SoK: Enabling Security Analyses of Embedded Systems via Rehosting

Andrew Fasano
fasano@mit.edu
MIT Lincoln Laboratory
Northeastern University
Boston, Massachusetts, USA

Tiemoko Ballo
tiemoko.ballo@ll.mit.edu
MIT Lincoln Laboratory
Lexington, Massachusetts, USA

Marius Muench
m.muench@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Tim Leek
trleek@ll.mit.edu
MIT Lincoln Laboratory
Lexington, Massachusetts, USA

Alexander Bulekov
alxndr@bu.edu
Boston University
Boston, Massachusetts, USA

Brendan Dolan-Gavitt
brendandg@nyu.edu
New York University
New York, New York, USA

Manuel Egele
megele@bu.edu
Boston University
Boston, Massachusetts, USA

Aurélien Francillon
aurelien.francillon@eurecom.fr
EURECOM
Biot, France

Long Lu
l.lu@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Nick Gregory
nmg355@nyu.edu
New York University
New York, New York, USA

Davide Balzarotti
davide.balzarotti@eurecom.fr
EURECOM
Biot, France

William Robertson
wkr@wkr.io
Northeastern University
Boston, Massachusetts, USA

ABSTRACT

Closely monitoring the behavior of a software system during its execution enables developers and analysts to observe, and ultimately understand, how it works. This kind of dynamic analysis can be instrumental to reverse engineering, vulnerability discovery, exploit development, and debugging. While these analyses are typically well-supported for homogeneous desktop platforms (e.g., x86 desktop PCs), they can rarely be applied in the heterogeneous world of embedded systems. One approach to enable dynamic analyses of embedded systems is to move software stacks from physical systems into virtual environments that sufficiently model hardware behavior. This process which we call "rehosting" poses a significant research challenge with major implications for security analyses. Although rehosting has traditionally been an unscientific and ad-hoc endeavor undertaken by domain experts with varying time and resources at their disposal, researchers are beginning to address rehosting challenges systematically and in earnest. In this paper, we establish that emulation is insufficient to conduct large-scale dynamic analysis of real-world hardware systems and present rehosting as a firmware-centric alternative. Furthermore, we taxonomize preliminary rehosting efforts, identify the fundamental components of the rehosting process, and propose directions for future research.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong.

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8287-8/21/06.
https://doi.org/10.1145/3433210.3453093

CCS CONCEPTS

• **Software and its engineering → Software reverse engineering**; **Software post-development issues**; **Dynamic analysis**; • **Hardware** → *Post-manufacture validation and debug*; *Simulation and emulation*; • **Computer systems organization → Firmware**; **Embedded software**; *Real-time systems*.

KEYWORDS

Dynamic program analysis; firmware security; emulation; embedded systems; internet of things; virtualization; rehosting

ACM Reference Format:
Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. 2021. SoK: Enabling Security Analyses of Embedded Systems via Rehosting . In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3453093

1 INTRODUCTION

Whereas enterprise edge systems are typically maintained by IT professionals, the rise of low-cost, consumer edge devices ("Internet of Things") has led to an increasingly large number of Internet-accessible machines which malicious actors can exploit. In 2016, the Mirai malware exploited insecure default credentials on many of these machines to create a botnet of approximately 600,000 devices [1]. But poor security posture is a problem emblematic of many embedded systems, not just consumer edge devices. Despite the frequent use of embedded systems for safety-critical, industrial applications, over 90% of embedded real-time operating systems fail to implement

virtual memory, cryptographically secure pseudo-random number generators, or basic exploit mitigations such as non-executable data memory, ASLR, and stack canaries [82]. Across a broad spectrum of applications, embedded systems are typically less secure than general-purpose computers due to a combination of technical and socio-economic factors. Embedded systems often contain numerous special-purpose peripherals leading to a larger attack surface than their general-purpose counterparts. BroadPwn [3], an unauthenticated Remote Code Execution exploit, attacks embedded systems (mobile phones) through a device peripheral (WiFi chipsets). A compromised peripheral may be leveraged to compromise the full system if the peripheral is Direct Memory Access capable [75] or if it can be used to exploit a vulnerability in a device driver [73, 74]. But these kinds of cross-component vulnerabilities cannot be discovered or replicated through analysis of a single application, since the exploit chain may leverage behavior of—and data flows between—multiple hardware and software components of a system.

Given a general lack of hardening and complex peripheral interactions, the ability to perform security analyses of embedded systems is critical to improving the security of these increasingly prevalent and network-connected devices. However, the vast majority of embedded systems security analyses performed today use only static analysis [11, 19, 22, 67, 85]. While this is a promising start, security assessments would be strengthened by the ability to conduct dynamic analysis of embedded systems [27].

Whole-system dynamic analysis can be achieved by decoupling a system's firmware from its underlying hardware to move—or *rehost*—the software into a virtual environment designed to run that firmware. However, this decoupling is no easy feat: firmware is written for a specific System-on-Chip (SoC), compiled for a specific CPU, and reliant on a specific set of peripherals. Rehosting a system into a virtual environment unlocks capabilities unavailable with physical systems: inspectability, mutability, replicability, scalability, and disposability. Such an environment can be leveraged to *inspect* execution of the rehosted system at arbitrary granularity and without the constraints of hardware-based solutions [41]. Similarly, the virtual system's state is fully *mutable*—CPU state or memory may be modified at any point in execution. Sources of non-determinism can be controlled in a virtual environment to ensure *replicability* of analysis results or behavior. As software, rehosted systems are *scalable*—they can be inexpensively cloned to conduct distributed analyses or to share with collaborators. Finally, virtual systems can be created on demand, and because they are *disposable*, analysts need not worry about damaging a rehosted system. Altogether, these properties enable analysts to conduct security analyses—such as smart fuzzing and symbolic execution—that would otherwise be infeasible on the original physical system or through static analysis.

Though rehosting has numerous benefits and clear use cases, it has not previously been approached in a systematic fashion. This paper seeks to:

- introduce the need for and challenges of rehosting (§2, 3, 5);
- quantitatively analyze obstacles preventing embedded firmware from running in whole-system emulators (§ 4);
- taxonomize preliminary work in the rehosting space (§ 5.2);
- identify key components of the rehosting process (§ 6); and
- establish a roadmap to guide future rehosting research (§ 7).

