical reasons related to the context of conducting security testing on embedded systems.

First, access to the source code is not always possible. Vendors often prefer to protect their software components behind proprietary licenses. Common examples in Linux-based systems include user-mode applications, binary kernel modules (e.g., GPU drivers) [110] and peripheral's firmware (e.g., WiFi or Bluetooth peripherals). Obtaining source code for firmware parts covered by free software licenses (e.g., under GPL) is not always straightforward. Although vendors are legally obliged to comply with source code redistribution when inquired, the process can be cumbersome in practice.

Second, recreating the toolchain and the building process for the target device can be challenging as it depends on many factors, such as the compiler, the kernel configuration, the kernel headers, and the patches.

Finally, hardware protections may prevent flashing the newly compiled firmware image on the device. For instance, the boot process may check the image's integrity before loading the firmware.

Our goal is to maintain the relevance and generality of our approach. To achieve this, we have chosen to implement our instrumentation at the POSIX interface without modifying the system itself. By leveraging the stable and public POSIX API, we believe we will be able to extend our analysis to Type-II firmware such as QNX-based firmware.

Given these considerations, practical analysis for real-world firmware often relies on binary firmware.

5.1.1 Process migration

Starting a dynamically linked program is made of two main steps. First, the kernel allocates the system resources and loads the program segments with its interpreter in memory. For an ELF file, this is the ELF interpreter. Next, the control flow is transferred to the entry point of the interpreter in user mode. The interpreter then loads the required shared libraries in memory, relocates objects and resolves the required symbols. All these steps are complex and lead to significant changes to the process memory layout, using system calls. Forwarding all these system calls to keep the emulated process synchronized with the one on the device would incur a significant execution overhead. Furthermore, the runtime loader of the emulator is significantly different the one from the kernel used on the device. As previously demonstrated [158] [31] [176] [130], such a discrepancy can lead to significant variations in the interpretation and significantly different memory layouts or symbol resolutions. This could for example lead to missing bugs both in the kernel runtime loader and the program.

Taking a snapshot of the process state, and allowing it to be transferred from the device to the emulator, helps to avoid those issues. However, snapshotting leads to another challenge: the restoration of the process, and in particular to identify which type of information needs to be captured and the how to save it.

We next explore possible techniques and tools for userland capturing the state and resources of a process running on a Linux-based system. In particular, we will discuss CRIU, Core dumps and GDBServer.

CRIU   The Checkpoint/Restore In Userspace project implements the process migration [48] functionality for Linux. It stops a process and all its children, capture their states to disk, and later, or in a different place, restores and restarts them. CRIU captures a lot of different system resources of a process: identifiers, sockets, files, pending signals, mount points, namespaces, cgroups, etc.

CRIU was initially created for snapshotting running containers. Unfortunately, its application to Linux-based embedded systems is problematic.

CRIU requires specific kernel features [49] such as `CONFIG_CHECKPOINT_RESTORE`, `CONFIG_NAMESPACES` and `CONFIG_UNIX_DIAG` to be configured at compile time. However, kernels found in embedded systems are often stripped of unnecessary options to conserve storage and memory usage minimal, making CRIU unable to run. Furthermore, CRIU does not provide the option to select which resources to capture, it either captures all resources or aborts. Therefore, without implementing this selection feature in the project, using CRIU on embedded systems can be difficult. For instance, we found that the namespaces and socket monitoring interfaces are rarely enabled. Both are essential options needed by CRIU during a checkpoint.

Core Dump A core dump is a file that captures the memory content and CPU context of a process at a specific time. Usually, it happens when the process receives a certain type of signal such as `SIGSEGV`, `SIGQUIT`, although the exact behavior depends on the kernel configuration. Core dumps are mainly used for debugging purposes in the context of post-crash analysis. However, it should be noted that core dumps may provide incomplete information. Other resources such as file descriptors, identifiers, and mount points are not included. Moreover, the file `/proc/[PID]/coredump_filter` controls which memory segments are written to the core dump.

The Extended Core File Snapshot (ECFS) project [69] enhances the core file format with additional features to examine processes in the context of memory forensic analysis. After generating the core dump file, the process `procfs` is inspected to extract complete information about it. A utility is available to continue the process execution from its snapshot. However, the support is mainly limited to the x86 architectures.

GDBserver A simpler alternative consists of using a debugger such as `gdbserver` for remotely starting or attaching to a process. This approach allows the inspection of various process resources, which can then be easily copied into the emulator.

5.1.2   The Chestburster Architecture

We now present our solution Chestburster which aims at providing snapshot and restore for Linux-based embedded devices.

The architecture of Chestburster is presented in figure 5.1. It consists of three main components: the *executor* on the physical device, the *tracer* inside an emulator, and the *orchestrator* managing both. This design centralizes the main features in the orchestrator while targets implement basic operations. As a result, this design provides the possibility to smoothly manage targets

and register callbacks for various instrumentation at different stages of the analysis.

The executor uses a statically compiled `gdbserver` to create process snapshots and execute the system calls remotely. We have developed a new library `sysforward` to trace system calls and communicate with the orchestrator in different execution environments. We have integrated this library into the QEMU user-mode emulator and extended it to load process snapshots. The avatar² framework [123] manages both the emulator and the physical device. Because avatar² mainly focused on monolithic firmware (Type-III), we have extended it to include support for processes, system call filtering and forwarding between targets as described in Secion 4.3.
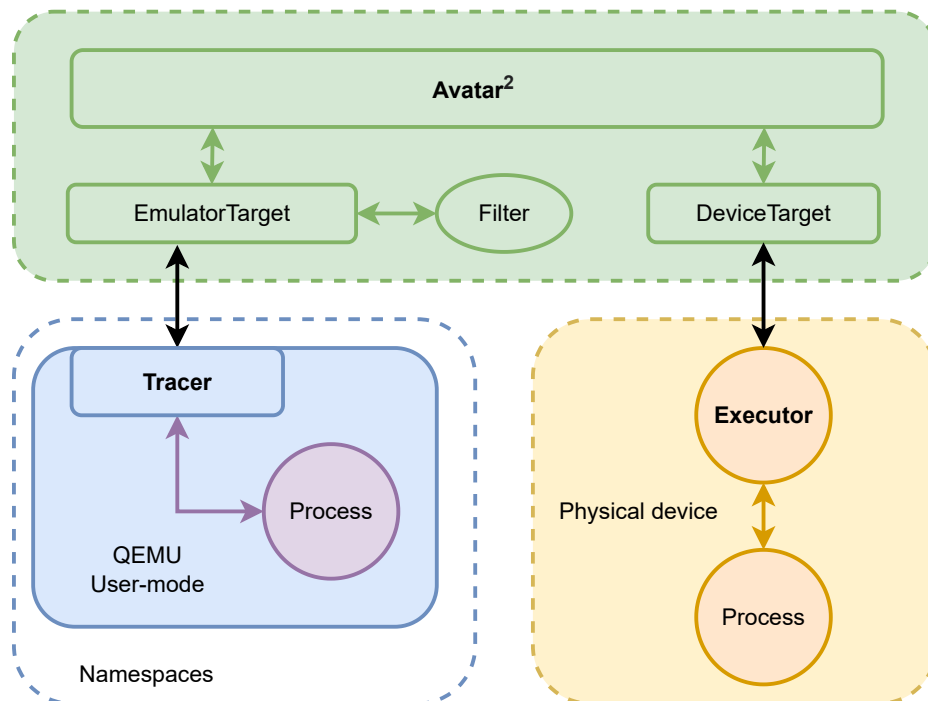
Figure 5.1: Chestburster architecture.

5.1.3 Analysis workflow

The analysis is done in two phases. First, the program is loaded on the physical device and the process context and memory are captured and transferred to the emulator. Then, the execution resumes and system calls are filtered to either be performed locally or forwarded to the device.

Process migration

The program is first loaded on the device and then migrated to the emulator for several reasons. The use of the kernel runtime loader on the embedded system allows the memory layout to be as close as possible to its original environment. Moreover, as explained in Section 5.1.1, loading various libraries forces extensive instrumentation of symbol resolutions and memory mapping operations. Additionally, the emulator provides better control over the process and its environment, making the restoration of the process more straightforward. Nonetheless, Chestburster is flexible enough to support the opposite scenario to start the process in the emulator and migrate a copy to the device.

The gdbserver approach was selected due to its simplicity and versatility. It is the approach that requires the least amount of additional instrumentation in comparison to other alternatives which comes with significant challenges. CRIU needs special kernel configurations to be enabled and does not allow for partial process snapshots. ECFS is limited to x86 architectures. The core dump approach requires implementing a suitable loader for the QEMU emulator.

The choice to stop the execution of the process after its initialization by the loader, but prior to starting (i.e., the beginning of main() function), simplifies the migration process by only requiring the capture of the memory mapping and CPU context. This approach eliminates the difficulties associated with the forwarding of dynamic library loading and the migration of complex resources, such as network connections, file locks, and devices.

In summary, Chestburster starts a gdbserver and the program to analyze on the physical device. A breakpoint is set after the loading but early in the life of the process. For example, a breakpoint near the main function is preferable. From there, the process state is captured via the /proc/[PID]/maps and /proc/[PID]/stat files and its memory content is dumped into an ELF file. This file is then transferred to the host in namespaces that replicate the device environment. The QEMU runtime loader restores the process memory layout and its gdbstub is used to copy the saved CPU context. The process execution can now be resumed.

System call forwarding

All system calls are intercepted on their entry. This interception happens inside QEMU, before QEMU's system call instrumentation. The system call number and its arguments are passed as raw values to the tracer in the sysforward library, which then sends them to Chestburster. It handles syscall parameter decoding by assigning a type to each argument and dereferencing the potential pointers. This facilitates the application of user-defined filters

to the system call. The filter outputs a decision on the action to carry out. The decision could be executing the system call locally on the host, remotely on the device, intercepting the system call exit or terminating the process. The choice of action is left to the discretion of the user. Figure 5.2 illustrates on a sequence diagram the flow of events for a typical forwarding decision.

For example, a typical forwarding strategy can be implemented with the following decision `NO_EXEC|FWD_ENTRY|FWD_EXIT`. The `NO_EXEC` decision means the system call should not be executed by the emulator. This is achieved by replacing it with a dummy call such as `sys_ni_syscall()` to keep the opportunity to intercept its exits without causing any side effects on the kernel. The `FWD_ENTRY` decision copies the arguments to the remote process on the device, executes the system call there, saves the return value and system error number and then returns to Chestburster. Finally, `FWD_EXIT` decision ensures that both local and remote system calls are synchronized on exit.

5.2   Implementation

To demonstrate our system call forwarding approach, we have implemented a prototype *Chestburster* targeting Linux-based systems on the MIPS 32 bits big-endian architecture. This platform was selected because of its widespread use in Linux embedded systems [44], including our case study, the Philips Hue Bridge. The implementation is generic enough for allowing future extensions to other architectures and POSIX systems in the future.

