Challenges in Firmware Re-Hosting, Emulation, and Analysis

CHRISTOPHER WRIGHT, Purdue University
WILLIAM A. MOEGLEIN, Sandia National Laboratories
SAURABH BAGCHI and MILIND KULKARNI, Purdue University
ABRAHAM A. CLEMENTS, Sandia National Laboratories

System emulation and firmware re-hosting have become popular techniques to answer various security and performance related questions, such as determining whether a firmware contain security vulnerabilities or meet timing requirements when run on a specific hardware platform. While this motivation for emulation and binary analysis has previously been explored and reported, starting to either work or research in the field is difficult. To this end, we provide a comprehensive guide for the practitioner or system emulation researcher. We layout common challenges faced during firmware re-hosting, explaining successive steps and surveying common tools used to overcome these challenges. We provide classification techniques on five different axes, including emulator methods, system type, fidelity, emulator purpose, and control. These classifications and comparison criteria enable the practitioner to determine the appropriate tool for emulation. We use our classifications to categorize popular works in the field and present 28 common challenges faced when creating, emulating, and analyzing a system from obtaining firmwares to post emulation analysis.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems**; **Firmware**; **Embedded hardware**; **Embedded software**; *Real-time systems*; • **Hardware** → *Simulation and emulation*;

Additional Key Words and Phrases: Firmware re-hosting, system emulation, embedded systems, emulation fidelity, emulator classification, binary analysis, reverse engineering, emulation challenges

ACM Reference format:
Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. 2020. Challenges in Firmware Re-Hosting, Emulation, and Analysis. *ACM Comput. Surv.* 54, 1, Article 5 (December 2020), 36 pages.
https://doi.org/10.1145/3423167

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This article describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government. SAND2020-10079 J.

Authors' addresses: C. Wright, 1106 S 5th Street, Lafayette, IN 47905; email: christopherwright@purdue.edu; W. A. Moeglein, Sandia National Laboratories, PO Box 5800 – MS 0671, Albuquerque, NM 87185; email: wmoegle@sandia.gov; S. Bagchi, 465 Northwestern Ave., West Lafayette, IN 47907; email: sbagchi@purdue.edu; M. Kulkarni, 465 Northwestern Ave., West Lafayette, IN 47907; email: milind@purdue.edu; A. A. Clements, Sandia National Laboratories, PO Box 5800 – MS 0672, Albuquerque, NM 87185; email: aacleme@sandia.gov.

0360-0300/2020/12-ART5
https://doi.org/10.1145/3423167

1 INTRODUCTION

The number of connected devices, from appliances to smart home and personal gadgets, has increased dramatically with the explosion of Internet of Things (IoT). Along with these every day gadgets, large-scale infrastructures, such as the electric grid, cellular networks, and other large control systems, have become smarter, digital, and interconnected [62, 98, 107, 120, 178]. Understanding how these systems work and discovering vulnerabilities in their firmware is an important and growing topic in academic and industrial research, with large companies paying millions of dollars for bugs found in their devices [1, 30, 32, 77, 118, 140]. The increased effort to protect connected devices has come from an increased awareness of their vulnerability and the attacks targeting them [3, 43, 82, 85, 101, 123, 166, 174, 183]. Many of these systems have access to sensitive financial data, personal information, or control critical processes. Malicious actors are exploiting vulnerabilities in these systems to cause harm to businesses, individuals, and critical systems such as electrical grids and cellular infrastructure.

Emulation of embedded systems is an emerging technique to accelerate discovery and mitigation of vulnerabilities in embedded system firmware. Embedded system emulation has traditionally been used during development to allow embedded software to be written and tested without the need for hardware. In cases where hardware is concurrently being developed, is costly to have in quantity, or is susceptible to damage, emulation is an appealing option. Just as emulation can be used to verify system behavior during development, it can be used for vulnerability research and analysis. Emulation provides the ability to deeply observe and instrument firmware in ways that are not possible on physical hardware. It can help analyze what operations are actually being performed at a lower level than static analysis of either the high level source code or even the binary level, and is a useful tool that is the basis for a host of vulnerability discovering techniques [35, 44, 91, 137, 146, 151, 160, 169] among other uses outlined in Section 2.

For a practitioner to use emulation or re-host[1] a firmware, there is a learning curve and a plethora of tools available. Our primary purpose is to provide an end-to-end guide to the practitioner for firmware re-hosting. We limit our focus on emulation and re-hosting to the embedded space, as virtually any system will have embedded devices, from wearable IoT devices to power plants, which if vulnerable will affect the entire system's security and functionality. We present an overview of current techniques/tools for the practitioner along with classification categories and techniques for evaluating which tool is best for the emulation task at hand. We provide tables for the practitioner to reference in subsequent challenge sections that specify whether a tool attempts to address the presented challenges, giving a starting point for the practitioner to evaluate the right tool to overcome challenges.

We proceed with a brief history and background of emulation in Section 2. We then introduce surveyed works in Section 3 before introducing different comparison techniques in Section 4. We continue by comparing surveyed works for emulation in Section 5. We discuss the necessary pre conditions and challenges for re-hosting firmware in Section 6. We then proceed to break the challenges into three categories, Pre-Emulation, Emulation, and Post Emulation, and provide tool comparisons in Section 7, Section 8, and Section 9 respectively. We finish by discussing different considerations when choosing re-hosting tools in Section 10 and a summary in Section 11.

[1]Re-hosting specifies that a binary that would run on a specific hardware is instead run on a host system using system emulation and is therefore "re-hosted."

2 EMULATION BACKGROUND

2.1 Evolution of Emulation

Since before the era of personal computers, emulation has been a technique used to broaden hardware use and increase simulation speed. For example, most printing software used to be designed for HP, so many non-HP printers would write an emulator to re-host the software designed for HP [73]. Re-using HP software allowed for faster time to market for new printers and reduced the development time and effort for creating new products.

Emulation theory was first developed in the early 1960s, with the 7070 Emulator for the IBM System/360 series of mainframe computers being the first implemented emulator [83]. This emulator allowed IBM's customers to continue running their existing applications after upgrading their hardware. As was the case for the 7070 emulator, early uses of emulation was to avoid obsolescence and increase hardware compatibility with the limited available software. Over time, manufacturers started creating hardware emulators to allow software development before the hardware production; decreasing product development time. Emulators are now becoming a popular tool for security analysis and logic debugging [91, 129, 160, 169].

Around the same time frame, simulation was also used, but it allowed for executing and expanding systems beyond what existed. Simulation is sometimes referred to in scientific modeling and investigating, but in this context, simulation is another technique to model the internals of a system. In the computer science context, simulation is modeling a system with implementation of the internals, whereas emulation is modeling by replacing some of the internals of the system. By replacing some of the internals, emulation can sometimes reduce complexity or increase firmware re-hosting speed. Emulation will often allow for original machine code to run directly on the emulator. Beyond the computer science context, simulation and emulation are sometimes used interchangeably, with simulation sometimes refering to a system being replaced by software. For our case, the distinction is not important, but the tools we survey mostly refer to their techniques as emulation.

One of the early emulation successes was Bochs [96], which was released in the early 1990s. It emulated the underlying hardware needed for PC operating systems development, which enabled completely isolating the OS from the hardware. This isolation enabled restarting the emulator instead of reconfiguring hardware during OS development. Bochs was originally commercial licensed but was open sourced in 2000. In addition to Bochs, many other emulators were created including DOSBox [52], FX!32 [27], and PCem [126]. These solutions were mainly geared for x86 or PC emulation.

As multiple emulators emerged, the *execution and memory fidelity* (i.e., how closely the emulated system matches the real system, sometimes referred to as accuracy) varied from high (Cycle and Register accurate) to low (Module and Black box accurate). We discuss fidelity and give more classification points in Section 4.3, but fidelity and emulator speed are perhaps the greatest distinguishing factors between different emulators.

2.2 Emulation Bases

In the early 2000s, *Simics* [105] was created and evolved to emulate multiple architectures including Alpha, x86-64, IA-64, ARM, MIPS (32- and 64-bit), MSP430, PowerPC (32- and 64-bit), SPARC-V8 and V9, and x86 instruction set architectures (ISAs). It was originally developed by the Swedish Institute of Computer Science (SICS) before moving to Virtutech and eventually working its way to Wind River Systems, who currently sells it. Simics is designed to have fidelity at the instruction level, allowing for interrupt testing to occur between any pair of instructions. It also provides configuration management, scripting, automated debugging (forward and reverse), and other built in static and dynamic analysis tools to help with constructing an emulated system. One popular use for Simics was during the DARPA Cyber Grand Challenge to automatically vet submissions to check whether the submitted binaries adhered to the competition infrastructure [168].

In contrast to Simics and emulators that work at the instruction level fidelity, *QEMU* [11] gives up some accuracy to improve emulation speed. Instead of working as a sub-instruction simulator (performing multiple actions per instruction), QEMU execution and emulation occurs at the basic block level (sequential, non-control flow instructions), by translating entire blocks of instructions to the host system's instruction set and executing the translated instructions. This allows QEMU to work much faster, as it does not have to check for interrupts at each instruction, and caching of blocks greatly reduces translation overhead. Because of its open-source license and community, QEMU has become one of the staples in academia and for industry professionals. It emulates the IA-32, x86, MIPS, SPARC, ARM, SH4, PowerPC, ETRAX CRIS, MicroBlaze, and RISC-V architectures and provides peripherals for many systems, making it and Simics two of the most widely used emulators.

One of the newest emulators available is *Ghidra Emulator*. Ghidra [119] is an open source software reverse engineering tool developed by the National Security Agency (NSA). The initial release in March 2019 contains emulation tools that allow for traditional software reverse engineering and emulation to be combined into the same environment. Because of the richness of features in these tools, there has been a large user base since release. Ghidra uses their own processor modeling language called Sleigh and an intermediate register transfer language called P-code, with each machine instruction translating to up to 30 P-code instructions. This implies that the Ghidra Emulator works at the sub-instruction level (multiple emulator instructions performed per machine instruction), giving a relatively high execution fidelity as a base. Ghidra currently supports various existing architectures including X86 16/32/64, ARM/AARCH64, PowerPC 32/64, VLE, MIPS, MSP430, Z80, AVR, and so on. To add a new architecture is simple in their framework, with the user only specifying how the new architectures instructions are dissassembled into the intermediate P-code language. Ghidra includes loaders, disassemblers, decompiler and analysis tools with the base of supplied analyses written in Java. Beyond the built-in emulator, there have been Ghidra P-code emulators emerging that allow for partial re-hosting of firmwares [153].

2.3 Related Vulnerability Discovery Techniques

As emulation has been used more frequently for vulnerability discovery, it is worth mentioning some closely related vulnerability discovery techniques that often leverage emulation. The techniques introduced are integrated into some frameworks that work with the base emulators, requiring at least some familiarity during discussion. We do not go into extensive depth for the areas, but rather overview the techniques briefly, and recommend further reading for an in depth review. We mention some of the tools that are more popular, but it is not necessary to understand their differences or techniques in the scope of our review, rather the techniques are mentioned briefly in subsequent sections when integrated into a tool.

Symbolic Execution is where symbols representing arbitrary values are supplied as inputs to a program (similar to letters in algebra representing numbers). The goal of symbolic execution is to analyze a program to determine what inputs can cause different parts of a program to execute. Rather than analyze and follow a single path with concrete values, a symbolic execution engine will use symbols to describe all program execution paths that can execute by using constraints on symbols.

Concolic Execution is when the tool will switch between using symbolic symbols and concrete symbols (like an algebra symbol having a set value, e.g., $x = 5$) during emulation or execution. When reverse engineering firmware, an analyst will occasionally want to determine under what conditions a program will execute a certain portion of code. Symbolic and Concolic execution are tools that help solve this problem among other vulnerability discovery uses. Symbolic execution is now a common software testing practice, even though it was introduced in the 1970s [20]. Commonly used symbolic and concolic execution tools include References [10, 13, 15, 16, 19, 23, 24, 28, 38, 47, 61, 79, 103, 111, 141, 144, 147, 151].

Fuzzing is an automated vulnerability/bug discovery technique where random inputs are provided, and the system observed for undesired behavior (e.g., crashes). There are many challenges to fuzzing and various tools that try to address these challenges. In the context of emulation, fuzzers often leverage the visibility into, and control over, a binary's execution to optimize their random inputs to improve their exploration of the binary. Some popular fuzzing tools that can be integrated with emulation are References [2, 14, 39, 51, 63–71, 80, 108, 127, 134, 135, 145, 156–158, 184].

3 SURVEYED WORKS

Prior to introducing emulation comparison axes, we introduce surveyed works that enable emulation and firmware re-hosting. We do not discuss or reintroduce Simics, QEMU, or Ghidra Emulator, as they are described in Section 2.2, but they are the base emulators used in the tools introduced in this section. Base emulators can either run in user mode emulation (i.e., running just user level applications) or full system emulation; with full-system emulation currently being the primary mode used for firmware re-hosting. Full-system emulation will emulate the processor as well as hardware peripherals; however, the set of emulated peripherals available in the base emulators are small compared to the diversity of hardware found in embedded systems. Much of the work in firmware emulation aims at solving this lack of emulated peripherals.

Avatar^2 [112] is a *dynamic multi-target orchestration and instrumentation framework* that focuses on firmware analysis. This tool was created by the same group as Avatar but is a multi-target orchestration tool that has completely been re-designed from the original Avatar implementation [185].

The main contribution of this tool is that it allows various other tools (angr, QEMU, gdb) to interact and transfer data. Using these tools it can enable hardware in the loop emulation, where portions of execution or memory accesses are carried out by physical hardware.

angr [151] is a symbolic execution engine, that has been integrated with Avatar^2 to enable combining symbolic execution and other analysis into the concrete base QEMU emulator. angr is a binary analysis framework that provides building blocks for many analyses, both static and dynamic. angr provides an architecture library, an executable and library loader, a symbolic execution engine, built in analyses, and a python wrapper around a binary code lifter. It is actively developed and used in various other academic works including References [9, 104, 124, 128, 148, 150, 158, 175, 176]. We include this as one of the surveyed works because of its ability to use base emulators while providing new approaches and solutions to overcome emulation challenges from beginning to end.