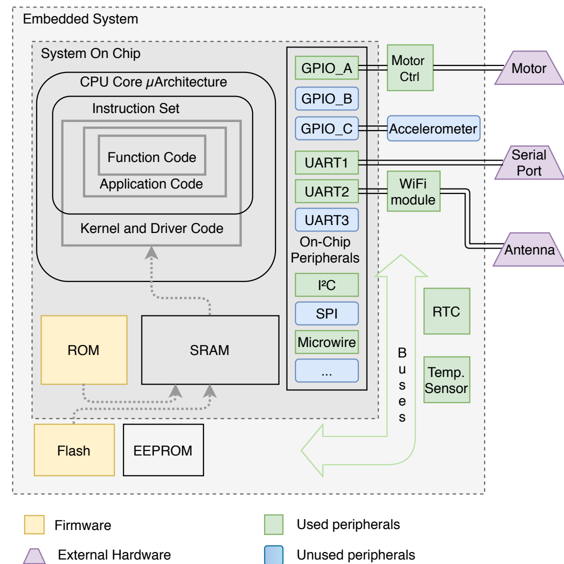


Figure 1: An illustrative embedded system.

2 REHOSTING FOR WHOLE-SYSTEM SECURITY ANALYSIS

In contrast to a hardware emulation system which comprehensively reproduces the features of specific hardware in a virtual environment, a rehosted embedded system is designed around a specific firmware image and must only reproduce the necessary hardware features that enable the firmware (or relevant components thereof) to run sufficiently in a virtual environment. We define these terms as follows:

Definition 1. Virtual Environment (VE): A software environment in which code can be executed transparently.

Definition 2. Hardware Emulation System (HES): A VE designed to accurately recreate the features of one or more selected pieces of hardware. Commonly called an emulator.

Definition 3. Rehosted Embedded System (RES): A combination of a firmware image and VE designed to sufficiently recreate the firmware's hardware dependencies such that analyses produce results representative of the firmware running on its original hardware.

HESs target hardware of varying scale, from an ISA or CPU [65] to an entire SoC [84], or even the physical systems connected to an embedded system [9]. When available, the VE provided by an HES should be the preferred way to conduct dynamic analyses of an embedded system's firmware. However, the hardware platforms supported by HESs are extremely limited in both scale and diversity. We believe this scarcity is due to the fundamental challenge of building an HES: supporting all the features of a given hardware platform requires a comprehensive understanding of the system and every feature that could be used.

In contrast, building a RES avoids this challenge by only modeling the hardware features necessary to enable a selected firmware

to run sufficiently for some analysis task. We define this process as follows:

Definition 4. Rehosting: The process of building an RES for a given embedded system to enable a specified analysis task. May include modifications to the firmware.

Consider the hypothetical embedded system presented in Fig. 1. By limiting the scope of the VE to only the necessary components, rehosting greatly reduces the barrier to entry for dynamic security analyses. Once firmware can be run in a VE, it is inspectable, mutable, replicable, scalable, and disposable, properties critical for dynamic security analyses that are generally absent from physical systems.

Inspectability, scalability, and disposability are critical for large-scale, coverage-guided fuzz testing where mutated inputs are passed into many copies of a system. By inspecting program execution, a fuzzer can measure code coverage to guide mutations of inputs, expediting discovery of new execution paths and potential bugs [45]. Inspectability also underpins dynamic taint analysis, a technique for tracking how program states and values are derived from specially marked inputs. Taint analysis can identify execution paths in which tainted inputs influence sensitive parts of a system. Taint information can aid both vulnerability discovery and general reverse engineering [71].

Mutability, replicability, inspectability, and disposability enable forced execution, a technique which explores a program's state by repeatedly executing uncovered branches via mutation of CPU state at branching instructions. Forced execution can be used to generate both control flow graphs and call graphs that are more accurate than those produced by static analysis. This technique can also be used to aid in dynamic type reconstruction [63].

More broadly, accurately rehosting a system into a VE where it is fully inspectable enables security analyses to consider any possible input to a system and the resulting behavior. If the inputs to an embedded system can be configured in such a way to produce undesired behavior, an attacker may leverage this vulnerability and craft the necessary inputs to exploit the system.

Although the process of rehosting a system fundamentally requires decoupling its software stack from its physical hardware, this decoupling can also occur at higher layers of abstraction. Logic within user space applications (*function layer*), or even which applications are run (*application layer*), may be modified to enable rehosting. If an OS is present, it may be modified to enable rehosting (*OS layer*). Finally, the peripherals and CPUs of a physical system may be modified to support rehosting (*hardware layer*). Vulnerabilities in embedded systems may be caused by one or more mistakes in a single abstraction layer or the interactions between mistakes across multiple layers. Therefore modifications must be made with caution.

2.1 Multi-Layer Vulnerabilities

To illustrate how vulnerabilities arise from interactions across system layers, consider the `auth_user` function in Listing 1 which compares user input against a password stored in non-volatile storage (NVRAM). While use of the `gets` function introduces a clear vulnerability contained in the function layer, other bugs in the snippet can turn into vulnerabilities depending on the hardware environment that executes this code. Note that neither the return value of

```
1  int secure_memcmp(char *s1, char *s2, int len){
2      int res = 0;
3
4      // Determine if any characters mismatch
5      for(int i = 0; i < len; i++)
6          res |= s1[i] ^ s2[i]
7      return res;
8  }
9
10 int auth_user(void) {
11     nvram_handle_t h;
12     char *pwd;
13     char buf[32];
14
15     // Read stored password from nvram
16     h = nvram_open("/dev/nvram", O_RDONLY);
17     pwd = nvram_get(h, "user_pass");
18
19     // Read data from (untrusted) user
20     gets(buf);
21     return secure_memcmp(buf, pwd, strlen(pwd));
22 }
```

Listing 1: Insecure authentication function (hypothetical).