5.2.1   Enhancing avatar² for Linux processes

Chestburster is based on the *avatar²* framework [123]. It acts as the central component of our system and is written in Python for rapid prototyping. Chestburster can generate and migrate process snapshots, as well as trace, filter and forward system calls between various analysis tools. To achieve this, we have extended the avatar² messaging system to handle new events related to system calls such as `SyscallEntry`, `SyscallExit`, `SyscallReturnExit`, `SyscallForwardEntry` and `NewProcess`. Additionally, we have implemented new abstractions to better represent the process and its memory space, as well as to facilitate the instrumentation of system calls. These instrumentations include argument decoding, filtering and synchronization mechanisms for system calls which modify the process layout (category *A* from Section 4.2.3). Filters are implemented as dictionaries with the key corresponding to the system call number and the value storing the rules applied on this system call.
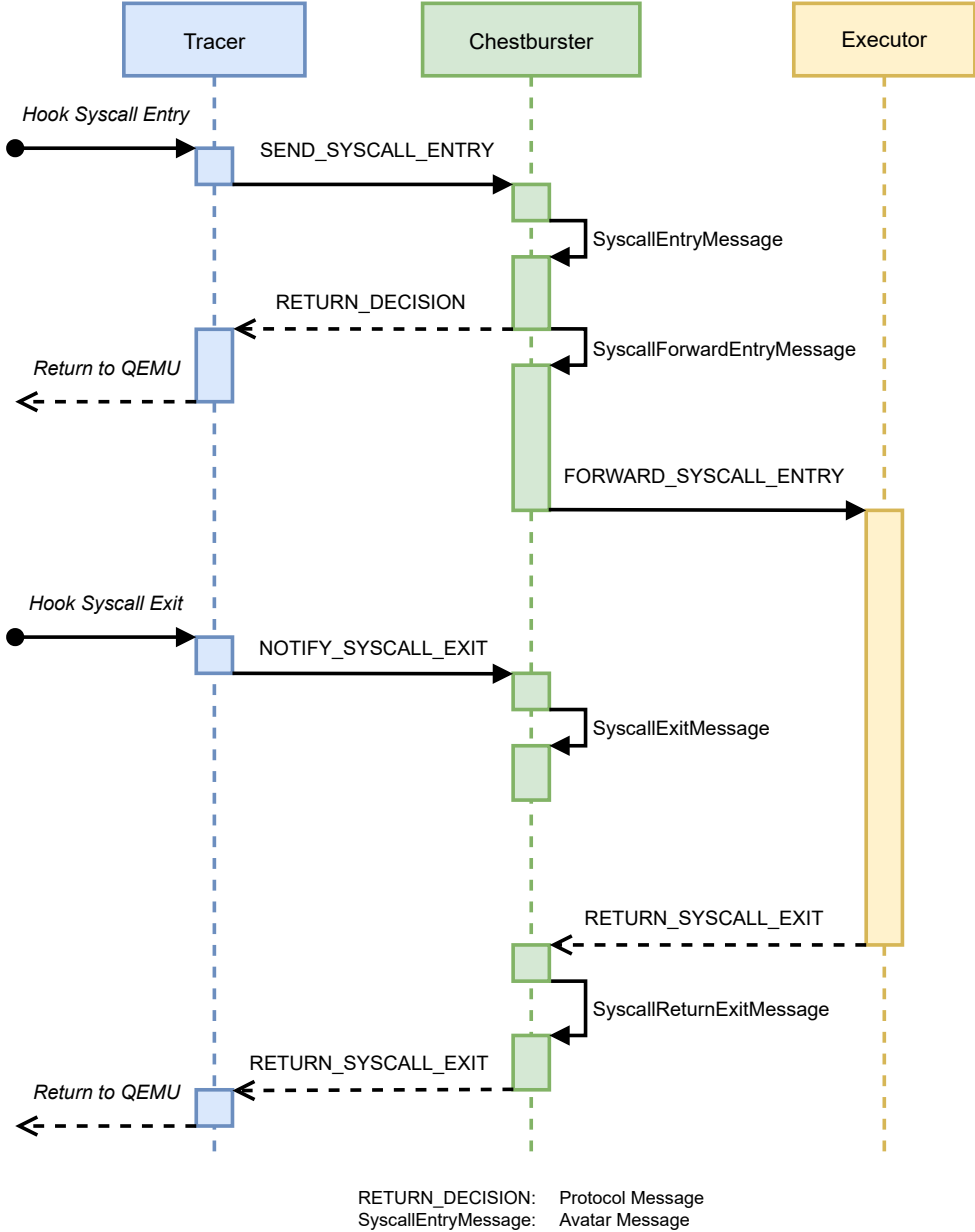
Figure 5.2: Sequence diagram illustrating the forwarded of a system call during parallel execution.

5.2.2 Process migration

We rely on the gdbserver approach to migrate the process running on the physical device to the emulator on the host. We set a breakpoint at the main() function to allow the program to load properly on the device. Once the breakpoint is reached, the memory mapping and the state of the process are collected through examination of the /proc/[PID]/maps and /proc/[PID]/stats files. The memory regions are then stored in an ELF file using segments. As a result, it can be loaded in the QEMU emulator like any program. Finally, the CPU context is transferred using GDB to resume the execution in the emulator.

5.2.3 The QEMU user-mode based tracer

We have implemented the tracer component of the state machine presented in Chapter 4 Figure 4.5 as a C library named sysforward. We have made this decision to be able to dissociate the *interception mechanisms* from the *tracing instrumentation*. It allows it to be integrated with multiple analysis tools. As a result, the library can be used in various contexts such as a standalone tracer based on ptrace, or enhance an emulator like QEMU user-mode.

The libsysforward consists of a tracing thread for each thread being traced, and a listening thread responsible for communicating with the orchestrator. This design enables multiplexing tracing communications over a single communication link. Furthermore, it implements the forwarding protocol as described in 5.2.4.

In particular, we have modified the QEMU user-mode emulator to include the sysforward library. We chose this emulator because we are interested in the execution of the user process part and not the whole operating system. The kernel part of the execution is performed on the physical device. In addition, the execution overhead is reduced because no hardware needs to be emulated. It also removes the burden to rehost all the OS services and their hardware dependencies, similar to what Firmadyne [44] tries to achieve.

The process is resumed in namespaces to separate the analysis environment from the rest of the host. The systemd-nspawn command allows starting a command in a lightweight container. It combines the classic Mount namespace from chroot with others such as PID, User, Cgroup, IPC, Time, UTS and Network.

The process PID is restored through the combination of the PID namespace with the set_tid parameter of the clone() system call.

5.2.4 Protocol

The communication protocol corresponds to the implementation of the state machines from figures 4.5, 4.6, 4.7. Each packet consists of a header with a fixed size and a payload of variable length. For our prototype, we have implemented the protocol on top of TCP, but it could be modified to work on top of any other communication protocol such as UDP.

The header is composed of three main fields:

1. The command.

2. The PID to multiplex multiples process tracings over a single connection.

3. The length of the payload associated with the header.

The payload is dependent on the command. We define 16 main commands:

- `Error` to indicate a failure.

- `NotifyNewProcess` to notify the orchestrator a new process is traced.

- `SendSyscallEntry` to send the system call number and its arguments.

- `NotifySyscallExit` to notify the results of a system call.

- `ReturnSyscallExit` to return to the tracer what should the system call return value should be.

- `ReturnDecision` to return the tracer the decision made by the filer.

- `NotifySignal` is used to inject signals.

- `ReadArgs`, `WriteArgNo` and `WriteArgs` to operates on the system call arguments.

- `ReadRegs`, `WriteRegNo` and `WriteRegs` to interact with CPU registers.

- `ReadMemory`, `ReadString` and `WriteMemory` to interact with memory content.

5.2.5 Executor

In order to execute the system calls on the device, we take advantage of the process attached to the `gdbserver` from the process migration. GDB is used to execute each system call in the following manner: the memory and the CPU context are copied, the system call is executed, its result is saved and the context is restored.

The `clone` system call has several options for creating new processes. Currently, Chestburster only supports creating new threads, using the `CLONE_THREAD`

flag. This is because the `libsysforward` has been designed to handle multiple threads only, because of its communication link with the orchestrator. In addition, GDB does not permit debugging multiple processes simultaneously, it only supports multithreading debugging. However, it is still possible to catch new process creation for `fork` and `clone` and initiate them under a new debugger and emulator instances. Moreover, `execve` requires starting a new process migration from the device to avoid the complicated task of process loading.

The `brk` system call either allocates or deallocates memory by adjusting the program *break* which represents the data segment size of the process. Chestburster first executes the system call on the device and retrieves its return value. It then compares it with the program break present in the emulator to determine if `brk` should be emulated with a `mmap` or `munmmap` system call in the emulator.

5.2.6 Limitations

At the time of writing, the Chestburster prototype has the following limitations:

- As explained above, it does not support directly the creation of new processes. However, this can be addressed with additional instrumentations.

- Signals are not yet supported, but they do not pose a significant challenge. They can be caught and injected by the debuggers.

- Shared memories and memory-mapped files accesses are not intercepted on the device. Therefore, events initiated on the device are not correctly forwarded to the emulator. This can be improved in the future using distributed shared memory.

5.3 Evaluation

In order to evaluate Chestburster, we performed two sets of experiments:

1. We look at the correctness of our approach, and in particular if our instrumentation influences the execution flow.

2. We analyze the system performance regarding execution time overhead over synthetic benchmarks.

The experiments are run on a laptop with an Intel i7-8650U CPU (8 cores) with a clock of 4.2GHz and 32GB of RAM. The embedded system used in our experiments runs the HUE Bridge 2.X firmware version 1935074050 with OpenWRT on a Linux kernel 4.4.60 compiled with GCC 4.8.3. The hardware is composed of the QCA4531 System-on-Chip with a MIPS 24Kc CPU at 650 MHz and 64 MB of RAM. We initially started with the Lima [1] development board, but for the evaluation we switched to the Philips Hue Bridge board where we reconstructed a compatible toolchain.

5.3.1 Execution correctness

Our evaluation starts by comparing the system call traces of the forwarded execution with the native execution on the physical device. Although it does not guarantee that the program under test follows exactly the same execution path as it does on the device, it does confirm the program similarly interacts with the environment through its boundaries, e.g., the system call interface. Moreover, the choice to compare execution trace with system call granularity is justified by the location of our instrumentation.

To trace the program's system calls on the device, we have used a statically compiled `strace`.

Paxtest suite

The *paxtest* suite [140] was designed to simulate exploits and test kernel security features work as intended. PaX is a security patch for the Linux kernel which implements mechanisms to prevent arbitrary read/write access from an exploit. The test suite is composed of two sets of tests. The first set of programs attempts various approaches to writing and running exploit code. The second set measures the system's randomization. We are interested in the first set because the programs test different memory layouts and permissions using corner cases. Listing 5.1 illustrates one such program: the `execstack`. Running them by forwarding the system calls to the device's kernel assures us these corner cases are correctly handled.

---

[1] https://www.8devices.com/products/lima

```
1  /* execstack.c - Tests whether code on the stack can be
      executed
2   *
3   * Copyright (c)2003 by Peter Busser <peter@adamantix.org>
4   * This file has been released under the GNU Public
      Licence version 2 or later
5   */
6
7  #include <stdio.h>
8  #include <string.h>
9  #include <stdlib.h>
10 #include "body.h"
11 #include "shellcode.h"
12
13 const char testname[] = "Executable data
                               ";
14
15 char buf[MAX_SHELLCODE_LEN] = { 'A' };
16
17 void doit( void )
18 {
19     fptr func;
20
21     copy_shellcode(buf);
22
23     /* Convert the pointer to a function pointer */
24     func = (fptr)&buf;
25
26     /* Call the code in the buffer */
27     func();
28
29     /* It worked when the function returns */
30     itworked();
31 }
```

Listing 5.1: execstack.c source code.

The programs represent combinations of scenarios running in various memory areas. The memory areas are anonymous mapping, data, heap and stack. The first scenario imitates a buffer exploit: write data to the memory area and try to execute it. The second scenario tries to disable the memory protection using mprotect() before imitating a buffer exploit. The third scenario tries to overwrite code in the text segments. The fourth scenario simulates return-to-function attacks using (1) strcpy and (2) memcpy. Both attacks are performed with RANDEXEC enabled and disabled. The fifth scenario checks shared library BSS (Block Started by Symbol) and data areas with and without similarly disabling memory protection as before.

We successfully ran the first four scenarios in a way similar to how they

Scenario	File name	Number of Identical / Different System Calls	Number of Identical / Different Memory Mapping	Test Output	Result
1st	anonmap.c	2 / 0	26 / 0	Killed	✓
	execbss.c	1 / 0	26 / 0	Killed	✓
	execdata.c	1 / 0	26 / 0	Killed	✓
	execheap.c	3 / 0	26 / 0	Killed	✓
	execstack.c	1 / 0	26 / 0	Killed	✓
2nd	mprotanon.c	5 / 0	26 / 0	Vulnerable	✓
	mprotbss.c	5 / 0	26 / 0	Vulnerable	✓
	mprotdata.c	5 / 0	26 / 0	Vulnerable	✓
	mprotheap.c	7 / 0	26 / 0	Vulnerable	✓
	mprotstack.c	4 / 0	26 / 0	Vulnerable	✓
3rd	writetext.c	5 / 0	26 / 0	Vulnerable	✓
4th	rettofunc1.c	3 / 0	26 / 0	NULL	✓
	rettofunc2.c	2 / 0	26 / 0	NULL	✓
	rettofunc1x.c	2 / 0	26 / 0	∅	✓
	rettofunc2x.c	2 / 0	26 / 0	∅	✓
5th	mprotshbss.c	14 / 22	29 / 4	*BUG*	✗
	mprotshdata.c	14 / 22	29 / 4	*BUG*	✗

Table 5.1: The programs from the paxtest suite are used to verify the execution correctness.

are executed on the device. However, the fifth scenario, which involved the mapping of shared libraries, encountered a bug during the forwarding that prevented their correct loading. Like with the initialization step, a way around this problem is to migrate the process on the device for the loading.

Table 5.1 presents the number of system calls that are identical between the execution on the device and in the emulator with forwarding. Furthermore, we manually inspected the memory mapping to ensure they were also identical. The column test result displays the output of the tests, identical on the device and during emulation. It confirms that the process migration was executed correctly and that the various memory mapping operations were carried out correctly.

Linux kernel selftests

The Linux *selftests* suite is composed of many tests that focus on specific code paths in the kernel. The *nolibc tests* checks system calls issued by the eponymous kernel *libc*. It focuses on implementing a minimal C library to reduce the size of binaries used by the kernel such as `mkinitramfs` [171].