HALucinator [35] addresses the challenge of providing peripherals not implemented in the base emulator by observing that the interactions with peripherals are often performed by a Hardware Abstraction Layer (HAL). It uses Avatar^2 and QEMU as bases to intercept HAL calls and replace them. They do this by manually providing replacements for HAL functions that execute during re-hosting. It uses a library matching tool to identify the HAL functions in the firmware. The tool relies on the assumption that the majority of firmware programmers use Hardware Abstraction Libraries when writing firmware, which in our experience is a relatively safe assumption.

PANDA [50] is an open-source platform that builds on top of the QEMU whole system emulator that is used for architecture-neutral *dynamic analysis*. The main advantage of PANDA is that it adds the ability to record and replay executions, allowing for deep, whole-system analyses.

System record has partially been addressed in a hardware-software co-design approach [161] as well as under more restrictive assumptions, using a purely software approach [159, 162]. Such whole system record and replay is challenging especially considering the timing requirements. PANDA, using the QEMU base and abstracting some of the analyses, allows for using a single analysis implementation across multiple computer architectures while maintaining the speed of QEMU, allowing some timing challenges to be addressed.

Muench2018 , titled "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices" [113], demonstrate that tools used for desktop vulnerability discovery and testing do not necessarily transfer to the embedded space. In their article, they present different techniques used for vulnerability assessment implemented by instrumenting an the emulator used for re-hosting the firmware. They implement segment tracking, format specifier tracking, heap object tracking, call stack tracking, call frame tracking, and stack object tracking by combining PANDA and Avatar^2.

Using QEMU as a base, both *Firmadyne* [25] and Costin Firmware Analysis (referred herafter as *CostinFA* [43, 44] "A Large-Scale Analysis of the Security of Embedded Firmwares" and "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces") will extract the filesystem from a given firmware and re-host the filesystem on their own kernel. They perform system emulation using only software, not involving any physical embedded devices. Their tools work only for software images that can natively use chroot with QEMU. They then perform static and dynamic analysis on their re-hosted firmware to report vulnerabilities. Recent efforts that use the same type of approach as Costin and Firmadyne is ARMX [146]. ARMX requires more user interaction and configuration and only works for ARM architecture devices while requiring the rootfs from the firmware and the extracted NVRAM from the firmware. Given these extra requirements, the results of the emulated devices using their technique is usually of high quality.

PROSPECT [84] and *SURROGATES* [90] enable emulation by forwarding hardware and peripheral accesses to the actual device; a technique known as hardware in the loop. PROSPECT forwards peripheral and hardware interactions through a normal bus connection to the device, but allows for analysis and implementation without needing to know the details about the peripherals and external hardware connected to the system. SURROGATES, in contrast, uses a custom low-latency FPGA bridge between the host PCI Express bus and the system under test, allowing forwarding and state transfer to and from the system's peripherals with transfers much faster than the original Avatar [185] system.

P^2IM [57] uses a drop-in fuzzer (AFL [2]) to provide inputs to their base QEMU emulator. They abstract peripheral and hardware IO and then use the fuzzer for providing the feedback to the base emulator. Their approach is different from existing emulation approaches as it does not use hardware or detailed knowledge of the peripherals, as the fuzzer provides interactions. The fuzzer enables executing firmware with simple peripherals to be emulated, but its ability to enable emulation of firmware processing data from complex and stateful hardware is unknown.

In contrast to using a random fuzzer, *Pretender* [72] attempts to re-host firmware by using machine learning to provide models of hardware interactions. Their system will record hardware interactions and all accesses to memory mapped input and output (interacting with peripherals is done through a specified memory-mapped address, MMIO) regions along with any interrupt that occurs during execution before performing a peripheral clustering and dividing the recordings into sub-recordings for each peripheral. They then train a memory model, trying to select and train on known models for each peripheral. The analyst then decides how to introduce inputs into the system.

In summary of the tools that are surveyed (highlighted in bold throughout Section 2 and Section 3) the general-purpose emulators surveyed that have a wide variety of specialties and uses include Avatar^2, CostinFA, Firmadyne, Ghidra, HALucinator, Muench2018, Panda, QEMU, and Simics. Emulators that use hardware in the loop include Avatar^2, SURROGATES, and PROSPECT. Emulators that can use symbolic execution and/or fuzzing include angr, Ghidra, HALucinator, and P^{2}IM. The only emulator that uses machine learning for models is Pretender, which at the moment to be successful requires fairly simple firmware. If the reader is anxious for a flow-chart of which tool to use, then refer to Figure 4 that is explained in Section 10. While this is not an exhaustive list of tools or methods, as there are many more available including References [17, 18, 33, 40, 48, 48, 60, 75, 76, 82, 86, 87, 92, 94, 99, 136, 139, 155, 156, 164, 167, 173] among others, we believe the scope of tools and their applicability is encompassed in the tools mentioned above.

4 EMULATION COMPARISON AXES

Prior to discussing the core challenges of emulation, we first introduce axes over which the comparison of different emulators can be compared. Various emulators and emulation techniques are used for different purposes and thus make different design decisions. Establishing common axes to evaluate emulators is necessary to enable the practitioner to do a useful comparison before choosing the ideal emulation tool for their use.

4.1 Emulation Techniques

A significant challenge to re-hosting firmware is the large expanse of hardware peripherals that need to be emulated; thus, one evaluation axis is the fundamental technique the emulator uses to provide these peripherals. The technique employed will directly relate to another axis of comparison—the complexity of the hardware that is feasible to emulate. Some systems are simple with no peripherals, whereas others may be connected to Remote Terminal Units (RTU), Programmable Logic Controllers (PLC), Field-Programable Gate Array (FPGAs), multiple sensors, databases, Human Machine Interface controllers (HMI), and so on. The amount of hardware the practitioner wants to emulate will range from a single chip or sensor, all the way up to the entire large-scale system. The amount of hardware and complexity of the hardware emulation is largely limited by the peripheral emulation technique used.

The main peripheral interaction techniques used include hardware in the loop (HITL), *learning*, *fuzzing*, and *abstraction replacement*. HITL will use the emulator to perform instruction execution, but if accesses are made to hardware peripherals they are forwarded to the actual hardware. *Learning* refers to using machine learning to provide hardware interactions, whereas *fuzzing* will use random generation to provide simulated hardware interactions. *Abstraction replacement* provides peripheral hardware functionality by identifying software abstractions within the firmware and replaces execution of these abstractions with its own implementations. Examples of surveyed works that allow HITL are SURROGATES [90] and Avatar^2 [112]. P^{2}IM [57] uses fuzzing, and Pretender [72] uses Learning, whereas Firmadyne [25], CostinFA [43, 44], and HALucinator [35] use abstraction replacement.

4.2 Types of Systems

In addition to how hardware peripherals are provided, it is important to consider the type of system the emulator is designed to support. The range and capabilities of embedded systems ranges from large multi-processor systems running customized versions of desktop OSes (e.g., Linux) to low cost, low power micro-controllers running a few KB of code without an OS. The challenges and techniques in emulating these systems vary and may or may not translate from system to system. We reuse the classification presented by Muench et al. [113], though use our own names instead of numbers for the types of systems, splitting the embedded system types into three different classes, based on the type of firmware they execute.

General Purpose Embedded Systems (GPES): Also known as Type 1 embedded system, use a general-purpose operating system that is primarily used on servers and desktop systems. Examples include real-time Linux, embedded Windows, and Raspberry Pi. The operating systems are retrofitted for the embedded space but retain many desktop level features, but with stripped down components, and are coupled with lightweight user space environments such as busybox or uClibc. Tools such as Firmadyne and CostinFA require the embedded system they work on to be Linux based systems and will only work on this type of system. Emulating these types of systems greatly benefits from the work done to enable emulation of desktop software and operating systems (e.g., QEMU directly supports emulating the Linux Kernel).

Special Purpose Embedded System (SPES): Type 2 devices from Reference [113] use operating systems specifically developed for embedded systems. They are often commercial products and closed source. Examples include μClinux, ZephyrOS, and VxWorks. These systems are usually single-purpose electronics or control systems. Some of the features that distinguish these systems are that the OS and applications may be compiled separately and the system is not derived from a desktop operating system. Thus, many emulation techniques from the desktop space do not work, and emulation must start from scratch. Re-hosting these systems requires re-hosting both the kernel and user space. Also adding to the challenge of emulating these systems is the fact that the separation between the Kernel and user space is often blurred.

Bare-metal Embedded Systems (BMES): Type 3 devices are embedded systems without a true OS abstraction that we refer to as bare-metal embedded systems (BMES). They often do not have an OS or may include a light-weight OS-Library. An example is an Arduino system. In both cases, the application will directly access hardware and the OS (if present) and applications are statically linked into a single binary. Recent work [35, 57, 72] focuses on re-hosting these systems.

We find this axis of comparison useful as it helps to determine what emulation techniques an analyst should consider for a given firmware. However, in practice, classifying a system is not necessarily cut and dry. Rather the classification is a continuum on the embedded space. For example, an embedded system that started out using UNIX OS 30 years ago may have originally been classified as a GPES, but as the system morphed over time, the same system currently may now be better classified as a SPES.

4.3 Fidelity

Introduced in Section 2, *fidelity* is perhaps the most important comparison axis but also the most difficult to quantify. The difficulty comes from limited ability to inspect the internal state of hardware, and is further complicated by the ability to compare states. In an effort to enable better understanding of fidelity, we classify fidelity along the conceptual axis of *execution fidelity* and data or *memory fidelity*. This enables comparison of the conceptual limits of fidelity on a two-dimensiona (2D) plane. Work by Costin [44] has a general classification of emulators that have kernels and applications. The classification we present here is more general, applies to both re-hosting and emulators, has more classification points, and is applicable to all types of systems—GPES, SPES, and BMES. The fidelity classifications are from the perspective of the firmware (software), and whether the emulation "looks" and "acts" like real hardware. This implies that we do not need to differentiate between memory (whether DRAM, SRAM, Flash, etc.) that looks the same to the firmware, we just refer to this as internal memory. If there is another driver required to use specific memory (such as an SD card), then we consider this external memory.

Execution fidelity describes how closely execution in the emulator can match that of the physical system. We bin techniques into the categories *BlackBox, Module, Function, Basic Block, Instruction,*

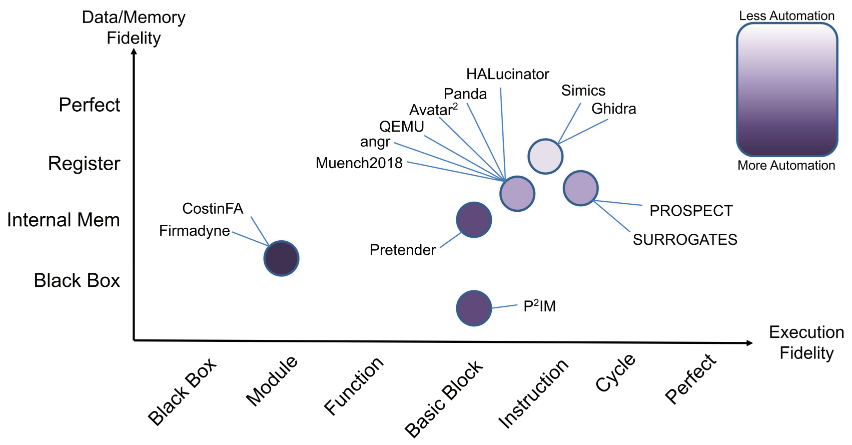

Fig. 1. Categories of fidelity.

Cycle, and Perfect, with execution fidelity increasing from BlackBox to Perfect. A system emulated with *BlackBox* fidelity exhibits the same (or sufficiently similar) external behavior as the real system, but internally may or may not execute any of the same instructions a real system would execute. Module fidelity provides fidelity at the module level. For example, Firmadyne replaces the original firmware's kernel with its own to enable re-hosting the original firmware. Thus some modules of the firmware are executed unmodified, and others completely replaced. Function level fidelity accurately models the system at the function level (e.g., HALucinator replaces entire functions to enable emulation). Similarly, basic block and instruction level fidelity accurately emulate at the basic block and instruction level layers. Beyond instruction level is cycle level, which faithfully emulates to cpu instruction cycle (e.g., the gem5 [12] simulator). Perfect emulation means that emulation is exactly the same as it would be on the actual hardware, which to our knowledge no current emulator achieves.

We categorize data/memory fidelity increasing from the coarsest granularity to finest granularity as BlackBox, Internal Memory, Register, and Perfect. BlackBox fidelity means that the data externally visible to the system or external memory (e.g., HDD or SDD) is the same (e.g., for a given input we get the same output). Internal Memory means that the internal memory (e.g., RAM) is consistent with hardware for a given point in execution. Register level fidelity means that both internal memory and registers are correct at the given execution fidelity and perfect means that all memory components work the exact same as the given system at the level of execution fidelity needed. In Blackbox, Internal Memory, and Register Memory levels, these classifications are usually for specific areas of interest in the firmware. This means that there is a blur between the classification points, as some sections of the firmware may be at the Blackbox level where the user does not care much about the internals, but in a few sections of high interest the firmware emulation may be at the Internal or Register Level fidelity. The Perfect Emulation level is on the scale but currently is virtually unobtainable. Depending on the practitioner, Perfect Emulation level can mean Register level throughout the entire firmware, or it could mean everything is exact, down to the cache for every execution cycle. In Figure 1 we show how the most prevalent firmware emulation techniques fit into this classification framework.