`nvram_open` nor `nvram_get` is checked. These functions may return `NULL` if the peripheral acting as NVRAM fails. While this would lead to a crash on general-purpose computers, embedded systems commonly fail to deploy memory protections and allow mapping of the `NULL` page. In such a scenario, an attacker who can predict the contents of the `NULL` page would be able to bypass the authentication check.

Even seemingly bug-free portions of code can turn into vulnerabilities depending on the hardware environment in which they are executed. The `secure_memcmp` function in Listing 1 aims to prevent timing side-channel attacks for password-retrieval by introducing a constant time string comparison. However, this mitigation is predicated on the assumption that the system executes `OR` and `XOR` operations without data-dependent differences in speed. If this assumption is incorrect, an exploitable timing side-channel may exist.

If a security analysis is conducted using a VE which fails to sufficiently capture behavior of these layers, the analysis results may be inaccurate. As such, it is essential to understand any modifications to abstraction layers made by an RES.

3 CHALLENGES TO BUILDING VIRTUAL ENVIRONMENTS

Hardware and software of embedded systems are tightly coupled and tailored to perform a set of specific operations. Examples of such systems are provided in Appendix A. Embedded systems are incredibly diverse as each is designed to satisfy a novel combination of use case, power, performance, and cost constraints. These constraints impact all phases of the system design including instruction set architecture selection, peripheral selection, hardware design, and software functionality. To satisfy these constraints, modularity and standardization—typically emphasized on general purpose computers—are routinely sacrificed for custom logic that assumes specific hardware configurations. These assumptions introduce challenges for security analyses that seek to evaluate firmware in a VE.

As a result of this diversity, multiple taxonomies for classifying embedded systems exist. We follow the security-oriented classification proposed by Muench et al. [60] which categorizes embedded systems based on their deployed OS type. **Type-1** systems use general purpose OSs retrofitted for embedded systems; **Type-2** systems use custom embedded OSs; and **Type-3** systems do not use OS abstractions at all. Non-embedded systems are described as **Type-0**.

3.1 Obtaining Firmware

Conducting dynamic analyses of any system fundamentally requires access to the code the system executes (compiled or as source). But unlike with traditional software systems, possession of an embedded system does not immediately enable analysis of its logic. Obtaining its firmware may require significant resources, especially as firmware may be encrypted-at-rest or guarded by hardware readout protections. In these cases, invasive hardware attacks [61, 64, 79] can typically be used to extract firmware from an embedded system but require specialized equipment, such as scanning electron microscopes and focused ion beams [78]. On the other hand, non-invasive hardware attacks (e.g., connecting to debug interfaces) and software-based techniques (e.g., downloading/intercepting firmware updates or software exploitation) for extracting firmware vary from one embedded system to the next. When available, these approaches provide an alternative path to firmware extraction without the need for specialized equipment [80].

3.2 Understanding Instruction Set Architectures

Though some embedded systems use programmable logic chips such as field programmable gate arrays (FPGAs) or complex programmable logic devices, most rely on a primary, general-purpose CPU. An instruction set architecture (ISA) describes how a CPU decodes and executes machine instructions.

The x86 ISA family is used for the vast majority of general-purpose computers and, as such, many systems have been developed to analyze and emulate it [48, 70]. On the other hand, the embedded market routinely uses ARM, MIPS, PPC, AVR, and other ISA families. Within each family there are often incompatibilities between various offerings, and some ISAs, such as MIPS and Xtensa, allow vendor customization which creates even more diversity.

The diversity of ISAs used in embedded systems poses challenges to security research as analyzing machine code for any given system requires a detailed understanding of the system's ISA. A VE must capture this understanding through a Virtual Execution Engine:

Definition 5. Virtual Execution Engine (VXE): A mechanism for interpreting instructions for a given ISA in a VE.

A VXE may provide this interpretation using a model of an ISA specification or by running an interpreter on an intermediate representation of version of compiled code (e.g., McSema [25]). A HES should provide a VXE that fully captures an ISA's semantics, but an RES need only support the subset of the ISA that is actually used by a given firmware.

3.3 Modeling Peripherals

Peripheral devices work alongside the CPU to provide additional functionality and interface with the external world. Traditional desktop systems use peripheral enumeration to dynamically discover devices as they are connected and disconnected over external (e.g., USB) or internal (e.g., PCIe) buses [30, 32]. Moreover, BIOS and UEFI both provide standardized OS/HW interface abstractions for desktop systems: the OS can query for peripherals not present on enumerable buses [31] and assume a standard I/O address space [5].

In contrast, embedded systems often lack an equivalent OS/HW interface abstraction and instead rely on a fixed set of permanently connected peripherals. Peripheral configurations are often tightly coupled with the OS and applications due to the manufacturer's knowledge of the hardware configuration. As such, VEs generally must model these peripherals to ensure systems behave as expected. Beyond simply ensuring functionality, modeling peripherals in VEs is critical for security analyses as peripherals are typically the source of attacker-controlled data.

3.4 Evaluating Fidelity

The fidelity of a rehosted system describes how well the behavior of an RES mirrors it's physical counterpart. The literature lacks a formal definition of rehosting fidelity and no large-scale fidelity evaluations have been conducted to date. However, individual rehosting techniques have been evaluated by measuring if systems accept network connections [10], collecting and comparing peripheral interactions [49], and comparing the similarity of instruction traces [8].

We identify that, in the general case, the problem of measuring rehosting fidelity is an example of an unsolvable variation on the equivalence problem [87]. Since current rehosting techniques commonly produce RESs with clearly distinct behavior from their physical counterparts, fidelity can typically be described by measuring the observable differences.

4 QUANTIFYING THE DIFFICULTY OF EMBEDDED HARDWARE EMULATION

To motivate the need for firmware-centric rehosting, we analyze two corpora of machine-parsable hardware descriptions, jointly representing over two and a half thousand embedded SoCs, and evaluate the tractability of using open source HESs to replicate the described hardware.

With these corpora, we measure the availability of VXEs for the described CPUs and HES support for peripherals found in the described systems. Subsequently, we estimate the complexity of the described peripherals and evaluate the feasibility of generating peripheral models as needed through a Monte Carlo simulation. Our results show that developing HESs does not scale and that a large gap exists between the hardware platforms supported by such systems and the platforms used by firmware.