These tests were compiled using *uClibc*, which is part of the toolchain compatible with our Linux firmware. Out of 66 *nolibc* tests covering 31 system calls, 3 failed because of a return address corruption during the forwarding. Another test fails (waitpid_min) but not because of our instrumentation. It fails in the same way on the device, therefore we consider it as a correct be-

havior. The tested system calls includes `getpid`, `getppid`, `getpgid`, `kill`, `sbrk`, `brk`, `chdir`, `chmod`, `chown`, `chroot`, `close`, `dup`, `dup2`, `dup3`, `execve`, `gettimeofday`, `ioctl`, `link`, `lseek`, `mkdir`, `open`, `poll`, `read`, `sched_yield`, `select`, `stat`, `symlink`, `unlink`, `wait`, `waitpid`, `write`. Table 5.2 displays the arguments used with the system calls and their corresponding return values.

5.3.2   Execution overhead

We have so far evaluated the correct execution of Chestburster, however, forwarding system calls to a different device have a significant performance impact.

To evaluate the runtime overhead, we have used multiple scenarios to determine its origins.

- Scenario **Baseline** runs the program with plain QEMU user-mode version 7.2.0.

- Scenario **Interception** runs the program with our fork of QEMU which integrate *libsysforward*.

- Scenario **Strace** enables the option `-strace` in plain QEMU which prints the system calls arguments and returns a value between the system call transitions.

- Scenario **Interception-Strace** runs the binary with the option `-strace` with our QEMU fork.

- In scenario **Tracing**, the program is run with Chestburster where all decisions are to trace and continue the execution in the emulator.

- Instead, scenario **Forwarding** forwards all system calls to the physical device where they are executed.

Both scenarios *Interception* and *Interception-Strace* do not send information to the orchestrator. Scenario *Forwarding* represents a worst-case scenario where all system calls are forwarded. In its usage, the filtering mechanism helps to decide if the system call should be executed locally or remotely.

All presented values have been normalized with respect to scenario *Baseline*. Moreover, each scenario was conducted in a dedicated namespaces using `systemd-nspawn`. Finally, we disabled the CPU hyperthreading to minimize variations in all measurements.

No	System Call	Output	Result
0	getpid()	6990	✓
1	getppid()	6374	✓
5	getpgid(0)	6990	✓
6	getpgid(-1)	-1 ESRCH	✓
7	kill(getpid(), 0)	0	✓
8	kill(getpid(), 0)	0	✓
9	kill(INT_MAX, 0)	-1 ESRCH	✓
10	brk(4096)	0	✓
11	brk(sbrk(0))	0	✓
12	chdir("/")	0	✓
13	chdir(".")	0	✓
14	chdir("/blah")	-1 ENOENT	✓
15	chmod("/proc/self/net", 0555)	0	✓
16	chmod("/proc/self", 0555)	-1 EPERM	✓
17	chown("/proc/self", 0, 0)	-1 EPERM	✓
18	chroot("/")	0	✓
19	chroot("/proc/self/blah")	-1 ENOENT	✓
20	chroot("/proc/self/exe")	-1 ENOTDIR	✓
21	close(-1)	-1 EBADF	✓
22	close(dup(0))	0	✓
23	dup(0)	3	✓
24	dup(-1)	-1 EBADF	✓
25	dup2(0, 100)	100	✓
26	dup2(-1, 100)	-1 EBADF	✓
27	dup3(0, 100, 0)	100	✓
28	dup3(-1, 100, 0)	-1 EBADF	✓
29	execve("/", (char*[]) [0] = "/", [1] = NULL , NULL)	-1 EACCES	✓
32	gettimeofday(NULL, NULL)	0	✓
38	ioctl(0, TIOCINQ, &tmp)	0	✓
39	ioctl(0, TIOCINQ, &tmp)	0	✓
40	link("/", "/")	-1 EEXIST	✓
41	link("/proc/self/blah", "/blah")	-1 ENOENT	✓
42	link("/", "/blah")	-1 EPERM	✓
43	link("/proc/self/net", "/blah")	-1 EXDEV	✓
44	lseek(-1, 0, SEEK_SET)	-1 EBADF	✓
45	lseek(0, 0, SEEK_SET)	-1 ESPIPE	✓
46	mkdir("/", 0755)	-1 EEXIST	✓
47	open("/dev/null", 0)	3	✓
48	open("/proc/self/blah", 0)	-1 ENOENT	✓
49	poll(NULL, 0, 0)	0	✓
50	poll(&fds, 1, 0)	*BUG*	✗
51	poll((void *)1, 1, 0)	-1 EFAULT	✓
52	read(-1, &tmp, 1)	-1 EBADF	✓
53	sched_yield()	0	✓
54	select(0, NULL, NULL, NULL, &tv)	*BUG*	✗
55	select(2, NULL, &fds, NULL, NULL)	*BUG*	✗
56	select(1, (void *)1, NULL, NULL, 0)	-1 EFAULT	✓
57	stat("/proc/self/blah", &stat_buf)	-1 ENOENT	✓
58	stat(NULL, &stat_buf)	-1 EFAULT	✓
59	symlink("/", "/")	-1 EEXIST	✓
60	unlink("/")	-1 EISDIR	✓
61	unlink("/proc/self/blah")	-1 ENOENT	✓
62	wait(&tmp)	-1 ECHILD	✓
63	waitpid(INT_MIN, &tmp, WNOHANG)	-1 ECHILD != (-1 ESRCH)	✓
64	waitpid(getpid(), &tmp, WNOHANG)	-1 ECHILD	✓
65	write(-1, &tmp, 1)	-1 EBADF	✓
66	write(1, &tmp, 0)	0	✓

Table 5.2: The programs from the paxtest suite are used to verify the execution correctness.
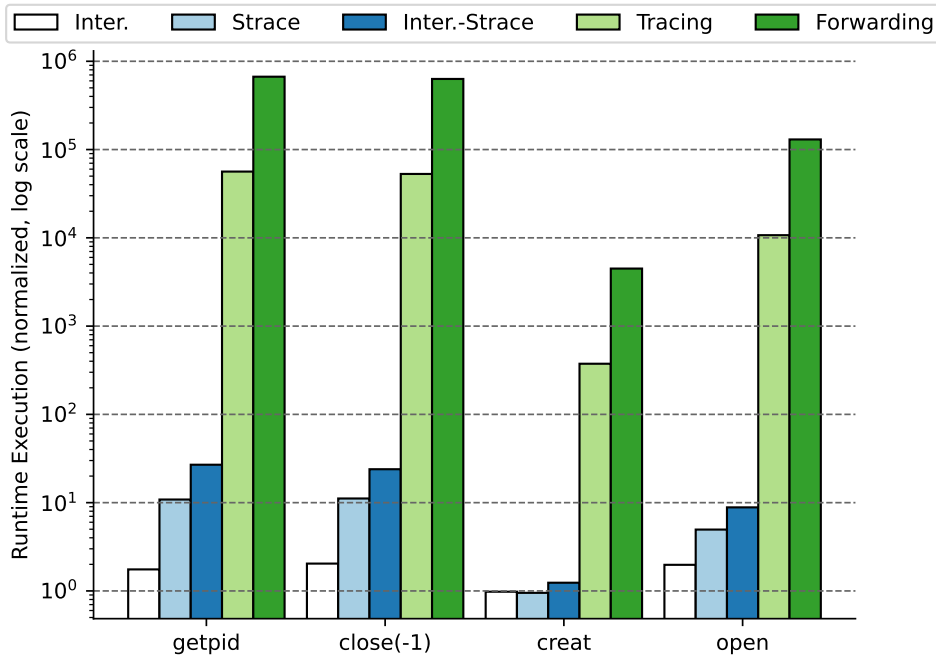
Figure 5.3: Impact of Chestburster on common system calls (a).

System call microbenchmarks

To measure the runtime overhead introduced by the different elements of
Chestburster, we ran microbenchmarks on various system calls. In particu-
lar, we measured the time from the perspective of the user mode process.

The methodology consists in executing each system call 1000 times in a
loop. The monotonic time is measured at the entrance and exit of the loop
using the `clock_gettime()` system call with the `CLOCK_MONOTONIC` clock. The
buffer for `read` and `write` was set to 256 bytes. The measured values are
then normalized by dividing them with the corresponding ones from sce-
nario *Baseline*, which involve the execution in plain emulation with QEMU.
Both `getpid` and `close(-1)` represent system calls that return rapidly in user
space. The system calls `creat`, `open`, `write`, `read`, `lseek`, `close` and `unlink` are
representatives of common I/O operations in Linux.

Figures 5.3 and 5.4 present the normalized runtime overhead while fig-
ure 5.5 highlights the composition of the overhead. We can see that the *For-
warding* scenario adds an extra order of magnitude compared to the *Tracing*
scenario which itself represents 3 orders of magnitude compared to the *Strace*
scenario. In contrast, the native QEMU system call logging adds an overhead
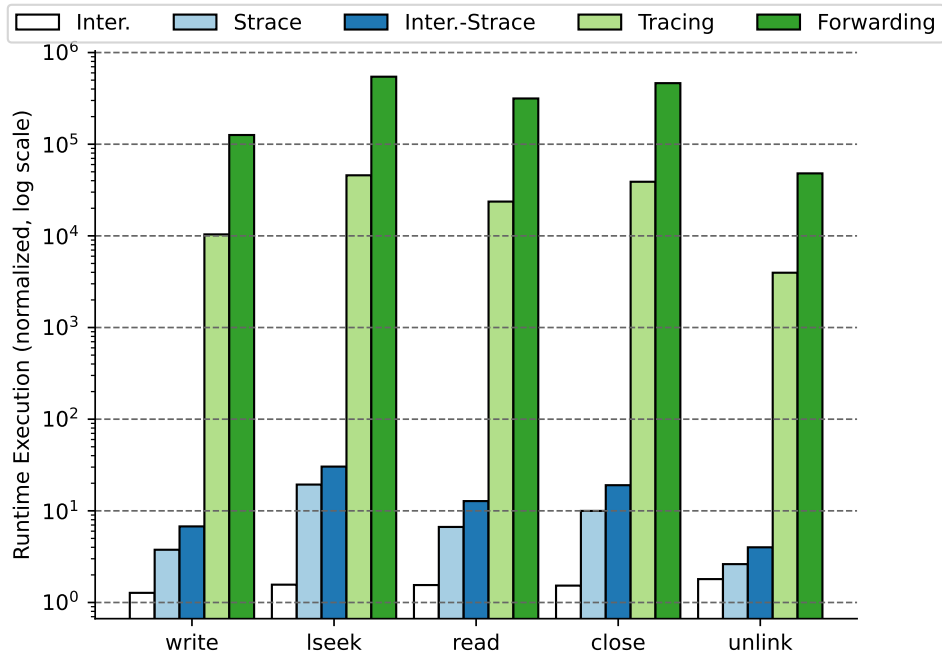of a factor of 10.

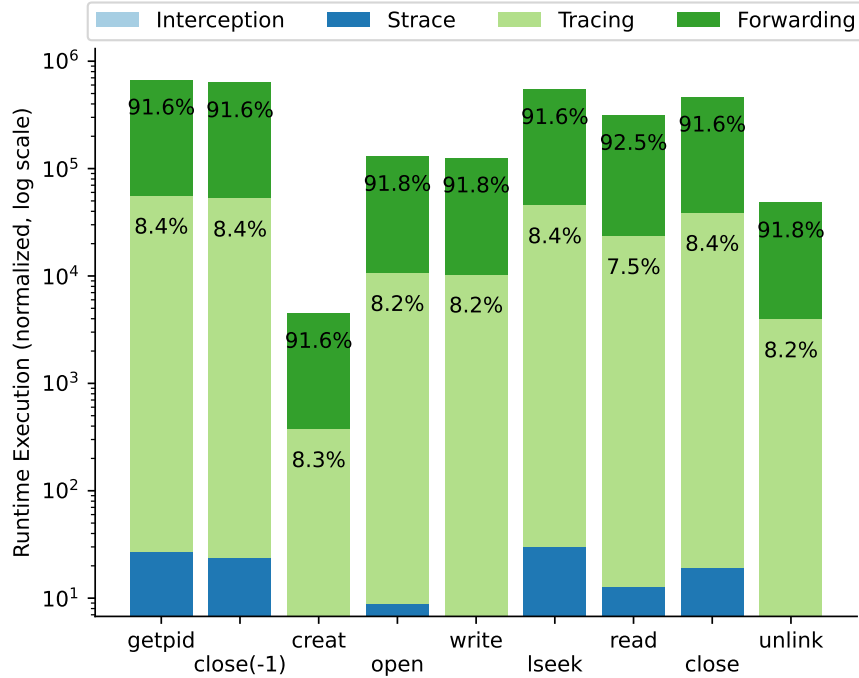**Figure 5.4:** Impact of Chestburster on common system calls (b).

Figure 5.5: Composition of overhead for common system calls.

Program	Total Forwarded System Call Number	open	close	read	write	ioctl	brk
cat	14	1	3	2	1	0	3
csplit	118	13	15	2	23	12	6
wc	20	1	3	2	9	0	2
sha512sum	87	1	3	2	73	1	4
dd	39	2	5	3	16	0	4

Table 5.3: Principal system calls issued by evaluated coreutils programs.