4.4 Purpose of Emulator

Of the surveyed research works in firmware re-hosting, the main focus points have been *Creating Emulators, Dynamic Analysis, Static Analysis*, and *Fuzzing*. Each of these focus points enables the emulator to answer specific questions. The purpose of emulators include vulnerability detection, enabling running legacy code, hardware replacement, development assistance, and system behavioral analysis. The purpose of the emulator directly influences the techniques employed, types of systems the emulator will work on and for, and determine the fidelity of the system.

4.5 Level of Control

Another axis for comparison related to the purpose of the emulator is the ability to control the exploration in firmware and what the tool can/should be used for. Control may perhaps be thought of as another axis in fidelity, quantifying whether you can control emulation and what is actually executed during emulation. It also refers to the level of interaction available to the practitioner. For example, HALucinator enables interactive emulation making it suitable for building virtual testbeds, whereas P^{2}IM enables fuzzing and would not be a feasible tool for testbeds.

For visualization ease we do not combine fidelity and exploration, but it is important to note that some tools and emulators do not allow for controlled exploration. For example, if your emulator is just fuzzing everything from memory to inputs, then it may have very high execution fidelity and low memory fidelity, but it has almost no exploration customization to specify what to execute, it solely relies on a random generator. Randomization and fuzzing enable high coverage fuzzing and vulnerability discovery, but it does not give a clear picture of actual real possibilities of execution.

5 CLASSIFICATION OF SURVEYED WORKS

Now that we have discussed different axes of fidelity, we further categorize and compare surveyed works in firmware re-hosting and emulation. Figure 1 shows a 2D space of fidelity categorization. On the horizontal axis is execution fidelity and on the vertical axis is data fidelity. As can be seen in the figure, there is not a single point for each category but rather a blurry bubble, emphasizing that fidelity categorizations are conceptual and a continuum from Blackbox to Perfect fidelity. As can be seen in Figure 1, the points on the plot range from dark to light shading. The darker the circle indicates that the tools is more automated, requiring less user interaction and requirements for setting up the tool. The lighter the circle in the plot indicates that the default tool requires more user interaction to get the tool setup. We purposefully do not give concrete numbers to the automation or the fidelity levels, rather we visually categorize automation and place fidelity in regions to re-emphasize that these categorizations are not concrete and may slightly move/change depending on tool use and the practitioner's opinions.

5.1 Hardware in the Loop

As various works modify a core emulator, the fidelity can be improved or reduced. When the fidelity is improved there is often a tradeoff made in performance or complexity. SURROGATES uses QEMU as a base emulator but adds specialized hardware to enable faster communication with real hardware. Specialized hardware allows for higher execution fidelity as the peripheral access is then perfect. HITL also increases the data fidelity, as there is no generalization for the peripheral model as is the case for PROSPECT. However, specialized hardware and HITL in general increases the complexity and cost of performing emulation while reducing scalability, as dedicated hardware is needed for each emulated system.

5.2 Instruction Level Execution Fidelity

We categorize Simics and Ghidra Emulator as instruction level execution fidelity while also having register level data fidelity because of their sub-instruction execution. While the coloring of the automation is the lightest of the works surveyed, there are ready made implementations of both of these emulators that can be copied or used out of the box to make the automation much closer to no interaction required. Yet, we color the automation at this level because of the tools defaults and to show the open ended-ness of using one of these emulators.

5.3 Basic Block Level Execution Fidelity

As mentioned previously, QEMU has basic block execution fidelity and RAM fidelity as a default on the data fidelity axis. These are the default categorizations, and they can be affected by other tools. Muench2018, angr, Panda, and HALucinator use QEMU as a base emulator without HITL, so in general their fidelity will be less than or equal to that of QEMU.

For any memory interaction, P2IM will do random fuzzing, meaning there is not even Blackbox fidelity for the data axis. Pretender uses machine learning to attempt to provide peripheral software replacement modules, reducing the data fidelity but still has some internal memory correct and at least Blackbox fidelity on the data axis. As these works reduce fidelity (on either axis) they also decrease the amount of effort needed for their emulation. Muench2018 and PANDA perform tracking/recording as mentioned in Section 3. While replaying and replication is enabled, the fidelity is still maxed out at the Basic Block and Internal Memory/Register fidelity because of the QEMU base emulator fidelity.

angr also uses QEMU for the emulation part of the tool, while also using symbolic execution. Symbolic execution is difficult to put on the emulation fidelity grid, because it has multiple states as it executes, but by using QEMU for concolic and concrete execution, angr receives the same fidelity as the QEMU base. In contrast, Avatar^2 enables the interaction between virtually any of the tools, allowing the fidelity to be at any of the tools fidelity categorization depending on what is programmed by the analyst, but we show its fidelity point defaulting around QEMU, angr, PANDA, and HALucinator, because they are default tools that are easy to get running in the Avatar^2 framework.

5.4 Module Level Execution Fidelity

Firmadyne and CostinFA lose both execution and data fidelity from their base QEMU emulator as they only extract the filesystem and run the code through their own kernel. However, by doing this they drastically increase the automation of their tools, making mass analysis of firmware more scalable.

6 QUESTIONS AND CHALLENGES

The remainder of the article focuses on you, the analyst, wanting to re-host a firmware and discusses the challenges that are encountered, along with techniques and tools that can be used to address them. The challenges and tools surveyed provides a reference for the average practitioner or a starting point for new researchers in the area of firmware re-hosting and system emulation.

6.1 Questions of Purpose and Value

Each emulation tool has requirements that must be met before being able to re-host firmware. Some tools will bypass common emulation challenges with the technique they use. Ideally, the tool will overcome challenges automatically; otherwise, the practitioner must do so manually. Common preconditions are discussed in Section 7; however, this discussion covers the challenges faced and not necessarily why they are encountered.

Before deciding to emulate a system or re-host a firmware, an analyst has a question they want answered. The idea of why to emulate a system is key to building emulators and is often not emphasized in emulation papers. When an analyst wants to find vulnerabilities, there are vulnerabilities that may be discovered at each fidelity level. It is therefore important to know what types of vulnerabilities you are looking for, e.g., is it a firmware logic error that can be detected by just correctly emulating the memory, or is it an instruction or hardware bug that needs higher execution fidelity?

Before emulating a system it is also important to do an analysis on the estimated cost and value-for-money evaluation of getting the emulation to work correctly and the tools to be used to perform emulation. To emulate some systems it may take a small team of engineers 6 months to a year to build and test for the desired system. This may be a bargain or may be too expensive. Analyzing the tools available and how to use them can speed up or even change the traditional approach of building an emulator (manually reverse-engineering the hardware, re-hosting firmware until failure and incrementally adding functionality).

6.2 Key Research Questions

We wish we could say that you should research one specific area, but for every solution/tool that currently exists, there are shortcomings. To understand what is *key* to your research, it is necessary to understand what you would like to solve with emulation. The key area we see that needs addressed is speeding up the time to emulate a system—this includes tools from all the subsequent sections to overcome the challenges. A script written for one tool may work to address a specific challenge, but that script needs to be ported if using another tool, e.g., a script for Ghidra to help find the entry point will not necessarily work with angr or Simics.

This *key* research area of speeding up emulation really encompasses research across all types of embedded systems (GPES, SPES, and BMES) and includes new ways of overcoming problems presented in Section 7, Section 8, and Section 9. Any tool or technique that more efficiently addresses the challenges is a useful area of research.

6.3 Challenges

Now that we have introduced surveyed works and classification criteria, we present some of the core challenges faced during firmware re-hosting and system emulation. During the emulation pipeline, various challenges are encountered and we split these challenges into:

- Pre-Emulation
- Emulation
- Post Emulation

Pre-Emulation are challenges that are pre-requisites to emulation execution and overcoming these challenges enables executing the first instruction in the emulator. These challenges include obtaining firmware, unpacking the firmware, and gaining and understanding how to configure the emulator to re-host the firmware. Once the first instruction is executed in the emulator, we consider the Emulation stage to have been started; however, as execution progresses, greater understanding of the firmware is obtained, which leads to refinement of the emulator configuration and implementation. Thus, we break Emulation into Emulator Setup (challenges that enable further refinement of the emulator and more complete execution) and Emulation Execution (challenges fundamental to emulation itself). Finally, after emulation there is the Post Emulation stage where the execution is analyzed and validated. The challenges presented here have been mentioned in part and used as the foundation for multiple industrial and academic works, but we present them in entirety here for completeness.

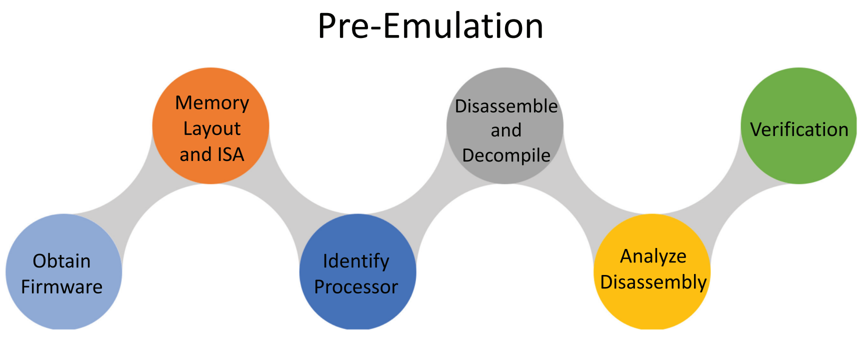

Fig. 2. Categorization and flow of some of the steps required for system emulation during Pre-Emulation.

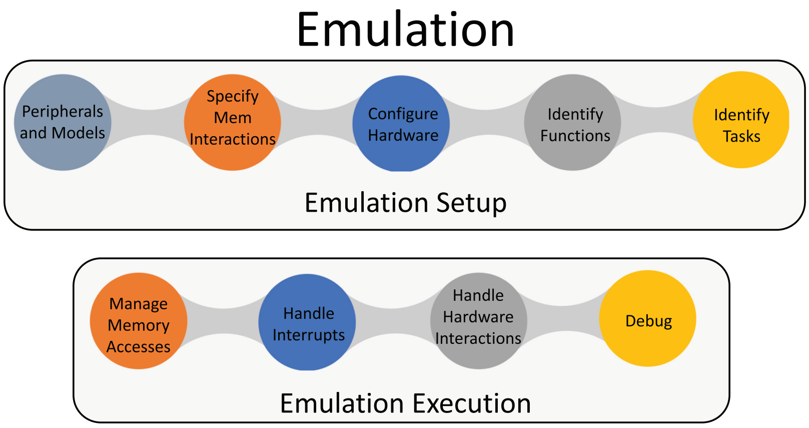

Fig. 3. Categorization and flow of some of the steps required for system emulation during Emulation Setup and Execution.

In the tables throughout the rest of the article, we survey different techniques that attempt to overcome the various challenges faced during emulation and binary analysis. If there is a check-mark, then the tool attempts to address the issue; but, it does not necessarily solve the problem and some tools work better than others, though we do not specify our opinions on the matter. If there is a dash, then that means the tool bypasses the challenge by the technique they employ. If there is no mark or dash, then the tool does not address the challenge. In some cases the tools have prerequisites to work correctly. The emulator may require the analyst/user to figure out some of the challenges manually and pass the solution to the emulator to get past the challenge.

We provide some of the flow and common challenges faced during emulation in Figure 2, and Figure 3 though the figures do not encompass all the challenges that may be faced, with some challenges left off the figures all together for brevity. These figures are general flows and each bubble represents one or more challenges encountered through the process. The different challenges are further addressed in detail in Section 7, Section 8, and Section 9.

7 PRE-EMULATION

In Figure 2, we show the flow that a practitioner will usually explore during the Pre-Emulation stage of re-hosting a firmware. With each step indicating challenges that need addressed before continuing to address the next. The challenges presented here are similar to those presented by Costin [43]. Here we focus on the challenges of emulating and analyzing a single firmware, whereas Costin focuses on challenges for acquiring and analyzing thousands of firmwares available to download from the Internet automatically.

Before re-hosting a firmware, an analyst or practitioner will usually have a system they want to emulate or a firmware they want re-hosted. In some cases, the system architecture is unknown or the analyst may not even have the firmware (e.g., when working on a bug bounty or proprietary hardware). Even after obtaining the system or firmware in mind, key information must be identified prior to re-hosting a firmware. Steps and challenges prior to setting up the emulator is what we refer to as the Pre-Emulation stage. This stage may also include verification of information and understanding gained prior to actual emulation (i.e., Disassembly, Initial Analysis, and CFG Recovery), though verification is not strictly necessary. We note that verification of correctness (which can include formal verification and/or behavioral verification) is useful when challenges in subsequent phases are encountered, as it narrows down where certain problems stem from.

The information required to begin emulation varies by emulation technique but includes obtaining firmware, determining memory layout, figuring out the ISA, identifying the processor, analyzing the binary, lifting/disassembly of firmware, and an initial firmware analysis. As a quick reference to the practitioner, different techniques that attempt to address the challenges present in Pre-Emulation are summarized in Table 1.

One of the main tools used to address these challenges is binwalk [139]. Binwalk is mostly used for extraction of the content from firmware images, but has other features that are useful. It can try to determine ISA (not necessarily the processor), extract files from a blob, do a string search, find signatures such as common encryption formats, disassemble using capstone [22], calculate entropy, and perform binary diffing. Some of the other tools are mentioned in the subsequent subsections. A simple example of reverse engineering from a blob that mentions some of the challenges below was presented at BlackHat 2013 [41]. They present an end-to-end unpacking of firmware with some of their formats that may be helpful to reference for a newcomer in the field. Another helpful reference for IoT firmware based on OpenWrt is maintained by OWASP. They provide a platform to educate software developers and security professionals on vulnerabilities in IoT devices [121].

7.1 Obtaining Firmware

The first challenge to re-hosting firmware is obtaining the firmware. In the simplest case it can be downloaded from the vendor's website. If not directly available from the vendor, then third-party sites, such as Github, have firmware available that has been used for academic research. Other ways to get firmwares include obtaining example firmwares from development boards that can be compiled with various operating systems and toolchains.