4.1 Datasets

Our datasets consist of 1,956 individual Device Tree Blob (DTB) files from Linux Kernel version 5.11.4[1] and 618 manufacturer-provided

[1] https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.11.4.tar.xz (February 2021)

System View Description (SVD) files[2] conforming to the Cortex Microcontroller Software Interface Standard. DTB files are standardized descriptions of hardware (CPU and on/off-chip peripherals) for Type-1 embedded systems, agnostic of OS or architecture, and are parsed by an OS kernel during boot to drive hardware initialization. Systems in our Linux dataset range from development boards (e.g., ARM Versatile Platform Baseboard) to commercial products (e.g., Nintendo Gamecube). Linux DTB support dates back to 2005 [51] with usage in PPC kernel ≥ 2.6 and ARM kernel ≥ 3.7.

The SVD files describe ARM Cortex hardware for Type-2 and Type-3 embedded systems, such as medical devices and mesh network transmitters respectively. Though OS-agnostic, they describe exclusively ARM architecture systems. SVD files are used during development and debugging to understand the memory-mapped interface between a CPU and an SoC's peripherals [2].

Prior firmware measurement studies [10, 19] used web crawlers to collect publicly available firmware images. These approaches are difficult to reproduce as copyright restrictions prevent corpora distributed and link rot prevents crawlers from running successfully after release. In contrast, kernel source and SVD datasets are largely available and mirrored, making our approach fully reproducible. Appendix B describes our methodology in detail. Our analysis code is publicly available and containerized[3].

4.2 Approach

We use two corpora to analyze 2,574 real-world hardware configurations. We only consider architectures (DTB corpus) or silicon vendors (SVD corpus) with 10 or more distinct samples. Each sample maps to exactly one real-world embedded system. We quantify:

(1) **VXE availability** by contrasting corpora CPU models against those supported in a mature, open-source HES;
(2) **VE diversity** by measuring the number of unique peripherals in corpora SoCs;
(3) **VE complexity** by measuring complexity of corpora SoCs as a function of peripheral driver code size; and
(4) **Tractability of HES implementation** by simulating VE creation to measure transferable work.

All of these analyses use data from our DTB corpus while diversity (2) and tractability (4) are supplemented with information from our SVD corpus.

4.3 Availability: Virtual Execution Engines

A critical component of a VE is its VXE. To evaluate the availability of VXEs for real-world embedded systems, we contrast CPU models present in our DTB corpora with those supported in the QEMU emulator [4], the predominant, open-source, HES. Matches are based on exact CPU *core*, e.g., `cortex-a9`, not *ISA version*, e.g., `ARMv7-A`. While using an alternate model of the same ISA version may sometimes be possible, in practice such a substitution may introduce discrepancies (e.g., illegal extension instruction). Appendix B.1 provides additional details of our CPU model matching methodology.

The results of this comparison are presented in Table 1. Note how little CPU support has increased over time, both in absolute counts and as a percentage of corpus CPUs. In the worst case for modern QEMU, only 12% of the observed ARM64 CPU models are supported. Appendix B.4 provides data for the similarly miniscule increase of peripheral totals across these versions.

[2] https://github.com/posborne/cmsis-svd
[3] https://github.com/igloo-re/rehosting_sok

Table 1: Observed CPU models supported by QEMU versions.

	v2.11.1 (Feb. '18)		**v5.2.0** (Dec. '20)	
Arch	Models avail.	Dataset supported	Models avail.	Dataset supported
ARM	31	20%	36	20%
ARM64	33	9%	39	12%
MIPS	15	50%	16	50%
PPC	407	53%	407	53%

Table 2: Peripheral diversity.

(a) Type-1 Linux Systems (DTB corpus)

Arch	|SoC|	Unique P	$\mu \pm \sigma$ P/SoC	$\tilde{x}$ P/SoC
ARM	1,310	6,858	58 ± 26	55
ARM64	430	3,653	58 ± 24	59
MIPS	20	270	21 ± 11	16
PPC	196	1,422	31 ± 19	27

(b) Type-2 and Type-3 ARM Cortex Systems (SVD corpus)

Vendor	|SoC|	Unique P	$\mu \pm \sigma$ P/SoC	$\tilde{x}$ P/SoC
Atmel	147	416	34 ± 10	30
Freescale	133	561	49 ± 13	47
Fujitsu	100	237	44 ± 9	41
NXP	24	374	28 ± 18	21
STMicro	72	852	59 ± 22	58
SiliconLabs	10	62	40 ± 2	40
Spansion	88	193	44 ± 9	42
TI	52	95	27 ± 4	26

4.4 Diversity: Unique Peripherals

Beyond the VXE, a VE for an SoC must also handle peripheral interactions. Across our corpora, we see 14,715 distinct peripherals, a quantity that far exceeds the number of supported peripherals in any modern HES. Methodology for peripheral identification is detailed in Appendix B.2.

Table 2 shows the SoC count and unique peripheral count for each architecture (DTB corpus) and each ARM Cortex vendor (SVD corpus). We also calculate the mean with standard deviation of peripheral count per SoC and the median peripheral count per SoC (represented as $\tilde{x}$). From these data, we see that a Type-1 PPC HES would need to support 1,422 unique peripherals to support all 196 Type-1 PPC SoCs or 31 peripherals, on average, for any single SoC. The diversity of peripherals present in Type-2 and Type-3 ARM Cortex systems varies by silicon manufacturer, but the mean and median peripheral per SoC is comparable to that of Type-1 ARM systems.

In contrast to the diversity of peripherals in real-world embedded systems, modern open-source HESs support relatively small sets of peripherals. For example, QEMU version 5.2.0 has models for 337 ARM64 peripherals and 216 PPC peripherals while OVPsim [53], an HES leveraging standardized peripheral models, has only 23 ARM peripherals and 3 MIPS peripherals.

Table 3: SLOC for open-source device drivers.