Coreutils

Coreutils programs are essential components in any embedded Linux-based system. They form a central part of shell scripts that are used for initializing, updating, and maintaining these systems, in particular for routers.

Using Chestburster, we managed to run 56 out of the 110 binaries from *GNU coreutils*. The programs that run focus on operations on file content, directory listing, file statistics, conditions, file name manipulation, user information, system and working contexts. The 56 programs are cat, tac, nl, od, base32, base64, basenc, fmt, pr, fold, head, tail, split, csplit, wc, sum, cksum, b2sum, md5sum, sha1sum, sha224sum, sha256sum, sha384sum, sha512sum, sort, uniq, comm, ptx, tsort, cut, ls, dir, dircolors, dd, stat, sync, echo, true, false, test, expr, basename, dirname, pathchk, realpath, stty, printenv, id, logname, whoami, groups, who, date, hostid, nice and factor.

We conducted a performance evaluation on five coreutils programs: cat, csplit, wc, sha512sum, and dd. To obtain accurate results, we calculated the arithmetic mean of five runs, along with an additional warm-up execution. Table 5.3 displays the total number of system calls and the most significant ones related to the filesystem. Figures 5.6, 5.7, 5.8, 5.9 and 5.10 provide a breakdown of the overhead composition when taking as input a file of 0, 128, 1024, 65k and 1M random bytes. We did not include the 1MB case for dd because the program execution ends normally before copying all the bytes. We did not identified the root cause of the issue. No timeout is present and the trace only contains a succession of read and write system calls. However, in the experiment environment we observed the bs option has an influence on the number of bytes copied before exiting. Switching from 512 to 1k makes dd ends from 130kB to 930kB.

The figures show that the majority of the overhead is caused by the forwarding which also adds an extra order of magnitude compared to the *Tracing* scenario. Moreover, we do not observe any strong correlation between the file size and the runtime execution. This may be explained by the fact, some of the programs copied a buffer with a fixed length for transfering the memory with the kernel.
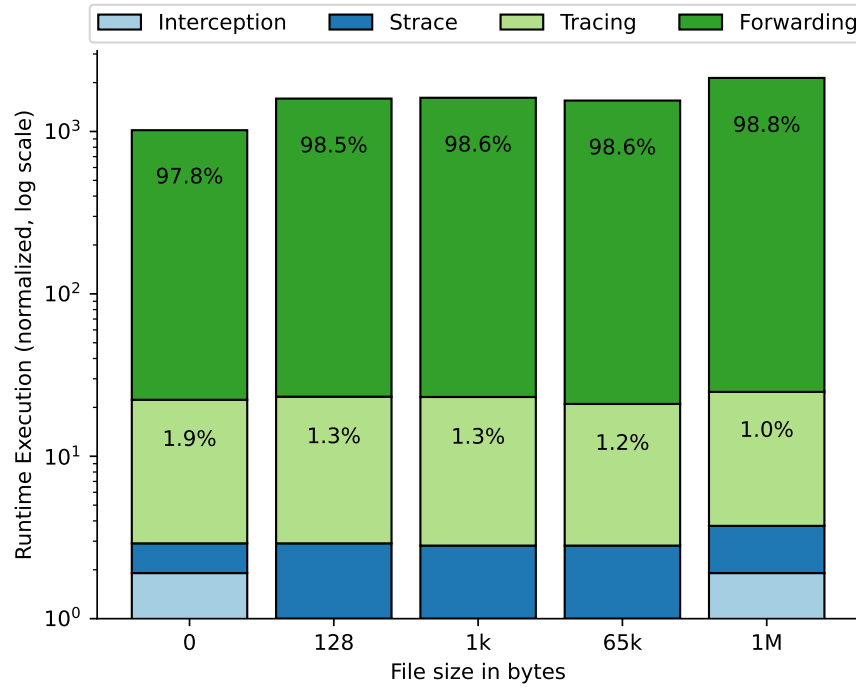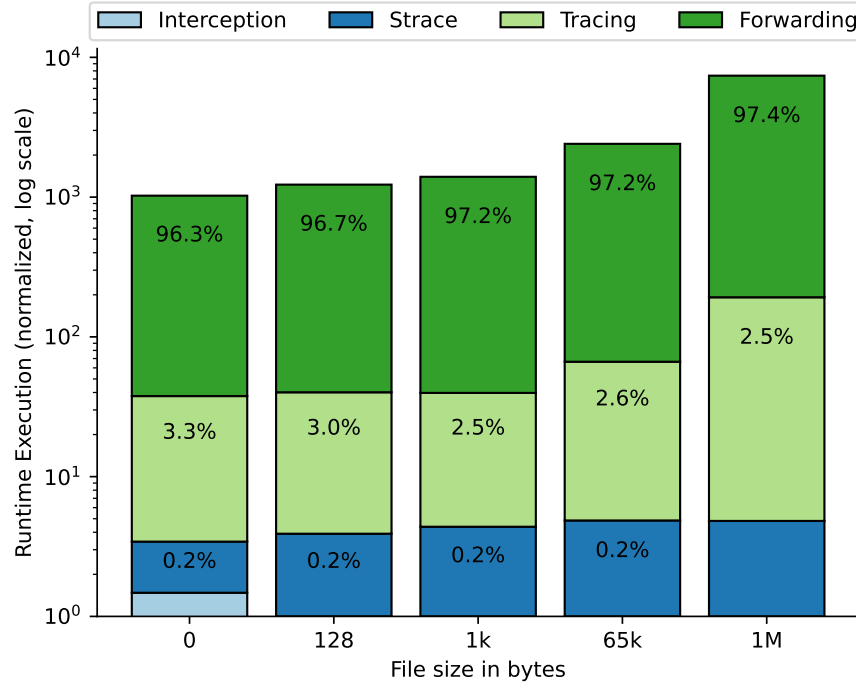
Figure 5.6: Composition of overhead for cat.

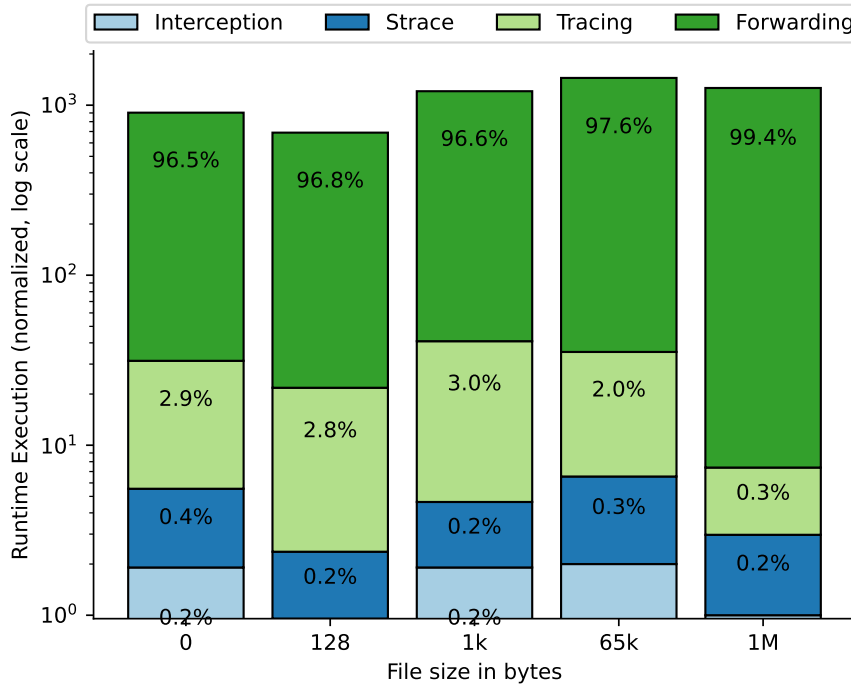**Figure 5.7:** Composition of overhead for csplit.

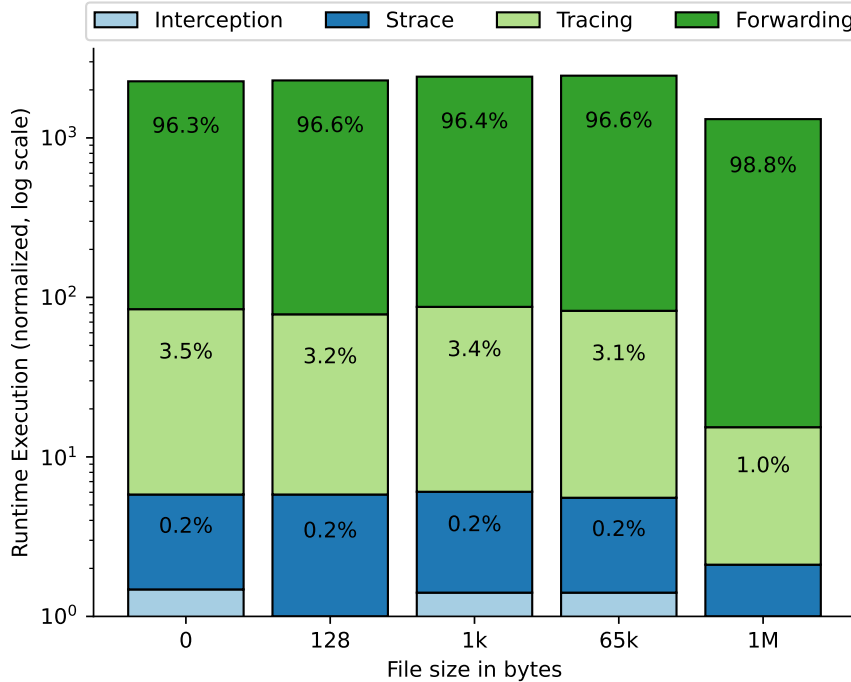**Figure 5.8:** Composition of overhead for wc.

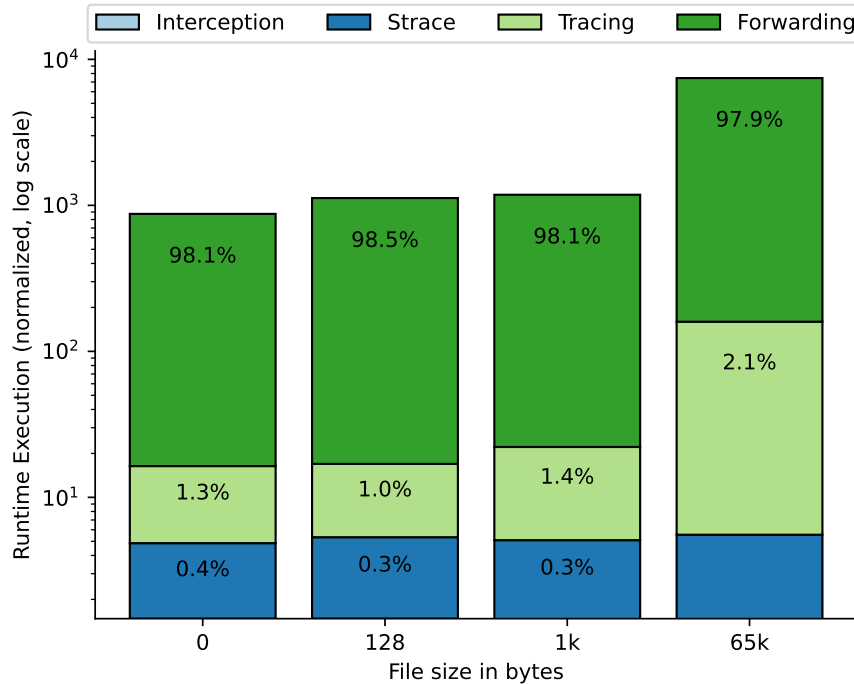**Figure 5.9:** Composition of overhead for sha512sum.

Figure 5.10: Composition of overhead for dd.

5.4 Conclusion

In conclusion, this Chapter provides insights into various methods for migrating processes from a device to an emulator. We described the implementation of Chestburster for hybrid execution by filtering system calls and forwarding memory to the device. Through its evaluation, we examined the correctness of program execution and measured the performance impact of decoding, filtering and forwarding system calls in various scenarios. Our findings showed that memory forwarding plays a significant role in the overall overhead, particularly due to the larger file size handled by the programs.

We aim to further examine the execution correctness by utilizing the Linux Test Project (LTP). It is a comprehensive testsuite to validate the functionality of the Linux kernel and in particular the system call interface. By using LTP, we hope to provide additional insights into the accuracy of our forwarding implementation and the robustness of the process migration.

Embedded systems such as routers often include web servers for network services and device management. These systems often use certificates and cryptographic keys stored in NVRAM and offer web access for controlling other devices, such as cameras. Therefore, the performance of Chestburster on these web servers is worth investigating. Another network

service that runs on an embedded system is the `ipbridge` application from the Hue Bridge. This application connects physical devices, such as Zigbee lights, to the local network and cloud services. The security and privacy of these systems are crucial, and it has been shown that they are critical [25, 119, 152, 173]. The application communicates with a Zigbee modem that is separate from the Linux system. This typical example illustrates the difficulty to analyze such system only in an emulator, as this would lack the context of the environment such as other devices present on the network. As such, it would be useful to instrument its execution to assess privacy risks and existing vulnerabilities.