Downloading the firmware is not always possible, and even when possible, the firmware can have embedded proprietary file formats that are not easily extracted. For example, files can be compressed or contain firmware for multiple architecture files with final compilation of the firmware being done on the hardware during boot-loading. In some cases, firmware is combined with the operating system, such as the case of BMES or SPES types of systems, whereas for others downloaded firmware is only the user level application and the operating system kernel also must be obtained separately to perform full-system emulation as is sometimes the case for GPES. To overcome embedded unpacking issues, sometimes a network capture tool (e.g., References [31, 53, 55, 88, 106, 116, 117, 125, 130, 163, 165, 179]) connected to the actual hardware may be used to capture network traffic during a firmware update. The firmware is then extracted from the captured packet payloads.

In addition to downloading, firmware can sometimes be extracted from hardware. Vasile et al. [170] in their survey of hardware-based firmware extraction techniques showed a high percentage

Table 1. Pre-Emulation Challenges

Paper/Technique	Obtaining Firmware	Determine ISA	Finding Base Address	Finding Entry Point	Determine Memory Layout	Identify Processor	Disassembly/ Recover CFG
angr [151]		✓	✓	✓		✓	✓
Avatar^2 [112]							✓
CostinFA [43, 44]		✓	–	–	–	–	✓
Firmadyne [25]	✓	✓	–	–	–	–	–
Ghidra [119]		✓	✓	✓	✓	✓	✓
HALucinator [35]				–			✓
Muench2018 [113]							✓
P^{2}IM [57]							✓
PANDA [50]							✓
Pretender [72]							✓
PROSPECT [84]							✓
QEMU [11]							✓
Simics [105]							✓
SURROGATES [90]							✓
ARMX [146]		–	–	–	–	–	–
BANG [75]		✓	✓	✓	✓	✓	✓
BAT [74]		✓	✓	✓	✓	✓	
Binwalk [139]		✓	✓	✓	✓	✓	
CLIK on PLCs [82]							✓
Datasheets		✓			✓	✓	
Dtaint [26]							✓
Dytan [33]							✓
FIE [48]						–	✓
Firmalice [150]		✓	✓	✓			✓
firminsight [59]	✓	✓			✓		
firmware-mod-kit [60]		✓	✓	✓	✓	✓	
firmwaredb [172]	✓						
FirmUSB [76]						–	✓
HumIDIFy [167]		✓	✓	✓	✓	✓	
ICSREF [86]		✓	✓	✓	✓	✓	✓
IDA-PRO [136]		✓	✓	✓	✓	✓	✓
Inception [40]							✓
KLEE [19]							✓
OFFDTAN [177]							✓
PIE: Parser [36]							✓
Radare2 [164]		✓	✓	✓	✓	✓	✓
Scraper (python)	✓						
subzero	✓	✓					
Spedi [87]		✓	✓	✓	✓	✓	✓
strings (linux command)		✓				✓	

✓means the tool attempts to address the given challenge. "–" means the tool bypasses this challenge, usually by the emulation technique used. A blank means the tool does not address the challenge. The first 14 entries in the table are bolded and are the main tools profiled in detail throughout.

of systems expose UART interfaces that are sufficient to obtain firmware dumps. In addition to UART ports, debug ports (e.g., JTAG), and USB ports can be used to dump firmware [56, 186]. On some devices, a physical acquisition my be achieved by using the flash memory read command after reverse engineering the firmware update protocol in the bootloader [182]. If these ports are locked and secured, then another option is to remove the memory from the circuit board and connect to another system to dump the firmware. Removing memories from the printed circuit boards comes at considerable risk of damaging or destroying the hardware. Using debug ports requires the debug port be present, and unlocked, both of which are increasingly considered poor security design.

Of the tools we surveyed, Firmadyne and CostinFA obtained firmware from vendor websites, whereas P^{2}IM, angr, and Pretender used firmware available from a third-party vendor (Github). HALucinator and Pretender also used development examples from real embedded boards in their evaluation. For the case of Simics, QEMU, and Ghidra they are base emulator tools and not academic papers, hence they do not specify how to obtain firmware, expecting the user to have a firmware before using the tool. The other surveyed works did not specify how their firmware samples were obtained, just what the firmware was, or what the system was that they did emulation for.

The authors of Firmadyne developed and released a scraper that crawls embedded system vendors websites and downloads any firmware that they can identify. They then unpacked the firmwares in a generic way and if the firmware unpacked correctly, Firmadyne would continue with emulation, otherwise it would crash. Of the 23,035 firmwares Firmadyne scraped, they extracted 9,486 of them. Of those extracted, 1,971 were successful in naive emulation. The large percentage of failure (90%) shows obtaining and extracting firmware is a real challenge and is difficult to overcome in many circumstances. CostinFA posted URL publicly that would try to unpack and analyze a firmware [42]. CostinFA collected an initial set of 759,273 files scraped from publicly accessible firmware update sites, and filtered that down to 172,751 potential firmware images. A sampled set of 32,356 images were then analyzed and 38 vulnerabilities were discovered [43], though it is unclear the level of emulation or re-hosting involved. Other resources that may be helpful in obtaining firmware include Python scraper tools and other open source repositories [59, 131, 142, 172].

7.2 Instruction Set Architecture

After obtaining the firmware, it is necessary to determine what the ISA the firmware uses so that the emulator can disassemble the firmware into the correct machine instructions, endianness (e.g., little- or big-endian), and word-size. In addition to determining the ISA family (e.g., ARM, PowerPC, X86, MIPS, ARM64, AVR, etc.), sometimes the ISA version is needed to correctly disassemble instructions, e.g., is it ARM with Thumb support and floating point instructions or not?

The ISA can most commonly be determined from a datasheet of the processor being emulated. If the hardware is unknown and a datasheet cannot be identified, static analysis techniques can be used. These techniques first try to determine if the file format is of a known file format (e.g., ELF, PE2, Mach-O) using the file utility, then looking at other signatures in the firmware (e.g.„ encryption, compression, etc.), or analyzing strings in the binary to guess the ISA. One well known tool that may help determine the ISA is binwalk [139]. Binwalk will use the capstone disassembler and try disassembling the binary for various types of ISA. If there are more than a specified number of instructions in a row (default 500) of a given architecture, then that is a strong candidate for the ISA. Of the tools surveyed, Firmadyne and CostinFA use binwalk and existing extraction tools to determine the ISA and then extract the filesystem, or extract the filesystem first and then determine the ISA—with the order of these two steps being highly specific to the firmware and analysis process at hand. Ghidra has headless scripts that can be run to try and determine the ISA (including one using binwalk), whereas angr provides the *boyscout* [5] tool that will try to determine the architecture and the endianess of the firmware. The other surveyed tools expect the ISA to be given to work correctly.

There is also recent research on using machine learning to classify ISA and endianness. These techniques usually rely on doing a binary similarity detection. They will use known architectures for training binaries, decomposing the binaries into smaller comparable fragments and transforming them to vectors to work with machine learning and stochastic gradient descent optimization methods. One such work by Clemens et al. [34] experiments on 16,785 different firmwares from 20 different architectures, and was accurate in classifying 17 of the architectures over 99% of the time, and the remaining third architectures (mips, mipsel, cuda) 50% of the time. De Nicolao et al. [49] leverages supervised sequential learning techniques to locate code section boundaries of binary files to help ease difficulty in analysis using a similar sized database for learning. Kairajärvi et al. [81] uses the ideas from both Clemens [34] and Nicolao [49] with improvements while using and publicly releasing a much larger and more balanced database for learning and testing. Other machine learning techniques that could be modified to determine the ISA are mentioned in Reference [181] along with other machine learning techniques for binary analysis. These techniques require large datasets for training to be accurate and will only work as well as their training data.

If the above automated methods fail, then the practitioner can attempt to manually identify the ISA by looking at signatures and strings in the firmware (certain compressions, signatures, copyrights, etc.) and/or brute force decompilation and performing an "eye" test on what looks promising.

7.3 Determine Base Address

In order for the firmware to execute in an emulator, it must be loaded at the correct address(es). Determining the base address is difficult if the firmware being re-hosted is a binary-blob (e.g., just a binary with no symbols or metadata). The base address where the firmware should be loaded can sometimes be found by hardware datasheets (for firmware executing from internal memory). If source and compilation tools are available, then base address information can be found in linker scripts.

Finding the base address is fundamental to many binary analysis techniques, and thus techniques to (try to) automatically determine it have been researched. Zhu et al. use strings and LDR instructions, comparing them and matching the offsets to determine the image base of the firmware [187, 188]. This is similar to the technique of listing the dwords occurring in a file with the list of strings in a file and lining up the distances between occurrences if possible. If there is a match, then subtracting the offsets will give you the image base address.

Firmalice [150] leverages jump tables in the binary by analyzing the jump table positions and the memory access pattern of indirect jump instructions. In a jump table there are a set of absolute code addresses that can give a better idea of where the firmware needs to be located at for the absolute addresses to work correctly. To find the jump tables, they scan through the binary for consecutive values that differ in their least significant two bytes. Finding the jump table is successful in many cases as jump tables are typically stored consecutively in memory. After finding the jump table, they then analyze all the indirect jumps found in the disassembly phase and the memory locations that they read their jump targets from. The binary is then relocated so that the maximum number of these accesses are associated with a jump table.

angr also attempts to determine the base address with their analysis script called *girlscout* [6]. This script will try to decide the base address by looking at functions and doing a statistical analysis to vote on the most likely base address. If automatic techniques do not work, then the base address may be discovered by brute force guessing and checking, as most base addresses are multiples of powers of two.

In the case of Firmadyne and and CostinFA, the filesystem from the firmware is extracted after determining the ISA, and then the filesystem is used in a custom kernel given to QEMU. By using their own kernel, these tools bypass the need to determine the base address, but it also reduces the execution fidelity, as it does not execute the original kernel. For the other surveyed tools it is assumed the base address is provided by the practitioner.

7.4 Finding Entry Point

After determining the base address of the firmware, the practitioner needs to determine the entry point (i.e., address from where to start execution). Entry point information can be encoded in the binary (e.g., Executable Linker Format (ELF) defines the entry point in metadata), or different analysis can be run to help give the practitioner entry point options.

For binary-blob firmware, angr, Ghidra, and IDA [136] have scripts that scan through the binary attempting to find function prologue instructions and function returns. From function information a directed function call graph can be generated and analyzed. Any root node of a weakly connected component in the call graph can be treated as a potential entry point. Function call graph creation and analysis requires that you already know the ISA (so the function prologue, function epilogue, and call instructions can be analyzed). The call graph technique often returns multiple entry points, from which the correct entry point for emulation must be identified. Additionally, firmware will often have multiple valid entry points (e.g., bootloader and interrupt service routines).

Instead of looking at the firmware to identify the entry point, some techniques rely on knowledge of the hardware they support to determine the entry point. For example, HALucinator targets ARM Cortex-M devices. The Cortex-M architecture defines that the initial program counter's value be stored at address 0x4 on reset. Thus, HALucinator finds the entry point by looking at address 0x4. If the hardware is known, then the datasheet will likely provide information about how the system begins execution.

For techniques that use a replacement kernel, as is the case for CostinFA and Firmadyne, finding the entry point challenge is essentially bypassed, as the entry point of the kernel is known, because they built the kernel themselves. If the practitioner is dealing with a known operating system or compiler toolchain, then the entry point function name may be specified (such as _start or _init). There are then techniques that can do function matching giving you the actual entry point. One such example is VxHunter [122] that will work for many firmwares with the VxWorks OS and compilation toolchain.

7.5 Determine Memory Layout

Determining the memory layout enables configuring where in the processors layout different types of memory are located. It sets the address for RAM, Flash, and MMIO. If the system is available with specifications, then usually the memory layout can be determined from datasheets. For ARM systems, there may be a CMSIS-SVD file that also defines the memory layout. These files are in a specified format that can be loaded into a reverse engineering (RE) tool such as Ghidra or IDA and an automatic analysis can be run to update the memory layout [97]. Other types of files that sometimes specify memory layout include EDC files and hwconf files.

If no documentation exists and RE tools fail, then physical examination of the components on the device that is being emulated may help to determine memory layout. Often markings and manufacture printing on the parts can be used to identify datasheets for the parts. If the memory layout is still unknown, then the emulator can be over provisioned with memory (e.g., giving significantly more memory than the physical system) and trial and error must be used to determine where code is located and where external peripherals are, then these interactions are usually mapped into memory mapped IO. Of the surveyed tools, memory layout must be specified by the practitioner, though in some cases tools make it easier, e.g., Ghidra when CMSIS-SVD file is present.

7.6 Identify Processor and/or Board Support Package

Depending on the fidelity the practitioner is targeting, often times identifying the exact processor is not necessary. As the case is with QEMU, the emulator only works off the ISA level features. For fidelity at the cycle level of emulation, tools such as gem5 [12] require knowing the exact processor as the same ISA instruction can be implemented differently at the cycle level for various processors. In QEMU, if a versatile machine is not used, there can also be errors, even if the ISA specified is correct.

In the case that the practitioner needs to solve the challenge of identifying the processor, solving memory layout may help. However, if the processor is known, then often the documentation for the processor will specify the memory layout. If you have access to the hardware, then most likely the processor will be labeled, solving the challenge.

If you do not have access to the hardware, the practitioner could do an aggressive instruction finding, followed by an analysis on how the instructions interact with memory. This analysis can then be compared against analysis on known processors to narrow down the processor candidates. This is still a manual process for the practitioner to then select and test different processors. If the vendor is known, projects that leverage known information from other reversed firmware may be used [138]. If the previous analysis are unsuccessful, the practitioner could run a brute force script to test all available processors in the base emulator. If still not successful, access to labeled hardware is required. For surveyed works based off QEMU, identifying the processor is not strictly necessary in all cases, though if the QEMU target is not versatile it may be required.