Arch	|DD|	$\mu \pm \sigma$ SLOC/DD	|SoC|	$\mu \pm \sigma$ SLOC/SoC
ARM	3,783	617.29 ± 650.50	1,310	43,036.59 ± 21,598.35
ARM64	2,414	665.87 ± 716.87	430	44,088.23 ± 20,404.85
MIPS	175	383.46 ± 465.71	20	9,406.93 ± 5,318.66
PPC	324	495.08 ± 428.79	196	22,879.97 ± 12,861.21

4.5 Complexity: Driver SLOC as a Proxy

We can approximate peripheral and SoC complexity for embedded Linux systems by measuring Source Lines Of Code (SLOC) for open-source drivers corresponding to peripherals referenced in our DTB corpus. Driver SLOC is a proxy for complexity: a device driver implements only half of the OS/hardware "conversation." Each driver interfaces with a hardware peripheral whose internal states and logic may not always be proportionally complex to that of the device driver. Thus, we use SLOC data to provide a rough estimate of the software engineering effort required to build a model of a given peripheral. SLOC computation methodology and an analysis of the correlation between QEMU peripheral vs. device driver SLOC is described in Appendix B.3.

Table 3 shows the mean and standard deviation of SLOC for device drivers and SoCs. Notably, the standard deviation for SLOC per driver can be higher than the mean, indicating considerable variability in driver complexity. Looking at ARM64 as an example, we can expect the average SoC to require over 44,000 device driver SLOC implemented as kernel code to manage its hardware. Although not every device driver will mirror the complexity of its associated hardware peripheral and our sample of open-source drivers may not be representative of all closed-source counterparts, these SLOC measurements hint at the scale of software engineering effort required to build HESs for disparate SoCs.

4.6 Tractability: Simulation of Emulating Hardware Systems

One potential strategy to build an HES that supports a large number of SoCs is to incrementally add support for new hardware components the first time each component is present in an SoC of interest. If such a strategy were to be pursued from scratch, building the HES for the first SoC would require modeling the VXE and all its peripherals. Extending the HES to support subsequent SoCs would require less effort if the VXE and peripherals were used by a prior SoC and thus already supported by the HES. This approach would be viable for generating VEs if, after some initial effort building models for common VXEs and peripherals, the problem were to become significantly easier. By running two Monte Carlo simulations of building an HES to support SoCs from our datasets, we discover that this is not the case and conclude that building or extending HESs is an impractical approach to building VEs for SoCs of interest.

In our simulations, we imagine an analyst selecting SoCs at random and building an HES by creating new peripheral models and VXEs whenever an SoC has a never-before-seen peripheral or CPU. To simplify result summary, we treat CPUs and peripherals equivalently throughout the simulation. This does not impact result validity since both must be modeled in an HES. We also do not subtract

```
SoCs = [...] // Systems from DTB or SVD corpus
for sim_round = 1 to 1000 do
    P_m = {∅}
    // Randomly sample 10% of SoCs and simulate updating HES
       to support each
    for i = 1 to |SoCs|/10 do
        system = get_rand_from(SoCs)
        P_u = {∅}
        foreach p ∈ system do
            if p ∉ P_m then
                P_u = (P_u ∪ {p})
            end
        end
        P_m = (P_m ∪ P_u)
        Record(|P_u|) // Per-system effort
    end
    Record(|P_m|) // Cumulative effort
end
```

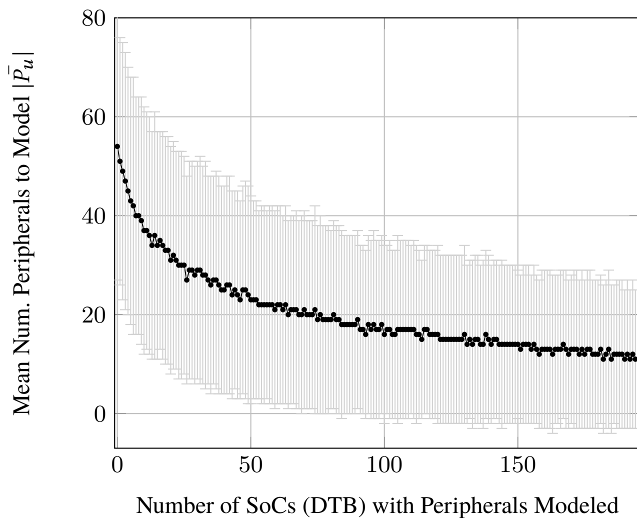
Algorithm 1: Simulation of peripheral modeling.

QEMU-supported CPUs or peripherals from the modeled total. This ensures our results reflect the difficulty of HES construction in general, not just the current state of QEMU. We contrast against QEMU only to provide context. For each of the randomly selected SoCs, we record how many new peripherals must be modeled to estimate the *marginal work* required to update the HES to support the new SoC given all the prior (simulated) work. Algorithm 1 depicts this simulation in detail.

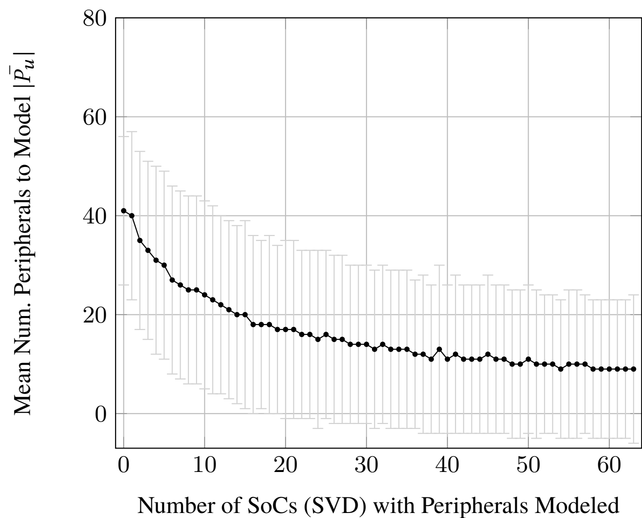
In each round of the simulation, we sample 10% of the relevant corpus (at random and without replacement) and simulate building an HES to support the selected systems. The peripherals supported by the HES are tracked in P_m, which begins as an empty set. As we iterate through the selected systems in a random order, we populate P_u with a list of the peripherals used by each system that are unsupported by the HES. After examining each system, we simulate updating the HES to support these peripherals by adding P_u into P_m. Thus, $|P_m|$ is a running total of aggregate effort, and $|P_u|$ is the per-system required effort. This Monte Carlo simulation runs for 1,000 rounds.