The benefits of our approach could be further demonstrated with fuzzing and symbolic execution. By leveraging these vulnerability research techniques, it would be possible to expand the scope to applications requiring a high level of integration with their environment. For instance, more programs now rely on trusted execution environments (TEE) to protect and store secrets such as cryptographic keys and certificates which were previously stored in NVRAM.

Chapter 6

Bench of Embedded system Experiment for Reproducible Research

Experiments including physical devices present challenges in terms of accessibility, sharing and reproducibility. In this chapter, we will see the reasons to promote the reproducibility of experiments, particularly when physical devices are involved. Additionally, we propose an infrastructure architecture to address these challenges in the context of firmware security analysis.

6.1 Motivations

A core aspect of the journey to expand knowledge is experimenting. Experiments help to validate or refute hypotheses. One of the experiments' most important attributes is reproducibility. The modification of experiment parameters is crucial in observing the impact of results and improvements. In this way, the experimenters better understand and study the phenomena at play. This is part of the reason why Science is an iterative process. By sharing reproducible experiments, other scientists can in turn add their expertise to them and progress.

In particular, an emphasis has been recently put on making research more reproducible in computer science, and in system security in particular [12,22,177,187]. For this purpose, conferences promote the publication of the code and data used in the research together with the paper. Several conferences now award badges to publications whose artifacts have successfully been reproduced by an evaluation committee [21, 92, 131, 134, 178]. These badges characterize the way the artifacts have been audited and how reproducible they are. For instance, the ACM Artifact Review and Badging

version 2.0 [21] defines three independent types of badges: *Artifacts Evaluated*, *Artifacts Available* and *Results Validated*. Each describes a qualitative set of requirements applied to the artifacts associated with the research. In addition, it also provides ACM's three definitions to clear any confusion:

- ***Repeatability:*** *the measurement can be obtained with stated precision by the same team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same location on multiple trials. For computational experiments, this means that a researcher can reliably repeat her own computation.*

- ***Reproducibility:*** *the measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts.*

- ***Replicability:*** *The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently.*

According to the type of artifacts the experiment manipulates, different challenges may arise at the time of publication. First, it is often the source code that is the most easily published with the paper. This brings transparency to experiments by providing the exact operations executed. However, source code alone isn't sufficient to make an experiment reproducible. Dependencies, compilers, and other toolchain elements have a significant impact on the binary produced. The second difficulty with reproducibility is if, and how, to share the data used during an experiment. Data copyright may raise concerns while malware studies face ethical issues in openly sharing datasets. User privacy also limits the scope of legitimate traffic, behavior and traces that can be collected and published. More trivially, the size of the dataset has a direct influence on its ease to be shared.

Third, the environment of the experiment has a great impact on its results. Thanks to virtualization technologies, sharing the exact environment for software is often straightforward. In contrast, trade-offs have to be made with hardware. For dynamic analysis, emulation is the primary solution to abstract hardware requirements. But it is not always feasible, as shown by Fasano et al. [71]: the task itself can be onerous, and there is a too broad variety of devices to support. To circumvent these obstacles, a different approach is to record the execution into traces that can be replayed later and

somewhere else [64, 174]. It is however impossible to explore new execution paths with a replay. Both approaches suffer from limitations, highlighting the importance of using physical devices in the analysis loop.

Fourth, the experiment setup has an essential influence on the measured results. A wrong configuration can yield a different outcome from the original study. It is a crucial aspect for a fair comparison between methodologies.

Experiments interacting with hardware devices introduce four challenges:

1. *The cost of the device:* How expensive is it? Is the budget available for purchasing it?

2. *Availability:* Is the device still produced and sold? Is the device available and legal to operate in the user's region of the world?

3. *Storage:* Where to keep the device? How to make it accessible to use it?

4. *Safety:* Can the device be dangerous when manipulated?

We want to address these last questions by proposing the *Bench of Embedded systems Experiments for Reproducible Research*, BEERR. It is an infrastructure aiming to ease access to physical devices used in system security publications by making them available remotely. In addition, we want to congregate and propose a collection of experiment code and data that can be easily used with these devices. We focus on dynamic firmware analysis with a strong emphasis on studies looking at interactions with the hardware. We believe BEERR can help future researchers by simplifying access to published experiments, removing the burden of hardware management, promoting fair comparison between studies and being a key to fostering new ideas. BEERR web interface is accessible at: https://beerr.s3.eurecom.fr.

6.2 Overview

6.2.1 Computer Testbeds

To overcome the difficulties in reproducing, sharing and comparing experiments, scientific communities are building testbeds. They take a wide variety of forms depending on their objectives and the problems to be studied. They can be small robots placed in an arena to study swarm intelligence as done in the Robotarium [142]; a grid of radio transceivers to analyze wireless interference and performance on different topology (R2Lab [137]); inspect IoT devices interactions within an environment (FIT IoT-LAB [24] and CorteXLab [113]); or to study distributed systems networking and services

such as Emulab [185], CloudLab [66], PlanetLab [141] and more recently EdgeNet [163].

To simplify administrative work, and management and to be accessible to as many people as possible, front-end projects have been created to federate multiple testbeds [3, 38] and offer a common interface. A recent example of this case is the Fed4FIRE+ project which ran for 5 years. It gathered 20 different testbeds in Europe and led to many publications [3].

Continuous Integration (CI) helps developers automatically merge their changes into the main development tree. However, before merging, the changes need to be tested to minimize new bugs or regressions. Embedded systems are known to be a very heterogeneous ecosystem. It is therefore difficult to know upfront if a specific code change would work on all the hardware the project wants to support. An example of such a case is the Linaro Automated Validation Architecture (LAVA) [103], which aims to test deployments on Linux-based systems for the ARM architecture.

In the context of system security, building a testbed is an effective approach to studying a system [75, 133, 162, 189]. A diverse range of applications are reasonable: benchmarking [116], training [23, 114], or investigating multi-stage attacks on distributed systems. In other words, in all situations where the reproduction of the environment is crucial to understand the events that are at stake. Examples of such cases are often related to networking attacks like DDoS [91], industrial control systems such as SCADA [23, 114, 118, 147], or Internet-of-Things [162] where physical devices communicate with external services often located on the Internet.

Fuzzing has recently experienced considerable interest in software testing and vulnerability research because of its efficiency in bug finding. Yet establishing good methodologies and metrics to compare fuzzing techniques remain a challenge for researchers. For this reason, Google proposes a service called FuzzBench [116] to evaluate and compare fuzzers against a set of benchmarks.

6.2.2 Architecture

BEERR is divided into two parts. The first part is called the front node. It hosts the website with the scheduler and stores the experiment codes and data. The website lets the user register an account and submit tasks to the scheduler. The latter takes care of allocating and setting up the resources as well as opening the connection to the selected experiment node. Code and data for experiments are combined in portable container images which are stored in a local registry.

The second part is composed of independent experiment nodes. Each of these nodes holds a gateway which is the main access to the experiment

node for the user. From it, the user can interact with the available devices, upload files, download container images from the local registry and run its analysis. Other components such as debuggers and power switches help in controlling the device state under study.
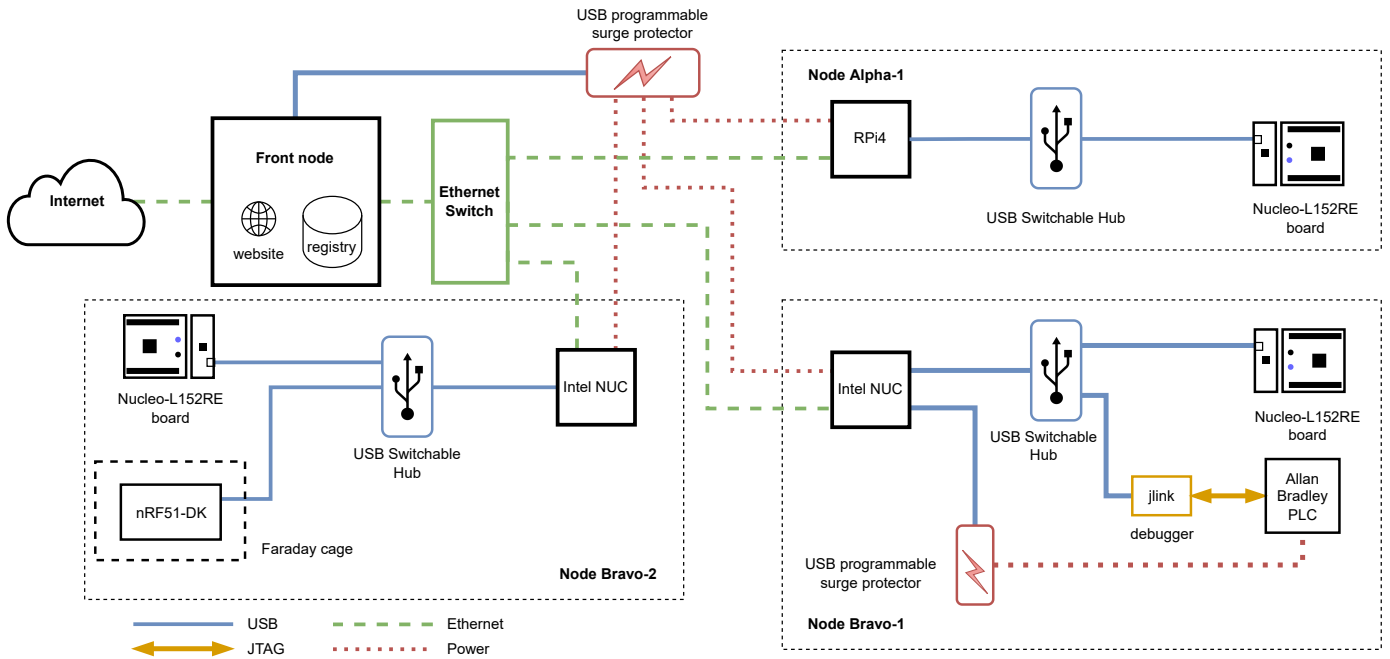

Figure 6.1: BEERR architecture.

The initial design is strongly inspired by existing testbeds, in particular, R2Lab [137] and FIT IoT-LAB [24] because they also target embedded systems. However, the type of analysis we want to perform is very different. Unlike R2Lab with its anechoic chamber, we don't aim to study radio propagation across different nodes in a controlled environment. Similarly, contrary to FIT IoT-LAB, there is no need to work with sensor networks, routing protocols or distributed applications on distinct typologies. As shown in the survey (Section 3.1), the analysis we target focuses on the firmware and code closely coupled with low-level hardware rather than across a pool of different devices. It may require powerful computing resources to handle a partial emulation of the system and its analysis. That is the reason we made the trade-offs to divide the testbed into independent bookable experiment nodes with a more or less powerful gateway. This design still offers the possibility to build systems composed of multiple devices behind a single gateway.

6.2.3   User Workflow

The typical workflow for the user first involves creating an account on the website and submitting its SSH public key. Then the user has to wait for the validation of his account from the person in charge of its affiliation. To create such a group of users (e.g., per university or company), an application explaining their motivations has to be sent to the administrators. The access is free of charge and mainly targets scientific activities.

The creation of an experiment requires selecting an experiment node, selecting a time slot and a duration, selecting an image to boot the gateway and optionally filling a public link to a git repository. This repository is a way for the user to prepare the experiment code upstream of its time slot. The repository may contain a `Dockerfile` which would be built and stored in the local registry.

When the booked time comes, the gateway is powered with the selected image and an SSH connection is opened between the front node and the gateway. The user can use this access to start a shell on the bare-metal gateway with root permissions. He can upload files, and install and configure the gateway as needed. No direct Internet access is provided on the gateway from our infrastructure, users can use SSH tunneling to share their own Internet connection.

Finally, the front node closes the SSH connection, resets the devices, cleans up the gateway disk and powers them off. The status of previous experiments is displayed on the website.

6.3   Implementation

6.3.1   Front Node

The front node hosts the website and the database with user and experiment data such as credentials, SSH keys, affiliation, experiment scheduled time, and repository link. The scheduler is integrated into the back end via the *Advanced Python Scheduler library*.

A local docker registry stores the experiments in the form of docker images. They are built locally from the git repository link provided by the user on the experiment submission.

The gateways boot on the network to facilitate image selection and management. We use *dnsmasq* because of its advantage to offer a complete and lightweight solution with DNS caching, DHCP and TFTP servers and its ease of configuration. The root filesystem is mounted using a union filesystem, OverlayFS. The read-only bottom layer is on an NFS server on the front node while the read-write upper layer is on the local disk. In this way, images are

easily added and updated by administrators while users can modify the system. Modifications are cleaned at the end of the experiment by removing the writable upper layer.