7.7 Disassembly, Initial Analysis, and CFG Recovery

As mentioned at the beginning of Pre-Emulation, disassembly, initial analysis, and recovering the Control Flow Graph (CFG) are not strictly Pre-Emulation challenges. However, it is important to verify that the work the practitioner has performed and challenges addressed to this point in time are correct. Verification of previous work is perhaps skipped in some pipelines, but we have found that verification before continuing on to the next stages will save time and reduce headaches in the Emulation phase, e.g., when multiple possible entry points (which may occur when there is a bootloader in addition to the main firmware under analysis), doing a validation and verification that you have the right entry point is worth the extra time to save the practitioner from wasting time re-hosting the bootloader that may not answer the emulation question at hand.

In the case that the practitioner has an idea of what the firmware is doing, such as is usually the case when the hardware is known, analyzing the CFG can help determine if the previous challenges such as base address, ISA, entry point are correct. When the CFG is recovered, it also conveys how well the disassembler and decompiler performed. This will give an idea of how successful emulation execution will be and/or the fidelity of emulation moving forward. Of the works surveyed, there are relatively few core disassemblers. angr uses capstone for parts of its disassembler, whereas QEMU and Ghidra use C/C++ to implement their disassembly. The other surveyed tools use a base emulator that will reuse one of these disassembly implementations. If the disassembler works perfectly, then the instructions have been disassembled correctly and the CFG can be recovered trivially if the control flow does not leave the main processor.

If the control flow leaves the main processor and goes to a co-processor (e.g., GPU or DSP) chip for initialization, then it is near impossible to recover the CFG for what occurs in the co-processor. The practitioner at this point can try to determine what state and memory changes occur by comparing the state of memory before and after control is passed to the co-processor. If control flow leaves the emulated processor, then fidelity of emulation is limited for that portion of the re-hosting, but in most cases is not a limiting factor in emulation.

If the hardware is not known and the practitioner does not have any idea of the control logic the firmware is trying to perform, then disassembly and analysis can still be of use for validation. Disassembly will ensure the correct ISA is known and possibly help validate the processor. A control flow graph can still be created and analyzed to determine if there are valid flows throughout the graph, giving at least a weak reassurance that the base address and entry point are correct.

Disassembly and analysis are iterative, meaning the practitioner will perform the disassembly, then analyze the results in iterations. Between iterations, they will then modify the inputs and parameters to the disassembly slightly depending on the analysis results. This process will continue until the analysis results match what is expected from the practitioner. During this iteration process, there will be tweaks to the ISA, processor, base address, entry point, memory layout, and so on. To solidify the ISA, processor and memory layout, the practitioner may analyze multiple firmwares for the same system concurrently, aggregating the analysis results to solidify results.

8 EMULATION

In Figure 3, we show the flow that a practitioner will usually explore during Emulation setup and execution stages of re-hosting a firmware. The stages will usually be interspersed during iterations of emulator development. These challenges generally are not linear, but rather occur in different orders depending on the firmware being re-hosted.

After the practitioner has determined the processor, memory layout, entry point and base address of the firmware, verifying support in the base emulator is performed. Ideally the base emulator has support for the processor, if not, the practitioner will have to create a new specification for the base emulator. QEMU and Ghidra have instructions on how to add support for a new processor [114, 171]. After a base emulator is available, the next challenges can be addressed.

Overcoming the challenges in pre-emulation enables loading the firmware into the emulator and beginning execution. For the execution to be faithful to the real system, additional challenges must be overcome. We break these challenges into sub-categories: setup and execution. Setup challenges are usually done statically when the emulator is either paused or stopped whereas execution challenges are when the emulator is actually running. As a reference to the practitioner, we provide Table 2, but do not provide a reference table for the execution, as all the surveyed works attempt to overcome the given challenges.

8.1 Emulation Setup

After you have overcome the challenges in Pre-Emulation, now you need to determine how to handle configuration, external interactions and memory. Emulation setup encompasses these problems and is closely tied with the actual emulation execution. Setup is often iterative between execution stages, and with each iteration more knowledge is gained about the firmware and its dependencies on the emulator and peripherals. The emulator is improved and execution of the firmware is also iteratively performed. This continues until the question the analyst was trying to address has been answered.

As mentioned in Section 4.1, the scope of emulation may vary from a single chip or microcontroller to a subsystem or a large distributed system. The scope of emulation that is targeted will affect the challenges faced during emulation. The problems we discuss here are not all encompassing, though we believe it is a sufficient basis to demonstrate major challenges currently faced during the Emulation Setup phase.

Table 2. Emulation Setup Challenges

Paper/Technique	External Hardware and Peripherals	Mem Interactions/ Setup	Configure Hardware	Missing Code	Function Identification and Labeling
angr [151]	✓	✓			✓
Avatar^2 [112]	✓	✓	✓	✓	✓
CostinFA [43, 44]	✓	✓	✓	✓	✓
Firmadyne [25]	—	✓	✓	—	✓
Ghidra [119]	✓	✓			✓
HALucinator [35]	✓	✓	✓	✓	✓
Muench2018 [113]	✓	✓	✓	✓	✓
P^{2}IM [57]	✓	✓	✓	✓	
PANDA [50]	✓	✓	✓	✓	
Pretender [72]	✓	✓	✓	✓	✓
PROSPECT [84]	✓	✓	✓	✓	
QEMU [11]	✓	✓	✓	✓	
Simics [105]	✓	✓	✓	✓	
SURROGATES [90]	✓	✓	✓	✓	
ARMX [146]	✓	✓	✓	✓	✓
Datasheets	✓	✓			
Firmalice [150]	✓	✓			✓
ICSREF [86]					✓
IDA-PRO [136]	✓	✓			✓
Inception [40]	✓	✓			✓
KLEE [19]	✓	✓			✓
OFFDTAN [177]		✓			
PIE: Parser [36]	✓	✓			✓
Radare2 [164]	✓	✓			✓
Spedi [87]	✓	✓			✓

✓means the tool attempts to address the given challenge. "—" means the tool bypasses this challenge, usually by the emulation technique used. A blank means the tool does not address the challenge. The first 14 entries in the table are bolded and are the main tools profiled in detail throughout.

8.1.1 Peripherals, External Hardware, and Modeling. Handling peripheral accesses is where a large amount of research is currently focused. Because of the variety of peripherals and vendors, it is likely the peripheral that is being accessed by the firmware is not implemented in the base emulator. Handling External Hardware and Peripheral interactions encompasses how to handle or represent interactions between the emulator and the peripherals. As mentioned in Section 7.5, statically determining the memory map will specify where the peripherals are located, not if they are used or when they are used. Dynamic analysis and execution can provide some of this information, discovering what peripherals are used when valid inputs for the given subset of peripherals analyzed. Depending on the fidelity of the peripheral model, i.e., the extent to which the peripherals are modeled, will constrain the dynamic execution to specific paths. Providing more realistic peripheral models will increase the amount of code executed and result in valid accesses to more peripherals that may not otherwise be accessed. During dynamic execution, if the emulator has an exception or crashes due to trying to access peripherals that are not mapped, then this usually occurs when there is an aliasing problem. Solving the alias problem is sometimes feasible statically, but may have to be solved dynamically during execution by keeping track of various addresses and using this information to update peripheral models for the emulator.

Handling peripheral interactions can be done by emulating the external device, by using the actual hardware (HITL), or by patching the firmware to bypass the interaction. The firmware modification may ignore some of these accesses (such as setting CPU clock rates), always give an increasing value, or read from a file of expected inputs. The necessary operation will partially be determined by the firmware being re-hosted. Using HITL will give the highest fidelity for execution and memory, but will prohibit the parallelization of emulation as it is constrained to availability of the connected hardware. HITL emulation also has significant challenges in synchronizing states between the emulator and hardware. For example, a timer may generate an interrupt on the hardware, causing it to be stuck continuously processing the timer's ISR, while the emulator has no timer and thus is not processing any interrupts, or another example is when the watchdog timer interrupt kicks in while the analyst is slowly emulating or using a debugger (such as GDB).

Providing abstractions requires manual effort and a thorough understanding of what the peripheral is performing, but it does provide high fidelity. Tools that use this approach include QEMU, HALucinator, PANDA, and Muench2018. Using a fuzzer, however, does not require device specific knowledge but may not give sufficient fidelity for many questions you want answered and is usually only useful to help find bugs and vulnerabilities. P^2IM uses the fuzzer approach and is successful with finding some vulnerabilities. Using machine learning, as Pretender does, is appealing as fidelity is perhaps slightly affected but could still give valuable insights beyond simple software bugs. The results of current machine learning approaches only show proof of concept on simple peripherals (e.g., serial/UART port being the most complicated) at this point, and it is still unclear if the approach will work for arbitrary hardware peripherals.

Of the other surveyed tools, Firmadyne and CostinFA assume the peripherals are part of their core kernel, otherwise the emulator will crash. They do not provide techniques around such challenges. SURROGATES and PROSPECT will forward peripheral accesses to the actual hardware, bypassing the modeling, but it introduces the state synchronization and delay of emulation challenges.

8.1.2 Memory Interactions and Setup. Most emulators allow the practitioner to specify where data, code, and peripherals are located. Doing so allows the emulator to set restrictions on emulation, such as crashing or notifying the practitioner if data are trying to be executed. Emulation setup will usually allow specifying different memory regions and types of interactions such as where RAM, Flash, and Memory Mapped peripherals are located. This emulation setup can also be described as configuring the emulator, which is necessary to start the emulator. Setup will include providing the information discovered during Pre-Emulation, including specifying the base address and entry point along with the memory layout and amount of memory available.

If a tool does not have built in support for memory, then reusing the base emulator support or providing the interaction through software are feasible options for the practitioner. In specialized cases such as SURROGATES and PROSPECT, interactions can be forwarded to the actual hardware. For HALucinator, Ghidra, PANDA, PROSPECT, and Muench2018 it is expected the practitioner will either use the built-in memory handlers or provide modules to allow memory interactions to perform correctly. For Firmadyne and CostinFA the memory and handling of memory is built directly into the kernel, so if there is an error with the memory or interactions that is not a firmware bug, their kernels will need to be modified.

8.1.3 Configuring Hardware. Setting up hardware is a challenge for emulation using HITL. This requires initializing it state such that its can be used, and then bringing it into the loop. To do this the practitioner will need to specify in the emulator how and when to forward interactions to the hardware. This may also require delays or using specialized hardware execute fast enough to be usable, as is the case for SURROGATES or Aveksha [161]. Aveksha is a system for nonintrusive tracing of execution at a high spatial and temporal granularity suitable for an embedded wireless node. Avatar^2 is a powerful tool when configuring hardware, and the provided framework is closest thing to allowing plug and play integration for HITL.

8.1.4 Missing Code. In some cases, as you recover the CFG you may notice you have missing code. Missing code usually occurs when the wrong entry point has been specified, there is patched out functionality that may write code (such as a bootloader), the firmware is not complete (e.g., either a partial firmware-update or unpacking was only partially successful), or if there is code on ROM chips in the device. Missing code is more common if the firmware is ripped off the actual hardware device. In some cases it is possible to patch out the missing code and still obtain the level of emulation fidelity the practitioner cares about, otherwise, missing code may make emulation infeasible.

If the emulator tries to execute missing code, then usually the system will throw an exception or crash altogether. The only technique we are aware of to overcome this challenge is replacing the code with models or skipping the code altogether. HALucinator replaces the code by using function intercept techniques that then allow for replacing such functionality with manually written models. For other surveyed tools, missing code will require some sort of manual intervention to overcome, perhaps manually providing the same functionality as HALucinator.

8.1.5 Function Identification and Labeling. Function Identification is necessary for some emulation techniques but not all. If emulation fidelity is at the module or function level, then the practitioner may want to determine certain functions, such as HAL calls, and provide abstractions for these functions. Function identification is not an easy problem and is the basis of a plethora of papers and is still a very active area of research [4, 8, 21, 29, 78, 93, 100, 109, 110, 132, 133, 149, 180].

Of the surveyed works, angr, Ghidra, and HALucinator have library matching built in to their frameworks. The other works either do not need to overcome such problems or they do not address function identification. angr, Ghidra and HALucinator will use existing techniques to try to identify some functions, such as using IDA FLIRT signatures, loading libraries and trying to match, and compiling existing HALs and comparing the re-hosted firmware to a database. Other techniques include identifying functions from their side effects such as is the case with Sibyl [152].

8.2 Emulation Execution

Execution deals with the classical problems of emulation as mentioned in Reference [11], expanding to include problems that are arising with the increasing uses of emulation. The different fidelities of execution will also use different techniques for emulation. Tools that are cycle accurate, such as gem5 [12], will decode the instruction depending on the processor and use the same depth and stages of the CPU pipeline as the original processor when emulating the instructions. Simics [105] has instruction level fidelity and will thus decode the instructions and update state after each instruction. QEMU uses basic block fidelity and does a translation from the basic block instructions of the target architecture to machine host instructions using QEMU's Tiny Code Generator (TCG). Ghidra Emulator will translate instructions to P-code, with each machine instruction translated generating up to 30 P-code instructions. Others will use various other intermediate representations including LLVM IR [95], VEX [115], REIL [54], BAP's BIL [18], Binary Ninja LLIL [173], and more. Of the surveyed tools, angr will translate to VEX. Each of these techniques have their own challenges and tradeoffs, which is why there is still research in this area.

For each of the surveyed tools there is a base emulator. angr, Avatar^2, CostinFA, Firmadyne, HALucinator, P^{2}IM, PANDA, Pretender, PROSPECT, SURROGATES and Muench2018 all use QEMU as their base emulator. angr also uses CLE loader to allow Avatar^2 targets to run concretely in their framework through what they term angr_symbion that combines symbolic execution and concrete execution. Cross tool integration essentially opens the door for any emulator to work with any other analysis tool, symbolic engine, or fuzzer. For traditional execution challenges (the different subsections below), the base emulator will usually address and overcome the challenge. Because the base emulator solves the challenges throughout execution challenges, we do not specify what each tool does for overcoming the given challenge, rather we address more generally what QEMU, Simics and Ghidra Emulator do.