To measure how much of our theoretical implementation work translates between SoCs, we consider the mean count of unimplemented peripherals, $|\bar{P_u}|$, in each of the sampled systems. This value is shown in Fig. 2, across all rounds for each of our simulations. Note that for the final, 195th, Linux system, we still have to implement 11.4 ± 14.4 peripherals, on average. These simulations suggest that even if an analyst chose to manually implement thousands of peripherals into an HES, missing peripherals would still be a major roadblock to supporting subsequent SoCs. Note that the simulation's random selection naturally accounts for "most common" peripherals. The simulation results show that peripherals are so diverse that prioritization is not helpful.

On average, developing an HES capable of supporting a randomly selected 10% of our SoCs would require supporting $3,947 \pm 159$ peripherals for Linux systems or $1,730 \pm 125$ peripherals for ARM Cortex systems. By contrast, QEMU 5.2 and OVPsim implement a total of $1,083$ and 216 peripherals respectively for the surveyed architectures.



(a) Unimplemented peripherals in 195 randomly sampled embedded Linux systems at each round of simulation.



(b) Unimplemented peripherals in 64 randomly sampled Cortex systems at each round of simulation.

Figure 2: Average number of peripherals that must be modeled when building 1,000 (simulated) HESs to support 10% of the embedded systems from each corpus. For each HES, a system is selected from the relevant corpus, its count of unmodeled peripherals ($|P_u|$) is recorded, and it's unmodeled peripherals are imagined to be modeled for subsequent system selections.

This means that emulating the SoCs in our random sample of the DTB corpus requires approximately four times the number of peripherals QEMU currently supports. This does not imply QEMU is a fourth of the way to being viable for embedded emulation; the total number of manually implemented peripherals, $|P_m|$, only grows if we select a larger percentage of the corpora (e.g., 25% or 100%) as we would be arranging to emulate more systems and, consequently, encounter even more unimplemented peripherals. Our 10% sample size was chosen to demonstrate that a leading modern HES cannot support even a small subset of either corpus. After two years of QEMU development, the situation has not improved. Moreover, new SoCs with new peripherals are constantly being manufactured, so peripheral diversity in-the-wild is likely to increase over time. This makes manual implementation an unending, expensive endeavor.

To estimate engineering effort for these peripheral implementations, we track driver SLOC for every modeled peripheral in the DTB corpus. Each time an unmodeled peripheral is encountered, we count its SLOC if source is available. For closed-source drivers, we use the architecture-specific average SLOC per driver. The mean SLOC total corresponding to the all peripherals implemented by the end of the simulation is 2,448,354. Hence, we conclude that manual peripheral implementation does not scale and that an HES with support for the majority of embedded systems will likely never exist.

5 THE CASE FOR REHOSTING

From the analyses presented in § 4, it is evident that embedded systems are remarkably diverse and impossible to fully support in HESs without automation. In spite of these challenges, there is still a clear need for dynamic analysis of the firmware running on embedded systems which can be accomplished by rehosting. Prior work has taken this firmware-centric approach with a variety of strategies [10, 20, 23, 43, 47, 49, 77, 86]. However, without a standard definition of the underlying process, advances in this space have typically been ancillary to enabling other research tasks, such as fuzzing embedded web applications [20]. We argue that by studying rehosting as a research problem in its own right, the broader research community can find more general solutions which will make narrower problems much easier to solve.

At present, the process of rehosting is more alchemy than chemistry—opaque, unrepeatable, and prone to failure. We hope to see a future in which rehosting is a systematic and scientific endeavor made possible with standard methodology and effective technologies. To that end, this section identifies the goals of rehosting, contextualizes prior work, identifies how different classes of embedded systems present different rehosting challenges, and examines how reducing the scope of a VE can ease the rehosting process.

5.1 Rehosting Goals

Decoupling firmware from its physical dependencies facilitates a wide spectrum of processes including reverse engineering, training, system evaluation and certification, vulnerability research, and exploit development. With each of these use cases, a different population of users wish to leverage rehosting towards a different goal. For instance, hardware and software vendors may wish to use rehosting during testing and development of a product. These vendors may use rehosting techniques that require expert knowledge of their target hardware platform and manually build a VE. On the other hand, third parties who lack detailed information on a target platform (e.g., security analysts, reverse engineers, or system integrators) may be interested in rehosting for vulnerability discovery, system understanding, or system verification. These users need an approach to

Table 4: Taxonomy of existing work according to how each layer is handled.
🌐: Passed through 💻: Emulated ○: Not modified ◪: Replaced x: Symbolic model