6.3.2 Experiment Nodes

Experiments are assembled in individual container images. We chose this solution because it facilitates the packaging, sharing, setup and resetting of experiments and parts of their environment. The essential advantage to use precise software versions from the time of publication offers the possibility to circumvent the hard task of maintaining experiments in different environments. A corner case occurs when a special kernel version is required. In this case, because the user has root shell access on the bare-metal gateway, he is free to use a virtual machine. Nevertheless, we did not observe such a situation in our survey (Section 3.1).

BEERR provides a set of pre-built images in the local registry. This allows the user to use them directly or use them as a base to build new images.

The first set of experiments we propose to reproduce is from the paper *Avatar²: A Multi-target Orchestration Platform* [123]. The second set targets the paper *What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices* [124] (see Section 3.1). In addition, we make available the already existing avatar² examples [1].

The experiments have been grouped into categories of nodes according to their nature and the devices they use. A node is composed of a gateway where the analysis is executed, and one or multiple devices which are subjects of the analysis. The user has root shell access on the bare-metal gateway where he can freely interact with the devices and build and run container experiments.

The gateways are either a Raspberry Pi 4 Model B 4GB or an Intel NUC BOXNUC8I5BELS1 with CPU Intel Core i5-8260U (Quad-Core 1.6/3.9 GHz, 8 threads, 6M cache), 16GB DDR4, 250Go NVMe SSD. The Raspberry Pi has the advantage to be relatively cheap, and space and power efficient. Yet its processor architecture is based on the ARM instruction set, which might cause compatibility problems with some experiments. Indeed, most research experiments work in a limited environment where portability is not always the main objective. For example, the PANDA emulator only supports host machines with x86_64 architecture. In this situation, the experiment needs to be carried out within the second node type using Intel NUC computers. We, therefore, use two categories of experiment nodes:

- **Alpha nodes**. The Alpha nodes are designed as low-cost entry nodes. They also allow new users to familiarize themselves with the envi-

ronment through a set of basic experiences. Each is composed of a
Raspberry Pi 4 with Nucleo boards and a USB switch.

- **Bravo nodes**. The Bravo nodes are more powerful and will be used
  when rehosting with powerful emulation is needed. Those experi-
  ments rely on an Intel NUC (instead of Raspberry Pi). This makes it
  possible to run more complex experiments or experiments relying on
  tools that are not portable to ARM architecture, e.g., those that rely on
  the PANDA emulator.
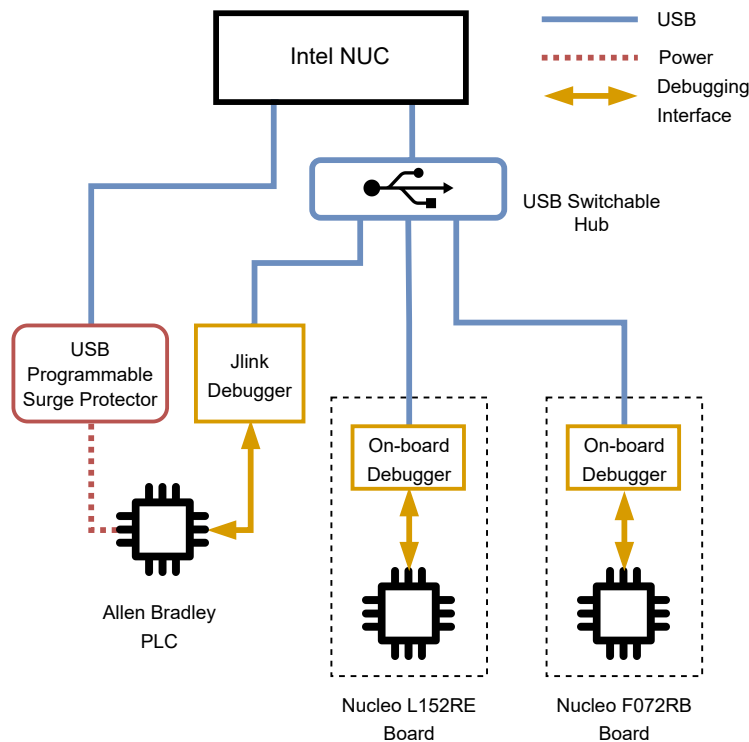
Figure 6.2: An example layout for a Bravo node.

6.3.3 Physical devices

The following devices are available on the nodes. We refer to Table 3.2 in
Section 3.1 to show which experiments they allow reproducing.

- STM32 Nucleo L152RE (ARM Cortex-M3) and F072RB (ARM Cortex-
  M0) boards,

- Nordic Semiconductor nRF51-DK for BLE (ARM Cortex-M3),

- Cypress CYW920735Q60EVB-01 BLE Evaluation Kit (SoC CYW20735 with ARM Cortex-M4),

- Allen Bradley 1769-L16ER-BB1B CompactLogix

In addition, other components are present to control the state of devices:

- Yepkit YKUSH, a USB Switchable Hub to turn on and off a USB link,

- a USB programmable surge protector to power devices using a power plug,

- SEGGER J-Link Debug Probes to interface with JTAG ports

6.4 Discussion

6.4.1 Infrastructure security

BEERR is mainly intended for the scientific community, without excluding other potential collaborations. The affiliation link makes the user accountable for their conduct. It is a best-effort approach where we rely on good behavior from the users. We plan to treat malicious behavior on a case-by-case basis and terminate the corresponding user or affiliation accesses.

We nevertheless implemented minimal mechanisms to protect fair access to the service and preserve the integrity of the system. Time spent using BEERR is divided into 55 minutes time slots and daily quotas to avoid monopolizing all resources. To preserve a clean state at the start of a session, we use an overlay filesystem to store modifications.

We do not protect against any attack attending to modify the gateway bootloader. Systems under test are also not protected against modifications. This is a desired behavior as the user might need to flash different firmware. However, nothing prevents permanent modifications such as a blowing fuse. We have chosen as the first step to giving freedom to the user before restricting its capabilities. We trust the user to do their best to not intentionally brick the devices. If users need specific requirements, we encourage them to contact us to discuss their feasibility. In the future as more expensive or hard-to-operate devices are incorporated, we might consider implementing a group policy for access to categories of experimental nodes.

Chapter 7

Conclusion and future work

Through this thesis, we explored various methods for improving the execution of firmware in an emulator by including the device in the analysis, a technique known as Hardware-in-the-loop. This approach addresses the limitations of emulation support for specific hardware such as peripherals as well as providing a higher degree of execution fidelity and accuracy close to that of the original environment. The BEERR testbed aims to tackle the challenge of hardware accessibility for security analysis and rehosting in particular. Through Chestburster, we focused on Linux user-mode programs and examined how the system call interface enables hybrid execution for processes between the emulator and the device. This presents numerous challenges due to the diversity and complexity of the system call interface and the limited control over the device and its running kernel. Indeed, Linux has approximately 380 system calls, and we have been able to confirm the functioning of forwarding for a hundred of them. To achieve this, we proposed a novel technique that can be implemented without making any modifications to the program or kernel and is versatile enough to be considered for other POSIX systems. This system call-forwarding method can be further improved to achieve a more comprehensive hybrid execution drawing inspiration from research on distributed operating systems. Moreover, this distributed execution opens up new perspectives and applications enabling more sophisticated security analysis usage on firmware devices such as fault injection, reverse-engineering, fuzzing, taint analysis and symbolic execution.

7.1 Distributed operating systems

Distributed operating systems aim to provide user programs with a unified view of a computer cluster. They offer the possibility to migrate processes and forward system calls to their original kernel.

MOSIX [33, 34, 107] describes a group of mechanisms to convert a UNIX system into a distributed operating system. It divides the program into two parts: the user context called remote and the system context called deputy. It allows the remote to be migrated to any node while the deputy has to stay in its original home node. Instead, our approach keeps two system contexts synchronized to allow the possibility to execute certain system calls locally. Gobelins [107, 120, 180] insert a middleware between the Linux kernel virtual and physical layers. This allows for the interception and deviation of various events to multiple nodes, enabling process migration. To achieve this, Gobelins modifies core kernel structures such as `task_struct`, `mm_struct`, `vm_area_struct` and `file_struct`. Plan 9 [143, 181] reduces the total number of system calls by presenting all system resources and services as files. This way, a remote procedure call (RPC) mechanism named 9P protocol is used to access both local and remote resources indifferently.

Although distributed operating systems share a lot of interesting techniques suitable for HIL firmware security analysis such as process migration and system call forwarding, they present significant drawbacks hindering their reuse as it. These limitations include a main focus on CPU-intensive workload, a lack of support for certain I/O operations [32, 107], the absence of a filtering mechanism to choose between local and remote system calls, and a lack of emulation support for non-x86 architectures. Additionally, the closed-source nature of available systems makes it difficult to extend their capabilities. Furthermore, the method of implementing system call forwarding through kernel patches is not deemed practical in the context of binary firmware analysis. Though it appears that MOSIX release 4 has fully reimplemented its mechanisms in user mode [32].

In our context, process migration is limited by the inability to capture the entirety of the process resources without instrumenting the kernel. Modifying the kernel on the device is challenging. Debugging ports may not be available and the necessary toolchain and kernel headers to compile kernel modules may not be accessible. Generating a toolchain and kernel headers compatible requires identifying the kernel configuration and accurately reconstructing the layout of various data structure [136]. While this has been achieved to some extent for memory forensics profiles, it remains a challenge for recompiling kernel modules. Therefore, migrating processes in a limited context where control over the kernel is not available remains a challenge.

7.2 Performance

Performance plays a critical role in the usability of any system call analysis. As seen in the state-of-the-art Section 3.3, there is an incentive to minimize the runtime cost of interception and instrumentation techniques. With Chestburster, we aimed at demonstrating the feasibility of the approach and exploring its ability to handle various instrumentation scenarios related to tracing and forwarding. To this end, the architecture is designed with an orchestrator developed in Python to control simple targets. However, the evaluation has highlighted the main overheads, and we discuss potential improvements to enhance the overall performance.

The general instrumentation of system call tracing, decoding and filtering could benefit from being integrated into the `sysforward` library. This would shift decoding and filtering from an interpreted to a compiled execution in addition to reducing the number of messages exchanged with the orchestrator.

The implementation of the filtering with a dictionary of rules can be improved with a state-of-the-art filtering mechanism. For instance, Linux uses eBPF filters to monitor events in the kernel including system calls. This provides greater flexibility in filtering system calls because filters are small programs that can perform operations, unlike rule-based systems. These filters can be supported using a virtual machine or just-in-time (JIT) compilation, similar to the way `seccomp-bpf` operates in the kernel. However, it requires more effort from the analyst to write suitable filters compared to the ease of using static rules. Therefore, efforts towards automating the creation of filters through static or dynamic analysis hold potential benefits. Previous works [60, 78, 79] have aimed for a similar goal to enforce the principle of least privilege by reducing the number of authorized system calls for an application.

The memory forwarding in Chestburster could be enhanced by incorporating additional techniques discussed in Section 4.2.4. In particular, the use of `userfaultfd` and `libsigsegv` to intercept access to specific memory regions would facilitate synchronization between the device and the emulator. This is currently a limitation of Chestburster's design, which has an asymmetric capacity to initiate interactions between the process and the device environment. For instance, while a `read()` system call is an action started by the process, a write on a shared memory from another process on the device may not be transmitted to the emulator. Moreover, the `ioctl` function let driver programmers create custom interfaces. It can complicate the identification of memory blocks to forward during the system call. To address this challenge, taint tracking and symbolic execution could be applied to driver binaries to recover the interface and determine which memory blocks to for-

ward.

To further improve performance and remove the need for continuously forwarding system calls involving device interactions, a caching mechanism could be introduced. Kammerstetter et. al. [96] proposed this through *run-time program state approximation.* To uniquely identify peripheral access in the cache, the program state, consisting of the CPU context and stack memory, is hashed as the key for the cache entry. This approach has the benefit of considering the state of the program during character device access, but it does not fully capture the comprehension of the driver behavior. Execution traces could help build models from recorded data, reducing the number of interactions with the device and potentially eliminating the need for forwarding in certain circumstances. Similar works have been done, but for monolithic firmware [86, 165]. Another option is the *augmented process emulation* [198] method which involves alternating execution between user mode and full-system emulation to issue system calls not supported by the host kernel. Enhancing hybrid execution by offering a choice between the three modes of execution, native on the device, user mode and full-system emulation, would grants the possibility to benefit from the best of each. However, similar to the challenge to generate automaticlly filters, there would need to be a decision-making process to determine when and why to switch based on the application behavior. Finally, for completeness symbolic execution could also be used to assist in inferring a peripheral model.

7.3 Application

POSIX systems

Type-II firmware may also implement the POSIX interface, as is the case for real-time operating systems like FreeRTOS, VxWorks, QNX, and eCos. While the hybrid execution approach could be beneficial for these systems, a major challenge lies in the executor component. Currently it is implemented through the use of a debugger and the `ptrace` system call that is not part of the POSIX API. Therefore, alternative methods for replaying system calls need to be explored.