8.2.1 Register Allocation. Each architecture will have different registers and different conventions. For example, the program counter (PC) on different architectures will be different registers such as R15 for ARM, PC for x86/x86-64/PowerPC, R0 for TI-MSP430. Most emulators will map each register to a concrete fixed host memory address or register. Mapping the registers to fixed host memory is versatile and probably the most portable solution and is done for QEMU, Simics, and Ghidra Emulator.

8.2.2 Direct Block Chaining. Block Chaining is directly related to QEMU, as Simics does a sub-instruction level emulation and execution, leaving the chaining to occur naturally with fall through if the instruction does not modify control flow. Ghidra will allow natural fall through, but allows for flow modification in their UI.

If the emulator does a translation, such as QEMU and the translation of entire basic blocks, then the emulator has a simulated program counter that is used to find the next blocks of code to be executed. These blocks are usually cached in memory to speed up execution, so there is a lookup in a hash table to retrieve the correct block. For some emulators, they will add instructions at the end of blocks to directly chain to the next block of instructions to execute.

8.2.3 Self-modifying Code and Translated Code Invalidation. On some CPUs self modifying code is not a problem, as there is a specific code cache invalidation instruction executed when code is modified. On other CPUs though, there may not be an invalidation instruction, so this becomes more difficult. In Reference [11], they handle self modifying code by keeping track of translated code and the corresponding host page as read only. If a write is performed to the code, then they invalidate the translated code allowing for the code to be rewritten. They do some more clever things when using a software MMU, as they do not always have to invalidate the code when only data is changed. Ghidra Emulator has permissions for writing and code is read only. Therefore if code is written the emulator will throw an error and exit. It is not clear how or if Simics supports self-modifying code.

8.2.4 Non-Volatile Memory. Non-Volatile Memory (NVM) are becoming more popular because of the improved latency and power efficiency compared to flash and other hard drives. Traditionally NVM has not been a challenge faced as it is now becoming more prevalent. Because NVM is relatively new, on boards that have NVM, most current system emulators are probably lacking in support. Some vendors give instructions on how to enable an emulated environment where you can build persistent memory (PMEM) applications without having the actual hardware. To overcome the NVM or PMEM challenges in system emulation, you need to determine how the memory interacts and provide a handler for it, much like QEMU does for normal memory. The instructions given by different PMEM hardware vendors on how to emulate their hardware will be crucial during implementation. QEMU, Simics and Ghidra Emulator handle NVM by allocating memory in the host system and having hardware interactions modeled by the analyst [46]. For QEMU, some tools provide some interactions such as NVRAM [58].

8.2.5 Direct Memory Accesses. In some cases co-processors may use direct memory access (DMA) for initialization, or peripherals may write directly to memory. DMA can be addressed by emulating co-processors, using HITL emulation, or if the firmware uses known function calls for DMA, the functionality can be replaced by intercepting and replacing the functionality. QEMU has been expanded by Avatar^2 in software, and SURROGATES using hardware, to allow for forwarding peripheral acesses to the actual hardware. Simics is usually a more custom solution and also allows for HITL with given modifications. HALucinator will handle DMA by intercepting HAL calls that perform DMA and will implement the needed memory modifications with manually implemented functionality.

8.2.6 Handling Interrupts. There are various ways to handle interrupts. Handling interrupts is another active research area as there are multiple ways to accomplish the interrupt handling. The emulator may or may not check at each instruction or translated block for interrupts, with tradeoffs for different implementations. In some cases the emulator may require that the user trigger interrupts, call the interrupt handler, or patch an automatic call at certain parts in the firmware. There are pros an cons to each of these techniques. For the case of QEMU, they do not check at each translated block whether there is a hardware interrupt pending, rather it relies on the user to asynchronously call a specific function and specify that an interrupt is pending. At that point the function resets the chaining of the current executing translated block, ensuring that execution goes to the main loop of the CPU emulation. In this main loop, it looks to see if there is a hardware interrupt pending. By requiring user asynchronous interrupt calls and checking after basic blocks for interrupts, it allows the emulator to be much faster with less overhead while still supporting interrupts. Simics will allow for interrupts between each instruction by contrast, allowing for a higher execution fidelity at the cost of speed. Ghidra Emulator can also have interrupts at the sub-instruction level between different P-code operations.

In both cases, if you are testing something such as an interrupt storm (sending multiple interrupts with various orders and frequencies), then you will most likely have to asynchronously specify interrupts to the base emulator. When those interrupts are handled will affect the fidelity of the system.

8.2.7 Multi-Threading. In some systems, multi-threading is enabled. Applications that use multiple threads may use locks/semaphores for inter-thread communication, but this requires that your system emulator allow multiple threads to "run" at the same time as well. If the system allows multiple applications to be run, then they will employ a scheduler and use either pre-emptive or co-operative scheduling. With co-operative scheduling the emulator essentially just needs to allow the multiple threads to execute, and those threads will manage themselves during interaction. For pre-emptive, this will require trigging an interrupt to cause the scheduler to run and perform context switching between the threads. During emulation, usually the practitioner will start with single threaded applications, running one at a time and gradually increase the complexity and number of threads as the system gains more functionality and fidelity.

As mentioned throughout, with the ability to have fidelity at the instruction level, Simics is known for being more useful when debugging multi-threaded firmwares or systems. QEMU is able to emulate such systems, but in some cases it will give a false sense of correctness, because threads can only interact at the end of basic blocks for QEMU, whereas Simics and Ghidra Emulator allow thread interaction between any pair of instructions, even in the middle of a basic block. QEMU can attempt to overcome this limitation by setting break-points at every instruction, which essentially makes each instruction a basic block, but this dramatically slows down the system to the point that the emulation is not useful.

8.2.8 Debugging. During system emulation, undoubtedly you will run into some errors, whether they are errors in the actual firmware and/or system, or your emulator has errors and bugs. In either case, you need to be able to integrate some debugging. GNU GDB is a popular option to integrate into tools, having plugins for some of the major softwares including Avatar^2, angr, Ghidra, QEMU, and so on. Another option we have found useful is first emulating a serial port or other form of printing messages, enabling "printf" style debugging. In addition to printing, directly inserting function calls into the firmware can be useful for debugging purposes.

QEMU works as well as the debuggers that connect to it, with logging and setting some break points that should be familiar for those who have used GDB. Ghidra Emulator also allows for a GDB bridge to be connected for debugging. Simics employs integrated debugger support with both forward and reverse direction debugging available.

8.2.9 Timing Constraints. If the purpose of your emulation is to answer questions regarding timing, then you may be limited by the emulation approach and tools you use. In some cases you may be able to use a hardware integration solution such as SURROGATES, with specialized hardware to forward memory or peripheral accesses to an actual device. If you need to parallelize your emulation, then HITL may not be feasible. In such circumstances, the practitioner may have to implement their own timing. Implemented timing can be cycles, instructions, basic blocks or something similar. Timing is in many cases closely related to enabling interrupts, as some interrupts are timing or watchdog interrupts and in those cases require some sort of triggering or timing implementation in your emulator.

If the base emulator is Simics, then the emulator has a better idea of the number of cycles required for emulation, even if the emulator is slower to perform the operations. QEMU, in contrast, will be faster, but non-deterministic in execution time, which may be unsuitable if timing guarantees are needed.

9 POST EMULATION

Post Emulation challenges are usually directly related to the purpose of emulation. Some of the challenges include determining/patching vulnerable code, comparing actual vs. wanted behavior, finding vulnerabilities, visualizing emulation results, and automating the previous steps overcoming challenges.

9.1 Finding Vulnerabilities

Finding vulnerabilities is arguably the most popular reason for system emulation. There are many techniques to find vulnerabilities, with fuzzing being one of the most popular techniques.

There are many other techniques used to find vulnerabilities, including data flow analysis, taint analysis, control flow analysis, record and replay execution analysis, dynamic and symbolic execution. Some tools use supervised machine learning for vulnerability assessment, e.g., Costin et al. [45] use ML to classify firmware to help both address firmware vulnerability discovery and vulnerable device discovery. With the vast amount of work in vulnerability discovery, in depth discussion and scope is not feasible in this article, so we suffice to say these methods exists, and refer the readers to existing evaluation articles, including [7, 89, 143].

9.2 Verification

Once you have a system emulator, whether your first version emulator or a polished version, the problem of verifying that the emulator does what it is supposed to do arises. While verification

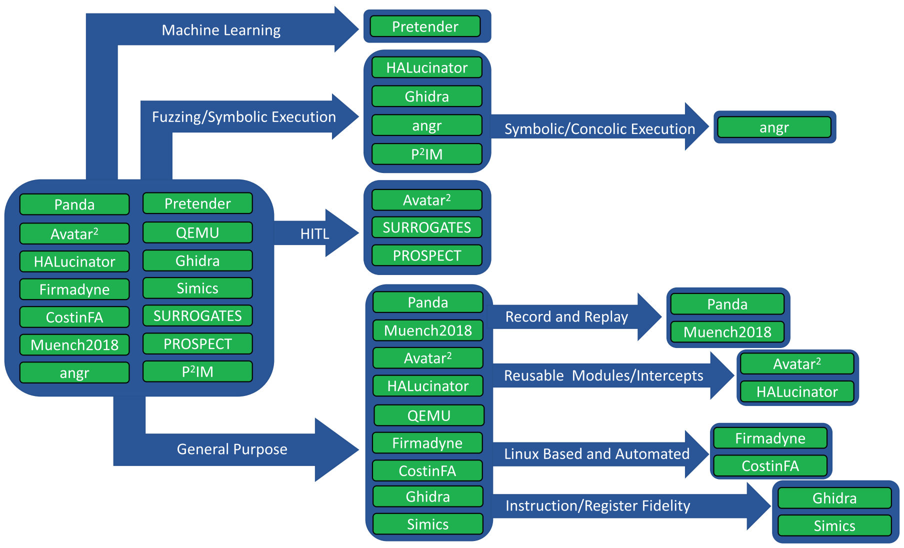

Fig. 4. Flow chart to choose emulator.

of SoC chip functionality before taping and production of a whole circuit system is well studied [102], verification of emulated systems and re-hosted firmwares seems to be lacking. None of the current tools verify whether they are correctly emulating the re-hosted firmware, beyond Blackbox behavior comparisons. In addition, there are no real benchmarks or verification techniques that are standard to test emulators. With the expansion of firmware emulation as an area of research, verification of the emulators and of re-hosted firmware execution is a critical need.

9.3 Analysis

As with normal hardware/software systems, emulated systems also are tested and analyzed after "completion." The firmware used during emulation may be analyzed and tested for bugs. The emulator itself may be analyzed to help with hardening firmwares that run on actual hardware. Analysis and post emulation includes vast amounts of effort in binary hardening, vulnerability detection and other cyber security efforts. All of the surveyed tools do some sort of post emulation analysis.

10 CONSIDERATIONS

When determining what base emulator or the different techniques/tools to use, there are pros and cons to each solution. Some of the main considerations we contemplate before choosing specific tools include hardware support, fidelity, performance, debug support/availability, and usability/control. These considerations are subject to the practitioner's opinion, though we have tried to objectively classify the tools fidelity and automation as shown in Figure 1.

In Figure 4, we provide an overview for giving a decision flowchart on choosing a tool, though we leave out a definitive path when trying to narrow tool use based on debug support, usability and control, as we have found opinions vary greatly for the various tools. On the left of the figure in a big box are all the surveyed tools. Following the arrows from there gives the key difference between the emulators. If the analyst is interested in machine learning, then Pretender will be a good place to start looking. Likewise, if interested in finding bugs/vulnerabilities using symbolic execution and fuzzing, then angr, Ghidra, HALucinator, and P^{2}IM are a promising starting points.

If you are interested in using actual hardware along with emulation, Avatar^2, SURROGATES, and PROSPECT are appropriate. Otherwise, general purpose emulators that have different strengths and uses include Panda, Muench2018, Avatar^2, HALucinator, QEMU, Firmadyne, CostinFA, Ghidra, and Simics. Once you have narrowed down the potential tools to a group of tools the following considerations should be evaluated.

Hardware support has perhaps been the most significant reason emulation has not been used in the past. The effort to emulate a system or re-host a firmware has traditionally been difficult to get working with systems (and some argue still is). This is slowly being remedied, with the core emulators having support for a wide variety of hardware. In addition, there is an increased effort to make adding new hardware and processors simpler and easier. One of the first considerations will still be if the tool you want to use has existing support for the hardware you want to emulate.

Emulator fidelity, as mentioned in Section 4.3, will narrow down which tools can be used to answer the question the practitioner is using emulation for. In some cases, symbolic execution is a feasible option to answer the emulation question at hand. In such cases, angr is the best option surveyed, being user friendly with a dedicated Slack [154] group with help and development channels. If symbolic execution will not work, then the practitioner can reference Figure 1 for help in narrowing down which surveyed tools can be used for the needed fidelity.

After considering fidelity, *performance* and *debugging support* also need considered. Depending on the emulation question, one of these attributes will be more important than the other. One of the main reasons emulation has not been used extensively in the past is because of the overhead and slow performance of emulators, but as software and hardware become more powerful, performance issues are being overcome.

When the performance of the emulator is a hindrance, partial emulation or specialized hardware are possible solutions. Ghidra Emulator allows for partial emulation, letting the practitioner start emulation at specific functions given they specify necessary inputs. SURROGATES provides specialized hardware to interact with memory faster. HALucinator sets breakpoints on user specified addresses provided in a YAML file. By analyzing only the necessary memory to answer a specific emulation question emulation performance can improved.