	System	Layer: Hardware	Layer: OS	Layer: Application	Layer: Function	System Type	Target ISA(s)	Binary?	Availability[a] Source	Availability[a] Dataset
Pure Emulation	BaseSafe [54]	◪	◪	◪	💻	2	ARM	✓	✓	✓
	Clements'21 [16]	💻 ◪	◪	○	○	2	ARM	✓		
	Costin'16 [20]	💻	◪	○	○	1	ARM, MIPS	✓		
	DICE [57]	💻 ◪	○	○	○	2 3	ARM, MIPS	✓	✓	✓
	Firm-AFL [88]	💻	◪	○	○	1	ARM, MIPS	✓	✓	✓
	Firmadyne [10]	💻	◪	○ ◪	○	1	ARM, MIPS	✓	✓	
	FirmAE [46]	💻	◪	○ ◪	○ ◪	1	ARM, MIPS	✓	✓	
	HALucinator [15]	💻 ◪	◪	○	○ ◪	2 3	ARM	✓	✓	✓
	Li'21 [50]	◪	◪	○	○	2	N/A		~	~
	LuaQEMU [17]	💻	—	○	○ ◪	3	ARM	✓	✓	
	P2IM [29]	💻 ◪	○	○	○	2 3	ARM	✓	✓	✓
	PartEmu [37]	💻 ◪	○	○ ◪	○	2	ARM	✓		
Hardware-in-the-Loop	Avatar2 [59]	🌐 💻 ◪	—	🌐 ○	🌐 ○	3	ARM	✓	✓	✓
	Charm [77]	🌐 💻	◪	○	○	1	ARM		✓	✓
	FEMU [49]	🌐 💻	○	○	○	0 1 2 3	x86			
	Frankenstein [69]	💻 ◪	◪	○	○ ◪	2	ARM	✓	✓	
	FirmCorn [35]	💻	🌐 ◪	○	○ ◪	1	ARM, MIPS, x86	✓	✓	~
	Kammerstetter [42]	🌐 💻	🌐 ◪	○	○	0 1	MIPS	✓		
	Pretender [36]	🌐 💻 ◪	—	○	○	3	ARM	✓	✓	✓
	Prospect [43]	🌐 💻	🌐 ◪	○	○	0 1	MIPS	✓		
	Surrogates [47]	🌐 💻	○	○	○	0 1 2 3	ARM	✓		
Symbolic Abstractions	FIE [23]	x	—	○	○	3	MSP430	✓	✓	
	FirmUSB [38]	x	—	○	○	3	8051/52	✓		✓
	Firmalice [72]	x	○ x	○	○ x	1 2	ARM, PPC	✓		
	Laelaps [8]	x	○	○	○	2 3	ARM	✓	✓	~
Hybrid Approaches	Avatar [86]	🌐 💻 x	🌐 ○	🌐 ○	🌐 ○ x	2 3	ARM	✓	✓	✓
	Inception [18]	🌐 💻 x	○	○	○	2 3	ARM		✓	~
	Mousse [52]	🌐	🌐 ○	○	○ x	1	ARM		✓	~

[a]Partial availability is indicated via ~.

rehosting that does not require expert knowledge or manual implementation effort. Despite their distinct end goals, both sets of users would benefit from research advancements that improve and potentially automate the rehosting process.

5.2 State of the Art

Due to the diversity of hardware platforms and dearth of documentation, it is rarely possible to create an HES that models every layer of even a single embedded system. As a result, RESs are commonly used as a less precise alternative to enable analyses that produce meaningful results about the original system. Rehosting systems must make decisions about how to model each layer of target embedded systems informed by the desired analysis outcomes. These systems make different trade-offs regarding which layers should be emulated precisely, modeled with some approximation, or passed through to a physical embedded system.

In Table 4, we systematize existing work, contrasting how rehosting systems handle various abstraction layers of target systems. At a given layer, a particular system may choose to emulate a component (💻), replace it (◪), model it symbolically (x), pass it through to real hardware (🌐), or leave it unmodified (○).

We also identify four broad approaches to rehosting: pure emulation, hardware-in-the-loop emulation, symbolic modeling of peripherals, and hybrid systems that combine hardware-in-the-loop with symbolic peripheral models. We refer readers interested in the historical relationships between these works to Appendix C. Lastly, a concurrent survey by Wright et al. [83] provides additional information on rehosting fidelity and deployed analysis techniques for common rehosting tools and approaches.

5.2.1 Pure Emulation. The most straightforward, though labor-intensive approach, to building a VE is emulating all the necessary components. As previously shown, building complete emulators for every component of an embedded system (i.e., an HES) is difficult and cannot scale. However, it is still challenging to identify and model a minimum set of necessary peripherals and features when building an RES. To simplify the process, an analyst may choose to ignore interactions with some peripherals, manually implement models, substitute peripherals with similar peripherals that have been modeled, or fall back on other rehosting strategies.

Ideally, pure emulation approaches would refrain from modifying the OS, application, or function layers and only replace the hardware and physical layers with emulated models to support detection of software vulnerabilities (e.g., memory corruption) but not hardware vulnerabilities. In practice, emulation-based rehosting techniques often modify the firmware to simplify the rehosting process and, in the process, excise some of the firmware from the VE. Type 1 firmware are commonly rehosted by using a generic kernel combined with the firmware's file-system in order to run user space binaries [10, 20, 88].

With Type-2 and Type-3 firmware, one rehosting technique is to allow an analyst to manually define models of peripheral behavior at higher abstraction layers [15–17, 50, 54]. For example, HALucinator [15] hooks calls into vendor-specific Hardware Abstraction Layer (HAL) functions and replaces them with Python approximations of the requested hardware functionality. An alternative approach is building peripheral models at lower abstraction layers [29, 37, 57]. For example, P2IM [29] observes MMIO access patterns in order to apply a pre-defined behavioral model.

5.2.2 Hardware-in-the-Loop Schemes. This approach, often referred to as *partial emulation*, addresses the problem of missing peripheral models by forwarding device interactions to the real hardware or extracting live snapshots from the running device. The software layers from a system are moved into a VE and mutated such that OS-hardware interactions are passed through to unmodified hardware running in the physical world.

This method yields high-fidelity models of the hardware-layer. In addition to physical hardware, this approach typically requires debugging access to the original execution environment, which is seldom present by default and often difficult to obtain. Additionally, the complexity added by the forwarding interface often leads to very high execution overheads (e.g., latency), limiting support for certain peripheral classes. Hardware-in-the-loop solutions generally do not scale since these solutions require a physical system paired with each VE. One notable exception is Pretender [36], which requires hardware only during a training phase in which peripheral models are generated from observations of real hardware behavior.

While most hardware-in-the-loop systems modify QEMU to use it as a VXE [36, 42, 43, 47, 49, 59, 69], relying purely on emulation is not a requirement, as demonstrated by Charm [77]. This system runs Android device drivers for ARM devices in a virtualized x86 environment and forwards MMIO to a physical device via USB 3.0. This design provides low-latency forwarding and high execution speeds, enabling Charm to fuzz device drivers.