Security testing techniques

The ability to run the firmware in an emulated environment is just a starting point for dynamic analysis. This opens up the possibility to conduct binary security testing techniques such as fuzzing and symbolic execution. Fuzzing efficiency on bug finding relies on the capacity of executing a large number of input in a short period of time. However, this can be hindered by the

```
1   syzkaller/sys/linux/dev_snd_midi.txt
2
3   [...]
4   syz_open_dev$midi(dev ptr[in, string["/dev/midi#"]], id intptr, flags
        flags[open_flags]) fd_midi
5   write$midi(fd fd_midi, data ptr[in, array[int8]], len bytesize[data])
6   read$midi(fd fd_midi, data ptr[out, array[int8]], len bytesize[data])
7   ioctl$SNDRV_RAWMIDI_IOCTL_PVERSION(fd fd_midi, cmd
        const[SNDRV_RAWMIDI_IOCTL_PVERSION], arg ptr[out, int32])
8   [...]
9   snd_rawmidi_params {
10          stream                  flags[sndrv_rawmidi_stream, int32]
11          buffer_size             intptr
12          avail_min               intptr
13          no_active_sensing       int32:1
14          mode                    int32
15          reserved                array[const[0, int8], 12]
16  }
17  [...]
18  define SNDRV_RAWMIDI_IOCTL_STATUS32     _IOWR('W', 0x20, char[36])
19  [...]
```

Listing 7.1: Syzkaller interface specification.

overhead of intercepting, filtering, forwarding system call and synchronizing the memory. It is therefore crucial to focus on improving performance and scalability.

Additionally, fault injection at the system call level can be used to evaluate the responsiveness of user mode applications [76, 186, 195]. Tampering with system call arguments and return values provide opportunities to exercise error handling code paths [102]. This technique can be used in combination with fuzzing to improve test coverage [138]. In embedded systems, fault injection can be utilized to test proprietary peripherals that are challenging to evaluate through other means.

Recovering semantics

The approach to decoding could be further improved to recover the system call semantics, leading to a deeper understanding of what the program is doing.

Syzkaller [84] is a fuzzer developed at Google that targets kernels through its various interfaces exposed via system calls. To facilitate program manipulation during the generation, mutation, minimization and validation phases, a description language called *syzlang* has been created. The programs are described as sequences of system calls. Each tested interface requires writing upstream and manually its specification. Listing 7.1 is an extract taken from the Linux audio MIDI interface.

The description language does not directly describe system calls, but rather the operations performed on and the data exchange with the inter-

face. Thinking at a higher level allows to take into account relationships within the data. For instance, in the `write$midi` operation, the relationship between the second and third arguments is expressed as `len bytesize[data]`.

In this way, it is possible to add more meaning to a trace of system calls by both gathering the sequence of calls to the interface and linking the transferred data (with the kernel) between calls. This would bring benefits for both a human being who could read a trace and understand more easily what the programs do; but also for a machine program for which it would be more comfortable to reflect on and operate on the trace similarly to what syzkaller is already doing for its test program generation. Thus, this would be useful in the first place to reverse engineering programs dynamically from their execution traces. For instance, semantics reconstruction from lower traces always has been a difficult problem as highlighted in [129]. Second, the idea may enhance the work of forwarding system calls for rehosting by being able to construct a model behind the interface which would allow removing the burden of forwarding system calls to the device.

Kernel drivers

Future work could also involve applying a similar hardware-in-the-loop approach to kernel drivers. The concept remains the same: using remote procedure calls (RPC) to transfer the control flow and copying necessary memory. Despite the lack of a stable internal API in the kernel, drivers need to export symbols for linking during loading. If low latency requirements are not a concern [170], it may be possible to intercept and perform RPC on these functions. This would enable deeper introspection about kernel driver execution without the need to boot the entire kernel [94, 104]. This could make it easier to identify vulnerabilities through fuzzing and symbolic execution. However, a major challenge would still be executing and replaying the calls within the kernel on the device.

Testbed

The BEERR testbed aims to address the challenge of hardware accessibility for hardware-in-the-loop security analysis by providing remote access and experiment sharing. However, the true advantage of HIL lies in the integration of the device in an environment. BEERR does not consider this aspect further than the different component on the same board. There is a significant challenge in setting up a full environment consisting of multiple devices that interact with each other. This raises questions about how to control the full environment, track its state, take snapshots, restore them, and have visibility into every aspect of the testbed, including the execution of firmware and wireless transmissions.

List of Tables

List of Figures

List of Acronyms

CGC Cyber Grand Challenge

FBW Fly-by-wire

HIL Hardware-in-the-loop

JIT Just-in-time compilation

RPC Remote Procedure Call

MVX Multi-Variant eXecution

I/O Input/Output

ISA Instruction Set Architecture

CPU Central Processing Unit

GPU Graphics Processing Unit

FPGA Field-Programmable Gate Array

VT Virtualization Technology

SoC System-on-Chip

ROM Read-Only Memory

EEPROM Electrically Erasable Programmable Read-Only Memory

RAM Random Access Memory

NVRAM Non-Volatile Random Access Memory

NAND NOT AND

USART Universal Synchronous/Asynchronous Receiver/Transmitter

UART Universal Asynchronous Receiver/Transmitter

SPI Serial Peripheral Interface

GPIO General Purpose Input/Output

DDR Double Data Rate

I2C Inter-Integrated Circuit

JTAG Joint Action Test Group

MMIO Memory-Mapped Input/Output

IRQ Interrupt Request

DMA Direct Memory Access

PCI Peripheral Component Interconnect

MMU Memory Management Unit

MPU Memory Protection Unit

OS Operating System

POSIX Portable Operating System Interface

ELF Executable and Linkable Format

BSS Block Started by Symbol

GOT Gloabal Offset Table

PID Process IDentifier

TID Thread IDentifier

UID User IDentifier

GID Group IDentifier

IPC Inter-Process Communication

RWX Read-Write-Execute

VFS Virtual FileSystem

LKM Loadable Kernel Module

FUSE Filesystem in Userspace

vDSO virtual Dynamic Share Object

QEMU Quick EMUlation

AFL American Fuzzing Lop

BPF Berkeley Packet Filter

eBPF extended Berkeley Packet Filter

CRIU Checkpoint/Restore In Userspace

RTOS Real-Time Operating System

SCADA Supervisory Control and Data Acquisition

PLC Programmable Logic Controller

DNS Domain Name System

DHCP Dynamic Host Configuration Protocol

TCP Transmission Control Protocol

SSH Secure SHell

TFTP Trivial File Transfer Protocol

NFS Network FileSystem

Bibliography

[1] avatar² examples repository. https://github.com/avatartwo/avatar2-examples.

[2] Documents about the D-17(B) guidance system. http://www.bitsavers.org/pdf/autonetics/d17/. (accessed: 01.01.2023).

[3] FED4FIRE+ website. https://www.fed4fire.eu/. (accessed: 15.12.2021).

[4] ftrace wiki page. https://www.kernel.org/doc/html/latest/trace/index.html. (accessed: 01.09.2019).

[5] Global embedded system market (2022 to 2027). https://www.globenewswire.com/en/news-release/2022/06/24/2468712/28124/en/Global-Embedded-System-Market-2022-to-2027-Featuring-Intel-Renesas-Texas-Instruments-and-Marvell-Among-Others.html. (accessed: 01.01.2023).

[6] Global embedded systems industry. https://www.reportlinker.com/p01171466/Global-Embedded-Systems-Industry.html. (accessed: 01.01.2023).

[7] GNU libsigsegv. https://www.gnu.org/software/libsigsegv/. (accessed: 01.01.2023).

[8] Kernel ABI readme. https://www.kernel.org/doc/Documentation/ABI/README. (accessed: 01.01.2023).

[9] Kernel syscall stable ABI. https://www.kernel.org/doc/Documentation/ABI/stable/syscalls. (accessed: 01.01.2023).

[10] LWN article on eBPF seccomp filters. https://lwn.net/Articles/857228/.

[11] perf wiki page. https://perf.wiki.kernel.org/index.php/Main_Page. (accessed: 01.09.2019).

[12] Sharing expertise and artifacts for reuse through cyber-security community hub (SEARCCH) project. https://searcch.cyberexperimentation.org/about. (accessed: 15.12.2021).

[13] signal - overview of signals. https://man7.org/linux/man-pages/man7/signal.7.html. (accessed: 01.01.2023).

[14] strace the linux syscall tracer. https://strace.io/. (accessed: 01.01.2023).

[15] Therac-25. https://en.m.wikipedia.org/wiki/Therac-25. (accessed: 01.01.2023).

[16] userfaultfd - create a file descriptor for handling page faults in user space. https://man7.org/linux/man-pages/man2/userfaultfd.2.html. (accessed: 01.01.2023).

[17] vDSO - overview of the virtual ELF dynamic shared object. https://man7.org/linux/man-pages/man7/vdso.7.html. (accessed: 01.01.2023).

[18] The virtual AGC project. https://www.ibiblio.org/apollo/. (accessed: 01.01.2023).

[19] Wind river SIMICS. https://www.windriver.com/products/simics/, 1998. (accessed: 01.01.2023).

[20] system call intercepting library. https://github.com/pmem/syscall_intercept/, 2016. (accessed: 18.02.2020).

[21] ACM. Artifact review and badging - version 2.0. https://www.acm.org/publications/policies/artifact-review-badging. (accessed: 15.12.2021).

[22] ACSAC. Paper artifacts. https://www.acsac.org/2020/submissions/papers/artifacts/. (accessed: 15.12.2021).

[23] Sridhar Adepu, Nandha Kumar Kandasamy, and Aditya Mathur. Epic: An electric power testbed for research and training in cyber physical systems security. In *Computer Security*. 2018.

[24] Cédric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frédéric Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. 2015.

[25] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019.

[26] Davide Balzarotti Andrea Oliveri. In the Land of MMUs: Multiarchitecture OS-Agnostic Virtual Memory Forensics. *ACM Trans. Priv. Secur.*, 2022.

[27] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, pages 583–600, 2016.

[28] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules. 2023.

[29] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. The mayhem cyber reasoning system. *IEEE Security & Privacy*, 2018.

[30] Michelle Bailey and Julie Doerr. Contributions of hardware-in-the-loop simulations to Navy test and evaluation. In *Technologies for synthetic environments: Hardware-in-the-loop testing*. SPIE, 1996.

[31] Julian Bangert, Rebecca Shapiro, and Sergey Bratus. Weird machines and revisiting trusting trust for binary toolchains. https://www.cs.dartmouth.edu/~sergey/trust/30c3-chain-of-trust.pdf, 2013. (accessed: 01.01.2023).

[32] A Barak and A Shiloh. The MOSIX cluster operating system for distributed computing on Linux clusters, multi-clusters and clouds. 2013.

[33] Amnon Barak, Shai Guday, and Richard G Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*. Springer, 1993.

[34] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. *Proc. 5-th Annual Linux Expo*, 100:50, 1999.

[35] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[36] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, 2005.

[37] Emery D Berger and Benjamin G Zorn. DieHard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 2006.

[38] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 2014.

[39] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[40] Matthias Börsig, Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker, and Ingmar Baumgart. Fuzzing framework for ESP32 microcontrollers. In *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, 2020.

[41] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

[42] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[43] Amat Cama. Corrupting the ARM exception vector table. https://doar-e.github.io/blog/2014/04/30/corrupting-arm-evt/. (accessed: 01.09.2019).

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

[45] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European conference on Computer systems*, 2010.

[46] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[47] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[48] CRIU Community. Checkpoint/restart in userspace (CRIU). https://criu.org/. (accessed: 01.01.2023).

[49] CRIU Community. CRIU linux kernel options. https://criu.org/Linux_kernel. (accessed: 01.01.2023).

[50] Emilio Coppa, Heng Yin, and Camil Demetrescu. SymFusion: Hybrid Instrumentation for Concolic Execution. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

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

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

[53] Nassim Corteggiani and Aurélien Francillon. HardSnap: Leveraging hardware snapshotting for embedded systems security testing. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[54] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[55] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *USENIX Security Symposium*, 2006.

[56] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*, 2018.

[57] DARPA. Cyber grand challenge (CGC) (archived). https://www.darpa.mil/program/cyber-grand-challenge. (accessed: 01.01.2023).

[58] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium*, 2013.

[59] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.

[60] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *RAID*, 2020.

[61] Sebastian Dietz. Firmware re-hosting, an evaluation and verification of FirmAE, 2020.

[62] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[63] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[64] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Program Protection and Reverse Engineering Workshop*, 2015.

[65] dong-hoon you. Android platform based linux kernel rootkit. http://www.phrack.org/issues/68/6.html. (accessed: 01.09.2019).

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet,

Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[67] Michael Eddington. Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/, 2004. (accessed: 01.01.2023).

[68] Hirofumi Eguchi and Tadashi Yamashita. Benefits of HWIL simulation to develop guidance and control systems for missiles. In *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing V*, 2000.

[69] elfmaster. Extended core file snapshot (ECFS). https://github.com/elfmaster/ecfs. (accessed: 01.01.2023).