Debugging support is also a key attribute to consider. If the practitioner is re-hosting a firmware with limited knowledge of what the firmware is doing, then more debugging support will be essential. As noted, one of the benefits of emulation is being able to examine memory at each point during the firmware re-hosting. Examining memory requires the use of debugging tools or logging. For base emulators, QEMU and Ghidra (through Reference [37]) have GDB integration. Using GDB, break points can be used to step through emulation. Simics also has built-in debugging and allows for break points, along with stepping forward and backward through emulation. It is important to note that debugging support will affect performance, as the more break points and tracking needed will slow down the emulator. Both performance and debugging support are active areas in research, with base emulators trying to implement faster tools while monitoring and debugging.

Usability is a combination of exploration and control as referenced in Section 4.5 and tool automation. If a tool is automated, performs fast enough, and can answer the emulation question at hand, then that tool is obviously the best option. However, at the moment, this is not a common scenario, and thus how much control the practitioner has with exploring and executing re-hosted firmware is also an attribute that must be considered. Tools with GDB integration allow for expanded control. Tools that use fuzzing or randomization have less control and may or may not be useful for the practitioner in some cases. In essentially all the tools surveyed, debugging is built in to the base emulators and/or is available through plugins or default built in.

11 SUMMARY AND CONCLUSION

As a practitioner is contemplating the correct tool to use for firmware re-hosting and system emulation, we suggest using the provided emulation comparison techniques and considerations we have presented here. We have discussed comparing emulation tools by the techniques they employ, the types of systems they work for, the purpose of the tool, how much exploration the tool can achieve, and the most useful classification technique, fidelity.

Along with comparison techniques, we have shown classification of surveyed works including angr, Avatar^2, CostinFA, Firmadyne, Ghidra, HALucinator, P^{2}IM, PANDA, Pretender, PROSPECT, QEMU, Simics, SURROGATES, and Meunch2018. These techniques and their differences were discussed as we presented the core challenges faced during emulation and firmware re-hosting, giving an overview for the new practitioner or researcher looking on where to begin and further research in the firmware re-hosting and emulation field. Automation in current tools reduces the fidelity of the system dramatically, but in many cases is still sufficient for some emulation questions to be answered. The endeavor to automate current tools is difficult because of the wide coverage of embedded systems, but we note the valiant effort the surveyed tools have made and recognize the importance of making tools easier and automated for practitioners to use.

System emulation and firmware re-hosting has mutated and evolved dramatically since its inception. The challenges faced when trying to emulate a system/re-host a firmware have grown as emulation has improved. This article has highlighted the need for further research and tools to address Pre-Emulation, Emulation, and Post Emulation challenges. As tools and solutions are invented and released, we have provided classifications and criteria on how to evaluate such tools.

REFERENCES

[1] 2017. $20M in Bounties Paid and $100M In Sight. Retrieved from https://www.hackerone.com/blog/20M-in-bounties-paid-and-100M-in-sight.

[2] AFL-Fuzz. [n.d.]. afl-fuzz. Retrieved from https://github.com/google/AFL.

[3] Irfan Ahmed, Sebastian Obermeier, Martin Naedele, and Golden G. Richard III. 2012. SCADA systems: Challenges for forensic investigators. *Computer* 45, 12 (December 2012), 44–51. DOI: https://doi.org/10.1109/MC.2012.325

[4] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. FOSSIL: A resilient and efficient system for identifying FOSS functions in malware binaries. *ACM Trans. Priv. Secur.* 21, 2 (2018), 8.

[5] angr. [n.d.]. boyscout. Retrieved from https://github.com/angr/angr/blob/master/angr/analyses/boyscout.py.

[6] angr. [n.d.]. girlscout. Retrieved from https://github.com/angr/angr/blob/master/angr/analyses/girlscout.py.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. DOI: https://doi.org/10.1145/3182657

[8] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium.* 845–860.

[9] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[10] BE-PUM. [n.d.]. BE-PUM. Retrieved from https://github.com/NMHai/BE-PUM.

[11] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference.* USENIX Association, Berkeley, CA, 41–41.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Arch. News* 39, 2 (August 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[13] BitBlaze. [n.d.]. FuzzBALL. Retrieved from https://github.com/bitblaze-fuzzball/fuzzball.

[14] boofuzz. [n.d.]. boofuzz.Retrieved from https://github.com/jtpereyda/boofuzz.

[15] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2013. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, New York, NY, 411–421. DOI: https://doi.org/10.1145/2491411.2491433

[16] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2015. Symbolic execution of programs with heap inputs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, 602–613. DOI: https://doi.org/10.1145/2786805.2786842

[17] Jonathan Broome and David Marx. 2000. Method and Iimplementation for Intercepting and Processing System Calls in Programmed Digital Computer to Emulate Retrograde operating System. US Patent 6,086,623.

[18] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A binary analysis platform. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 463–469.

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[20] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, 1066–1071. DOI: https://doi.org/10.1145/1985793.1985995

[21] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 169–182.

[22] Capstone. [n.d.]. Capstone Disassembler. Retrieved from http://www.capstone-engine.org/.

[23] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant, and Dawn Song. 2013. *Transformation-aware Exploit Generation Using a HI-CFG*. Technical Report UCB/EECS-2013-85. EECS Department, University of California, Berkeley.

[24] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, 380–394. DOI: https://doi.org/10.1109/SP.2012.31

[25] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*.

[26] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: Detecting the taint-style vulnerability in embedded device firmware. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 430–441. DOI: https://doi.org/10.1109/DSN.2018.00052

[27] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32 a profile-directed binary translator. *IEEE Micro* 18, 2 (March 1998), 56–64. DOI: https://doi.org/10.1109/40.671403

[28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *SIGARCH Comput. Arch. News* 39, 1 (March 2011), 265–278. DOI: https://doi.org/10.1145/1961295.1950396

[29] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium*. 99–116.

[30] Catalin Cimpanu. 2019. Android Exploits Are Now Worth More Than iOS Exploits For The First Time. Retrieved from https://www.zdnet.com/article/android-exploits-are-now-worth-more-than-ios-exploits-for-the-first-time/.

[31] Cisco. [n.d.]. Joy. Retrieved from https://github.com/cisco/joy.

[32] Cisomag. 2020. Tesla Offers US$1 Million and a Car to Hack its Model 3 Car. Retrieved from https://www.cisomag.com/tesla-offers-us1-million-and-a-car-as-bug-bounty-reward/.

[33] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, New York, NY, 196–206. DOI: https://doi.org/10.1145/1273463.1273490

[34] John Clemens. 2015. Automatic classification of object code using machine learning. *Dig. Invest.* 14, S1 (August 2015), S156–S162. DOI: https://doi.org/10.1016/j.diin.2015.05.007

[35] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[36] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. 2015. PIE: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*. Association for Computing Machinery, New York, NY, 251–260. DOI: https://doi.org/10.1145/2818000.2818035

[37] Comsecuris. [n.d.]. GDB Ghidra. Retrieved from https://github.com/Comsecuris/gdbghidra.

[38] ConsenSys. [n.d.]. Mythril. Retrieved from https://github.com/ConsenSys/mythril.

[39] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138.

[40] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-wide security testing of real-world embedded systems software. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, Baltimore, MD, 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani.

[41] Andrei Costin and Jonas Zaddach. 2013. Embedded devices security and firmware reverse engineering. In *black hat USA 2013 Workshop*. blackhat.com. https://media.blackhat.com/us-13/US-13-Zaddach-Workshop-on-Embedded-Devices-Security-and-Firmware-Reverse-Engineering-WP.pdf.

[42] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. [n.d.]. firmware.re. http://firmware.re/usenixsec14/.

[43] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, 95–110. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[44] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, New York, NY, 437–448. DOI: https://doi.org/10.1145/2897845.2897900

[45] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2017. Towards automated classification of firmware images and identification of embedded devices. In *ICT Systems Security and Privacy Protection*, Sabrina De Capitani di Vimercati and Fabio Martinelli (Eds.). Springer International Publishing, Cham, 233–247.

[46] Craig. 2012. Emulating NVRAM in Qemu. Retrieved from http://www.devttys0.com/2012/03/emulating-nvram-in-qemu/.

[47] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Computer Society, Los Alamitos, CA, 653–656. DOI: https://doi.org/10.1109/SANER.2016.43

[48] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, Berkeley, CA, 463–478.

[49] Pietro De Nicolao, Marcello Pogliani, Mario Polino, Michele Carminati, Davide Quarta, and Stefano Zanero. 2018. ELISA: ELiciting ISA of raw binaries for fine-grained code and data separation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 351–371.

[50] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, New York, NY, Article 4, 11 pages. DOI: https://doi.org/10.1145/2843859.2843867

[51] Christopher Domas. 2017. Breaking the x86 ISA. In *black hat USA 2017 Workshop*. blackhat.com. https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf.

[52] DOSBox. [n.d.]. DOSBox. Retrieved from https://www.dosbox.com/.

[53] DroidSniff. [n.d.]. DroidSniff. Retrieved from https://github.com/evozi/DroidSniff.

[54] Thomas Dullien and Sebastian Porst. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. Zynamics. https://static.googleusercontent.com/media/www.zynamics.com/en//downloads/csw09.pdf.

[55] EtherApe. [n.d.]. EtherApe. Retrieved from https://etherape.sourceforge.io/.

[56] FaceDancer. [n.d.]. FaceDancer. Retrieved fom https://github.com/usb-tools/Facedancer.

[57] Bo Feng, Alejandro Mera, and Long Lu. 2019. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling (extended version). *arXiv* abs/1909.06472. Retrieved from https://arxiv.org/abs/1909.06472.

[58] Firmadyne. 2018. firmadyne/libnvram. Retrieved from https://github.com/firmadyne/libnvram.

[59] firminsight. [n.d.]. Retrieved from https://github.com/ilovepp/firminsight.

[60] firmware-mod-kit. [n.d.]. Retrieved from https://github.com/rampageX/firmware-mod-kit.

[61] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional symbolic execution for JavaScript. In *Proceedings of the ACM on Principles of Programming Languages* 3, Article 66 (January 2019), 31 pages. DOI: https://doi.org/10.1145/3290379

[62] Prashant Gandhi, Somesh Khanna, and Sree Ramaswamy. 2017. Which Industries Are the Most Digital (and Why)? Retrieved from https://hbr.org/2016/04/a-chart-that-shows-which-industries-are-the-most-digital-and-why.

[63] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed Systems Security Symposium.*
[64] Google. [n.d.]. clusterfuzz. Retrieved from https://github.com/google/clusterfuzz.
[65] Google. [n.d.]. domato. Retrieved from https://github.com/googleprojectzero/domato.
[66] Google. [n.d.]. fuzzilli. Retrieved from https://github.com/googleprojectzero/fuzzilli.
[67] Google. [n.d.]. gofuzz. Retrieved from https://github.com/google/gofuzz.
[68] Google. [n.d.]. honggfuzz. Retrieved from https://github.com/google/honggfuzz.
[69] Google. [n.d.]. syzkaller. Retrieved from https://github.com/google/syzkaller.
[70] Google. [n.d.]. winafl. Retrieved from https://github.com/googleprojectzero/winafl.
[71] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell.* ACM, New York, NY, 13–20. DOI: https://doi.org/10.1145/2976002.2976017
[72] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2020. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses.*
[73] Jim Hall. [n.d.]. HP LaserJet The Early History. Retrieved from http://hparchive.com/seminar_notes/HP_LaserJet_The_Early_History_by_Jim_Hall_110512.pdf.
[74] Armijn Hemel and Shane Coughlan. [n.d.]. Binary Analysis Toolkit. Retrieved from http://www.binaryanalysis.org/old/home.
[75] Hemel, Armijn. [n.d.]. BANG—Binary Analysis Next Generation. Retrieved from https://github.com/armijnhemel/binaryanalysis-ng.
[76] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin Butler. 2017. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17).* Association for Computing Machinery, New York, NY, USA, 2245–2262. https://doi.org/10.1145/3133956.3134050
[77] Brendan Hesse. 2019. Earn Up to $1 Million from Apple's Expanded Bug Bounty Program. Retrieved from https://lifehacker.com/earn-up-to-1-million-from-apples-expanded-bug-bounty-p-1837106598.
[78] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools.* ACM, 1–8.
[79] Janala2. [n.d.]. Janala2. Retrieved from https://github.com/ksen007/janala2.
[80] Dave Jones. 2011. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium.*
[81] Sami Kairajärvi, Andrei Costin, and Timo Hämäläinen. 2020. ISAdetect: Usable automated detection of CPU architecture and endianness for executable binary files and object code. In *Proceedings of the 10th ACM Conference on Data and Application Security and Privacy.* Association for Computing Machinery, New York, NY, 376–380. DOI: https://doi.org/10.1145/3374664.3375742
[82] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. CLIK on PLCs! Attacking control logic with decompilation and virtual PLC. DOI: https://doi.org/10.14722/bar.2019.23xxx
[83] Aaron Kaluszka. [n.d.]. Computer Emulation History. Retrieved from https://kaluszka.com/vt/emulation/history.html.
[84] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: Peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security.* ACM, New York, NY, 329–340. DOI: https://doi.org/10.1145/2590296.2590301
[85] Stamatis Karnouskos. 2011. Stuxnet worm impact on industrial cyber-physical system security. In *Proceedings of the 37th Annual Conference of the IEEE Industrial Electronics Society.* 4490–4494. DOI: https://doi.org/10.1109/IECON.2011.6120048
[86] Anastasis Keliris and Michail Maniatakos. 2019. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *Proceedings of the Network and Distributed Systems Security Symposium.*
[87] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative disassembly of binary code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* ACM, New York, NY, Article 16, 10 pages. DOI: https://doi.org/10.1145/2968455.2968505
[88] Kismet. [n.d.]. Kismet. Retrieved from https://www.kismetwireless.net/.
[89] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, New York, NY, 2123–2138. DOI: https://doi.org/10.1145/3243734.3243804
[90] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies.* USENIX Association, Berkeley, CA.

[91] Christopher Kruegel. 2014. Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *blackhat USA 2014 Workshop*. blackhat.com. https://www.blackhat.com/docs/us-14/materials/us-14-Kruegel-Full-System-Emulation-Achieving-Successful-Automated-Dynamic-Analysis-Of-Evasive-Malware-WP.pdf.

[92] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, Vol. 14. 11–11.

[93] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, Vol. 13. 18–18.

[94] Christopher Kruegel, William Robertson, and Giovanni Vigna. 2004. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*. IEEE, 91–100.

[95] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 75–86.

[96] Kevin P. Lawton. 1996. Bochs: A portable pc emulator for Unix/X. *Linux J.* 1996, 29es, Article 7 (September 1996). http://dl.acm.org/citation.cfm?id=326350.326357

[97] Leveldown Security. [n.d.]. SVD-Loader-Ghidra. Retrieved from https://github.com/leveldown-security/SVD-Loader-Ghidra.

[98] R. Li, Z. Zhao, X. Zhou, G. Ding, Y. Chen, Z. Wang, and H. Zhang. 2017. Intelligent 5G: When cellular networks meet artificial intelligence. *IEEE Wireless Commun.* 24, 5 (2017), 175–183.

[99] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. 2011. VIPER: Verifying the integrity of PERipherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, 3–16. DOI: https://doi.org/10.1145/2046707.2046711

[100] Yibin Liao, Ruoyan Cai, Guodong Zhu, Yue Yin, and Kang Li. 2018. Mobilefindr: Function similarity identification for reversing mobile binaries. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 66–83.

[101] Ulf Lindqvist and Peter G. Neumann. 2017. The future of the Internet of Things. *Commun. ACM* 60, 2 (January 2017), 26–30. DOI: https://doi.org/10.1145/3029589

[102] Peng Liu, Chunchang Xiang, Xiaohang Wang, Binjie Xia, Yangfan Liu, Weidong Wang, and Qingdong Yao. 2009. A NoC emulation/verification framework. In *Proceedings of the 6th International Conference on Information Technology: New Generations*. IEEE, 859–864.

[103] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, New York, NY, 196–199. DOI: https://doi.org/10.1145/3092282.3092295

[104] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium*.

[105] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[106] Malcolm. [n.d.]. Malcolm. Retrieved from https://github.com/idaholab/Malcolm.

[107] James Manyika, Sree Ramaswamy, Somesh Khanna, Hugo Sarrazin, Gary Pinkus, Guru Sethupathy, and Andrew Yaffe. 2015. Digital America: A tale of the haves and have-mores. Retrieved from https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/digital-america-a-tale-of-the-haves-and-have-mores.

[108] Xavi Mendez. [n.d.]. wfuzz. Retrieved from https://github.com/xmendez/wfuzz.

[109] Gaurav Mittal, David Zaretsky, Gokhan Memik, and Prith Banerjee. 2005. Automatic extraction of function bodies from software binaries. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005*, Vol. 2. IEEE, 928–931.

[110] Harish Mohanan, Perraju Bendapudi, Abishek Kumarasubramanian, Rajesh Jalan, and Ramarathnam Venkatesan. 2012. Function Matching in Binaries. US Patent 8,166,466.

[111] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. arxiv:cs.SE/1907.03890. Retrieved from https://arxiv.org/abs/1907.03890.

[112] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A multi-target orchestration platform. In *Proceedings of the Workshop on Binary Analysis Research, Colocated with Network and Distributed Systems Security Symposium*.

[113] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium*.

[114] NationalSecurityAgency. [n.d.]. NationalSecurityAgency/ghidra. Retrieved from https://github.com/NationalSecurityAgency/ghidra/wiki/Frequently-asked-questions.
[115] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Conf. Program. Lang. Des. Implement.* 42, 6 (June 2007), 89–100. DOI: https://doi.org/10.1145/1273442.1250746
[116] Netresec. [n.d.]. NetworkMiner. Retrieved from https://www.netresec.com/?page=NetworkMiner.
[117] NetWorkPacketCapture. [n.d.]. Retrieved from https://github.com/huolizhuminh/NetWorkPacketCapture.
[118] Lily Hay Newman. 2018. Facebook Bug Bounty Program Makes Biggest Reward Payout Yet. Retrieved from https://www.wired.com/story/facebook-bug-bounty-biggest-payout/.
[119] NSA. [n.d.]. Ghidra. Retrieved from https://ghidra-sre.org/.
[120] U.S. Department of Energy. [n.d.]. The Smart Grid. Retrieved from https://www.smartgrid.gov/the_smart_grid/smart_grid.html.
[121] OWASP. [n.d.]. IoTGoat. Retrieved from https://github.com/OWASP/IoTGoat.
[122] PAGalaxyLab. [n.d.]. vxhunter. Retrieved from https://github.com/PAGalaxyLab/vxhunter.
[123] Dorottya Papp, Zhendong Ma, and Levente Buttyan. 2015. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *Proceedings of the 2015 13th Annual Conference on Privacy, Security and Trust.* 145–152. DOI: https://doi.org/10.1109/PST.2015.7232966
[124] Riyad Parvez, Paul A. S. Ward, and Vijay Ganesh. 2016. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering.* IBM Corp., Riverton, NJ, 116–127. http://dl.acm.org/citation.cfm?id=3049877.3049889
[125] PcapPlusPlus. [n.d.]. PcapPlusPlus. Retrieved from https://github.com/seladb/PcapPlusPlus.
[126] PCem. [n.d.]. PCem. Retrieved from https://github.com/Anamon/pcem.
[127] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 697–710.
[128] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 709–724.
[129] Richard Phillips and Bonnie Montalvo. 2010. Using emulation to debug control logic code. In *Proceedings of the 2010 Winter Simulation Conference* (2010). DOI: https://doi.org/10.1109/wsc.2010.5678904
[130] PixelCyber. [n.d.]. Thor. Retrieved from https://github.com/PixelCyber/Thor.
[131] Praetorian. [n.d.]. The Damn Vulnerable Router Firmware Project. Retrieved from https://github.com/praetorian-code/DVRF.
[132] Rui Qiao and R. Sekar. 2016. *Effective Function Recovery for COTS Binaries Using Interface Verification.* Technical Report. Technical report, Secure Systems Lab, Stony Brook University.
[133] Rui Qiao and R. Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, 201–212.
[134] radamsa. [n.d.]. radamsa. Retrieved from https://gitlab.com/akihe/radamsa.
[135] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed Systems Security Symposium*, Vol. 17. 1–14.
[136] Hex Rays. [n.d.]. Retrieved from https://hex-rays.com/products/ida/.
[137] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. KARONTE: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the IEEE Symposium on Security and Privacy.*
[138] Teddy Reed. [n.d.]. subzero. Retrieved from https://github.com/theopolis/subzero.
[139] ReFirm Labs. [n.d.]. binwalk. Retrieved from https://github.com/ReFirmLabs/binwalk.
[140] Corinne Reichert. 2019. Google's Android Bug Bounty Program Will Now Pay Out $1.5 Million. Retrieved from https://www.cnet.com/news/googles-android-bug-bounty-program-will-now-pay-out-1-5-million/.
[141] Samsung. [n.d.]. Jalangi2. Retrieved from https://github.com/Samsung/jalangi2.
[142] Chase Schultz. [n.d.]. firmware_collection. Retrieved from https://github.com/f47h3r/firmware_collection.
[143] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society, Washington, DC, 317–331. DOI: https://doi.org/10.1109/SP.2010.26
[144] Sen, Koushik. [n.d.]. jCUTE. Retrieved from https://github.com/osl/jcute.
[145] Kostya Serebryany. 2017. OSS-Fuzz-Google's Continuous Fuzzing Service for Open Source Software.

[146] Saumil Shah. [n.d.]. The ARM-X Firmware Emulation Framework. Retrieved from https://github.com/therealsaumil/armx.
[147] Asankhaya Sharma. 2014. Exploiting undefined behaviors for efficient symbolic execution. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, 727–729. DOI: https://doi.org/10.1145/2591062.2594450
[148] Shellphish. 2017. Cyber Grand Shellphish. Retrieved from http://phrack.org/papers/cyber_grand_shellphish.html.
[149] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium*. 611–626.
[150] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium.*
[151] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy.*
[152] Sibyl. [n.d.]. Sibyl. Retrieved from https://github.com/cea-sec/Sibyl.
[153] Sickendick, Karl. [n.d.]. pcode-emulator. Retrieved from https://github.com/kc0bfv/pcode-emulator.
[154] Slack. [n.d.]. Slack. Retrieved from https://angr.slack.com.
[155] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security*. Springer, 1–25.
[156] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: Automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things* (2019), 15–21. DOI: https://doi.org/10.1145/3338507.3358616
[157] SSRFmap. [n.d.]. SSRFmap. Retrieved from https://github.com/swisskyrepo/SSRFmap.
[158] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium.*
[159] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. 2010. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems.* ACM, 169–182.
[160] Florin Dragos Tanasache, Mara Sorella, Silvia Bonomi, Raniero Rapone, and Davide Meacci. 2019. Building an emulation environment for cyber security analyses of complex networked systems. In *Proceedings of the 20th International Conference on Distributed Computing and Networking* (2019). DOI: https://doi.org/10.1145/3288599.3288618
[161] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. 2011. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems.* ACM, 288–301.
[162] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: Software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks.* ACM, 286–297.
[163] TCPDump. [n.d.]. Retrieved from http://www.tcpdump.org/.
[164] Radare2 Team. 2017. *Radare2 Book.* GitHub.
[165] Telerik. [n.d.]. Fiddler. Retrieved from https://www.telerik.com/fiddler.
[166] Keen Security Lab Tencent. 2016. Car Hacking Research: Remote Attack Tesla Motors. Retrieved from https://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/.
[167] Sam Thomas, Flavio Garcia, and Tom Chothia. 2017. HumIDIFy: A tool for hidden functionality detection in firmware. 279–300. DOI: https://doi.org/10.1007/978-3-319-60876-1_13
[168] Michael F. Thompson and Timothy Vidas. 2018. CGC Monitor: A Vetting System for the DARPA Cyber Grand Challenge. Retrieved from https://calhoun.nps.edu/handle/10945/59209.
[169] Brian Van Leeuwen, Vincent Urias, John Eldridge, Charles Villamarin, and Ron Olsberg. 2010. Cyber security analysis testbed: Combining real, emulation, and simulation. In *Proceedings of the 44th Annual 2010 IEEE International Carnahan Conference on Security Technology.* 121–126. DOI: https://doi.org/10.1109/CCST.2010.5678720
[170] Sebastian Vasile, David Oswald, and Tom Chothia. 2019. Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices. In *Smart Card Research and Advanced Applications*, Begül Bilgin and Jean-Bernard Fischer (Eds.). Springer International Publishing, Cham, 171–185.
[171] Marek Vasut. 2017. Adding New Architecture to QEMU. Retrieved from https://events17.linuxfoundation.org/sites/events/files/slides/ossj-2017.pdf.
[172] Trygve Vea. [n.d.]. firmwaredb. Retrieved from https://github.com/kvisle/firmwaredb.

[173] Vector 35. [n.d.]. Binary Ninja. Retrieved from https://binary.ninja/.
[174] John Viega and Hugh Thompson. 2012. The state of embedded-device security (Spoiler Alert: It's Bad). *IEEE Symp. Secur. Priv.* 10, 5 (September 2012), 68–70. DOI: https://doi.org/10.1109/MSP.2012.134
[175] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. 2014. Dynamic hooks: Hiding control flow changes within non-control data. In *Proceedings of the 23rd USENIX Security Symposium.* 813–328.
[176] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Machiry Aravind, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *Proceedings of the 2017 Network and Distributed System Security Symposium.*
[177] Xiajing Wang, Rui Ma, Bowen Dou, Zefeng Jian, and Hongzhou Chen. 2018. OFFDTAN: A new approach of offline dynamic taint analysis for binaries. *Secur. Commun. Netw.* 2018 (2018), 13. 10.1155/2018/7693861
[178] Kayla Wiles. 2019. First All-digital Nuclear Reactor System in the U.S. Installed at Purdue University. Retrieved from https://www.purdue.edu/newsroom/releases/2019/Q3/first-all-digital-nuclear-reactor-control-system-in-the-u.s.-installed-at-purdue-university.html.
[179] Wireshark. [n.d.]. Wireshark. Retrieved from https://www.wireshark.org/.
[180] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 921–937.
[181] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. 2019. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access* 7 (2019), 65889–65912.
[182] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, and Taejoo Chang. 2015. New acquisition method based on firmware update protocols for Android smartphones. *Dig. Invest.* 14 (2015), S68–S76. DOI: https://doi.org/10.1016/j.diin.2015.05.008
[183] Miao Yu, Jianwei Zhuge, Ming Cao, Zhiwei Shi, and Lin Jiang. 2020. A survey of security vulnerability analysis, discovery, detection, and mitigation on IoT devices. *Fut. Internet* 12, 2 (February 2020), 27. DOI: https://doi.org/10.3390/fi12020027
[184] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium.* 745–761.
[185] Jonas Zaddach, Luca Bruno, AurÃlien Francillon, and Davide Balzarotti. 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed Systems Security Symposium.* DOI: https://doi.org/10.14722/ndss.2014.23229
[186] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. 2013. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference.* Association for Computing Machinery, New York, NY, 279–288. DOI: https://doi.org/10.1145/2523649.2523661
[187] Ruijin Zhu, Yu-an Tan, Quanxin Zhang, Yuanzhang Li, and Jun Zheng. 2016. Determining image base of firmware for ARM devices by matching literal pools. *Dig. Invest.* 16 (2016), 19–28. DOI: https://doi.org/10.1016/j.diin.2016.01.002
[188] Ruijin Zhu, Baofeng Zhang, Junjie Mao, Quanxin Zhang, and Yu-an Tan. 2017. A methodology for determining the image base of ARM-based industrial control system firmware. *Int. J. Crit. Infrastruct. Protect.* 16 (2017), 26–35. DOI: https://doi.org/10.1016/j.ijcip.2016.12.002

Received January 2020; revised July 2020; accepted September 2020