[70] Martha B Evans and Lawrence J Schilling. The role of simulation in the development and flight test of the HiMAT vehicle. Technical report, 1984.

[71] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[72] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020.

[73] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[74] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. Dissecting American Fuzzy Lop–A FuzzBench Evaluation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[75] Daniel S. Fowler, Madeline Cheah, Siraj Ahmed Shaikh, and Jeremy Bryans. Towards a Testbed for Automotive Cybersecurity. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[76] Alessandro Gario. A BPF-based syscall fault injector. https://github.com/trailofbits/ebpfault. (accessed: 01.01.2023).

[77] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 2017.

[78] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[79] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2C: Fine-grained configuration-driven system call filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[80] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007.

[81] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[82] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[83] Peter Goodman and Artem Dinaburg. The past, present, and future of cyberdyne. *IEEE Security & Privacy*, 2018.

[84] Google. Syzkaller, an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller. (accessed: 01.01.2023).

[85] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. *IEEE Access*, 2020.

[86] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[87] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 2016.

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

[89] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin RB Butler. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In *29th Annual Network and Distributed System Security Symposium, NDSS*, 2022.

[90] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Computer Architecture News*, 2015.

[91] Alefiya Hussain, David DeAngelis, Erik Kline, and Stephen Schwab. Replicated Testbed Experiments for the Evaluation of a Wide-range of DDoS Defenses. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020.

[92] IEEE. Submitting a paper to TPDS. https://www.computer.org/csdl/journal/td/write-for-us/15085. (accessed: 15.12.2021).

[93] Bart Jacob, Paul Larson, B Leitao, and SAMM Da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[94] Muhui Jiang, Lin Ma, Yajin Zhou, Qiang Liu, Cen Zhang, Zhi Wang, Xiapu Luo, Lei Wu, and Kui Ren. ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[95] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[96] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[97] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM symposium on Information, computer and communications security (AsiaCCS)*, 2014.

[98] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference*, 2020.

[99] Russel King. Linux patch: ARM: probes: avoid adding kprobes to sensitive kernel-entry/exit code. Commit: c608906165355089a4de3c9133c72e81e011096c. (accessed: 01.09.2019).

[100] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[101] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[102] Dmitry Levin. Can strace make you fail? strace syscall fault injection. https://archive.fosdem.org/2017/schedule/event/failing_strace/attachments/slides/1630/export/events/attachments/failing_strace/slides/1630/strace_fosdem2017_ta_slides.pdf. (accessed: 01.01.2023).

[103] Linaro. Linaro automated validation architecture (LAVA). https://validation.linaro.org/. (accessed: 15.12.2021).

[104] Qiang Liu, Cen Zhang, Lin Ma, Muhui Jiang, Yajin Zhou, Lei Wu, Wenbo Shen, Xiapu Luo, Yang Liu, and Kui Ren. Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[105] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[106] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. A Survey on Function and System Call Hooking Approaches. *Journal of Hardware and Systems Security*, 2017.

[107] Renaud Lottiaux, Pascal Gallard, Geoffroy Vallée, Christine Morin, and Benoit Boissinot. OpenMosix, OpenSSI and Kerrighed: a comparative study. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, 2005.

[108] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 2005.

[109] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[110] Dominik Maier and Fabian Toepfer. BSOD: Binary-only scalable fuzzing of device drivers. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[111] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[112] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. A framework for behavior-based malware analysis in the cloud. In *International Conference on Information Systems Security*, 2009.

[113] Abdelbassat Massouri, Leonardo Cardoso, Benjamin Guillon, Florin Hutu, Guillaume Villemaud, Tanguy Risset, and Jean-Marie Gorce. CorteXlab: An open FPGA-based facility for testing SDR amp; cognitive radio networks in a reproducible environment. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2014.

[114] Aditya P Mathur and Nils Ole Tippenhauer. SWaT: A water treatment testbed for research and training on ICS security. In *2016 international workshop on cyber-physical systems for smart water networks (CySWater)*, 2016.

[115] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[116] Metzman, Jonathan and Szekeres, László and Maurice Romain Simon, Laurent and Trevelin Sprabery, Read and Arya, Abhishek. FuzzBench:

An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[117] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990.

[118] Mocanu, Stéphane and Puys, Maxime and Thevenon, Pierre-Henri. An open-source hardware-in-the-loop virtualization system for cybersecurity studies of scada systems. In *C&esar 2019 - Virtualization and Cybersecurity*, 2019.

[119] Philipp Morgner, Stephan Mattejat, and Zinaida Benenson. All your bulbs are belong to us: Investigating the current state of security in connected lighting systems. *arXiv preprint arXiv:1608.03732*, 2016.

[120] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. Kerrighed: a single system image cluster operating system for high performance computing. In *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings 9*, 2003.

[121] Bernhard Mueller. Hooking android system calls for pleasure and benefit. https://www.vantagepoint.sg/blog/82-hooking-android-system-calls-for-pleasure-and-benefit. (accessed: 01.09.2019).

[122] Marius Muench. *Dynamic binary firmware analysis: challenges & solutions*. PhD thesis, Sorbonne université, 2019.

[123] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, 2018.

[124] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*, 2018.

[125] Collin Mulliner and Charlie Miller. Fuzzing the phone in your phone (Black Hat USA 2009), 2009.

[126] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium*, 2021.

[127] Anh Nguyen-Tuong, David Melski, Jack W Davidson, Michele Co, William Hawkins, Jason D Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. Xandra: An autonomous cyber battle system for the Cyber Grand Challenge. *IEEE Security & Privacy*, 2018.

[128] Zhenyu Ning and Fengwei Zhang. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[129] Dario Nisi, Antonio Bianchi, and Yanick Fratantonio. Exploring Syscall-Based Semantics Reconstruction of Android Applications. In *Symposium on Research in Attacks, Intrusion, and Defenses (RAID)*, 2019.

[130] Dario Nisi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Lost in the Loader: The Many Faces of the Windows PE File Format. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[131] NISO. NISO RP-31-2021 reproducibility badging and definitions. https://groups.niso.org/apps/group_public/download.php/24810/RP-31-2021_Reproducibility_Badging_and_Definitions.pdf. (accessed: 15.12.2021).

[132] Paul Olivier, Xuan-Huy Ngo, and Aurélien Francillon. BEERR: Bench of Embedded system Experiments for Reproducible Research. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022.

[133] Pradeep Sharma Oruganti, Matt Appel, and Qadeer Ahmed. Hardware-in-Loop Based Automotive Embedded Systems Cybersecurity Evaluation Testbed. In *ACM Workshop on Automotive Cybersecurity*, 2019.

[134] OSF. Center for open science badges. https://osf.io/tvyxz/wiki/1.%20View%20the%20Badges/. (accessed: 15.12.2021).

[135] A. Portnoy P. Amini. Sulley fuzzing framework. https://github.com/OpenRCE/sulley, 2010. (accessed: 01.01.2023).

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

[137] Thierry Parmentelat, Mohamed Naoufal Mahfoudi, Thierry Turletti, and Walid Dabbous. A step towards runnable papers using R2lab, 2019.

[138] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[139] Florian Pester. ELK Herder. 2014.

[140] Brad Spengler Peter Busser. Paxtest v0.9.13. https://www.grsecurity.net/~spender/paxtest-0.9.15.tar.gz. (accessed: 01.01.2023).

[141] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. *ACM SIGCOMM Computer Communication Review*, 2003.

[142] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017.

[143] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Computing systems*, 1995.

[144] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[145] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[146] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, 2003.

[147] Qais Qassim, Norziana Jamil, Izham Zainal Abidin, Mohd Ezanee Rusli, Salman Yussof, Roslan Ismail, Fairuz Abdullah, Norhamadi Ja'afar, Hafizah Che Hasan, and Maslina Daud. A survey of SCADA testbed implementation approaches. *Indian Journal of Science and Technology*, 2017.

[148] Hany Ragab, Koen Koning, Herbert Bos, and Cristiano Giuffrida. BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer. 2022.

[149] Ole André V. Ravnås. frida.re. (accessed: 18.02.2020).

[150] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[151] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: Testing Drivers without Devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[152] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT goes nuclear: Creating a ZigBee chain reaction. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[153] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *USENIX Security Symposium*, 2020.

[154] Majid Salehi, Luca Degani, Marco Roveri, Daniel Hughes, and Bruno Crispo. Discovery and Identification of Memory Corruption Vulnerabilities on Bare-metal Embedded Devices. *IEEE Transactions on Dependable and Secure Computing*, (01):1–1, 2022.

[155] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[156] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[157] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 2005.

[158] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. "Weird Machines" in ELF: A spotlight on the underappreciated metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[159] Yan Shoshitaishvili, Antonio Bianchi, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, et al. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 2018.

[160] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[161] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[162] Shachar Siboni, Vinay Sachidananda, Yair Meidan, Michael Bohadana, Yael Mathov, Suhas Bhairav, Asaf Shabtai, and Yuval Elovici. Security testbed for Internet-of-Things devices. *IEEE transactions on reliability*, 2019.

[163] Marco Simioni, Pavel Gladyshev, Babak Habibnia, and Paulo Roberto Nunes de Souza. Monitoring an anonymity network: Toward the deanonymization of hidden services. *Digital Investigation*, 2021.

[164] Mitchell E Sisle and Edward D McCarthy. Hardware-in-the-loop simulation for an active missile. *Simulation*, 1982.

[165] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. Conware: Automated modeling of hardware peripherals. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[166] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[167] Lindsay Sterle and Suman Bhunia. On solarwinds orion platform security breach. In *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, 2021.

[168] Robert Swiecki. honggfuzz. https://github.com/google/honggfuzz, 2010. (accessed: 01.01.2023).

[169] Saad Talaat. Intercepting system calls and dispatchers. https://ruinedsec.wordpress.com/2013/04/04/modifying-system-calls-dispatching-linux/. (accessed: 01.09.2019).

[170] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[171] Willy Tarreau. Nolibc: a minimal c-library replacement shipped with the kernel. https://lwn.net/Articles/920158/. (accessed: 31.01.2023).

[172] Microsoft Defender Security Research Team. Attack inception: Compromised supply chain within a supply chain poses new risks. https://www.microsoft.com/en-us/security/blog/2018/07/26/attack-inception-compromised-supply-chain-within-a-supply-chain-poses-new-risks/, 2018. (accessed: 01.01.2023).

[173] Mathieu Thiery, Vincent Roca, and Arnaud Legout. Privacy implications of switching ON a light bulb in the IoT world. 2019.

[174] Sam L Thomas, Jan Van den Herrewegen, Georgios Vasilakis, Zitai Chen, Mihai Ordean, and Flavio D Garcia. Cutting through the complexity of reverse engineering embedded devices. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

[175] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 1984.

[176] @ulexec. ELF crafting advance anti-analysis techniques for the linux platform. https://github.com/radareorg/r2con2019/blob/master/talks/elf_crafting/ELF_Crafting_ulexec.pdf, 2019. (accessed: 01.01.2023).

[177] USENIX. USENIX security '20 artifact evaluation information. https://www.usenix.org/conference/usenixsecurity20/artifact-evaluation-information. (accessed: 15.12.2021).

[178] USENIX. USENIX security '22 call for artifacts. https://www.usenix.org/conference/usenixsecurity22/call-for-artifacts. (accessed: 15.12.2021).

[179] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, et al. Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[180] Geoffroy Vallee, Christine Morin, Renaud Lottiaux, J Berthou, and Ivan Dutka Malen. Process migration based on gobelins distributed shared memory. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002.

[181] Eric Van Hensbergen and Ron Minnich. Grave Robbers from Outer Space: Using 9P2000 Under Linux. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[182] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[183] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. GHUMVEE: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security*, 2012.

[184] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *NDSS*, 2017.

[185] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[186] William Woodruff. How to write a rootkit without really trying. https://blog.trailofbits.com/2019/01/17/how-to-write-a-rootkit-without-really-trying/. (accessed: 01.01.2023).

[187] WOOT. WOOT '19 artifact evaluation information. https://www.usenix.org/conference/woot19/artifact-evaluation-information. (accessed: 15.12.2021).

[188] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware rehosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021.

[189] Muhammad Mudassar Yamin, Basel Katt, and Vasileios Gkioulos. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security*, 2020.

[190] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[191] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of Embedded Systems: A Survey. *ACM Computing Surveys*, 2022.

[192] Jonas Zaddach. *Development of novel binary analysis techniques for security applications.* PhD thesis, 2015.

[193] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[194] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2014. (accessed: 01.01.2023).

[195] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Maximizing error injection realism for Chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[196] Chengyu Zheng, Mila Dalla Preda, Jorge Granjal, Stefano Zanero, and Federico Maggi. On-chip system call tracing: A feasibility study and open prototype. In *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016.

[197] Min Zheng, Mingshen Sun, and John CS Lui. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *2014 international wireless communications and mobile computing conference (IWCMC)*, 2014.

[198] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[199] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.