5.2.3 Symbolic Abstractions. Another method to model hardware in VEs is to emulate software layers and consider all the values read from hardware to be symbolic. These approaches require a symbolic VXE such as KLEE [7] or S2E [13]. FIE [23], for example, uses KLEE as a VXE to symbolically execute the firmware while allowing for every valid interrupt to be raised at every instruction. This technique typically over-approximates hardware capabilities by assuming every peripheral is capable of returning the full range of possible values, which may lead to false positive analysis results and cause state-space explosion, even for small firmware programs.

5.2.4 Hybrid Approaches. Some rehosting approaches combine hardware-in-the-loop schemes with symbolic execution to allow for more flexible analysis scenarios such as bug finding or reverse engineering of hardware components.

Inception [18] is one such full-system hybrid solution. It consists of a custom JTAG debugger for near real-time hardware forwarding, a symbolic VXE based on KLEE, and a translator for merging lifted and compiled LLVM bitcode to cope with inline assembly. However, its implementation is tied to the ARM Cortex-M3 microcontroller and requires the firmware's source code, constraining its usability.

5.3 Effects of Different System Types

Naturally, approaches to crafting VEs differ depending on the target system. While Type-1 systems are generally the most complex class of embedded system, their operating systems commonly provide clear hardware abstractions. As a result, applications on these systems are self-contained, rarely interacting directly with hardware. In many cases, this means the kernel and drivers can be replaced in order to ease integration with a VE. This approach is often used by the systems with replaced (◪) OS layers as shown in Table 4.

On the other hand, there are fewer hardware abstractions present in Type-2 systems and none in Type-3 systems. While the underlying hardware in these systems is generally simpler, the amount of hardware modeling required to build a working VE is often higher. Notably, to our knowledge, no existing approaches to rehosting Type-2 systems make use of abstractions provided by the OS.

6 THE REHOSTING PROCESS

We study the rehosting approaches outlined in § 5.2 to identify common patterns and articulate a common rehosting process. We model rehosting as the process of building a specification for an RES and then iteratively evaluating and refining the specification until its fidelity is satisfactory. Although the rehosting process does not fundamentally require iterative refinement, the information available even in a low-fidelity VE is commonly used as it provides invaluable insights into the requirements of an RES.

Rehosting an embedded system S begins with an initial specification of an RES. A specification R is a 4-tuple that defines a rehosted system and consists of a VXE (cpu), firmware to execute (fw), models for its n peripherals ($\{p^j \mid j \in 1, 2, 3 \ldots n\}$), and miscellaneous configuration data (d).

6.1 Iterative Refinement

After the initial specification, R_0, is created, it can immediately be subject to detailed observations and analysis, even before its behavior sufficiently mirrors S. Clearly, such analyses cannot produce meaningful results with respect to S while these behaviors diverge, but they can instead be used to guide the rehosting process by revealing which component of R_0 led to divergent behavior. For example, if the VXE (cpu) fails to execute an instruction in fw, errors in the construction of cpu may become apparent. Alternatively, if an insufficient peripheral model returns a value that causes a divergence, a dynamic taint analysis can identify the deficiency in the model. As such, iterative evaluation and refinement will greatly aid the generation of an accurate RES. Fig. 3 captures this process in detail. To represent the RES through iterations of this process, we define the ith iteration to be:

$$R_i = (cpu_i, fw_i, \{p_i^j\}, d_i)$$

To build the initial specification, R_0, an analyst must obtain the firmware for the system, fw_0. Static analysis of fw_0 can identify the ISA for the original CPU, $\overline{cpu}$. To execute fw_0, there must be a VXE available for its ISA. If none is available, one must be developed, a complex task even if the ISA is well-documented.

In addition to a VXE for the ISA of $\overline{cpu}$, models for each peripheral with which fw_0 interacts, $\bar{p}^j$, must be developed as necessary. The aforementioned approaches to peripheral modeling fit into this model as follows. *Pure emulation* creates virtual models of peripherals comparable to the original peripherals: $\forall j : p^j \approx \bar{p}^j$. *Hardware-in-the-loop* configures cpu to pass interactions with peripherals p^j to $\bar{p}^j$ over a debugging channel between cpu and $\overline{cpu}$: $\forall j : p^j = \bar{p}^j$, sans latency. *Symbolic abstractions* treat peripheral outputs as unconstrained symbolic values ($\forall j : p^j \supset \bar{p}^j$) and leverage symbolic execution to build a VE. *Hybrid approaches* combine techniques from the prior two approaches to support more flexible analyses.

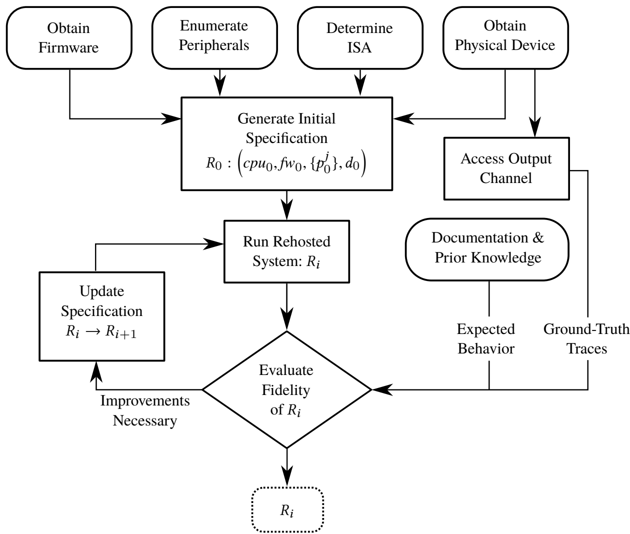


Figure 3: The process of rehosting an embedded system. Iteration improves fidelity by updating the RES specification.

While the initial R_0 may not be perfect, cpu_0 must be fairly accurate for the VE to run fw_0 in a meaningful way. By contrast dynamic analysis can more easily identify incorrectly modeled peripherals or invalid configuration data. While it may be theoretically possible to build an accurate specification in a single attempt, in practice an iterative dynamic analysis process is what makes RES construction feasible.

R_i is executed as follows, where $i=0$ in the first iteration: fw_i is run using cpu_i with peripherals p_i^j and configuration data d_i. Initial attempts to use R_i will likely fail to sufficiently mirror S, but since rehosted systems are introspectable, the execution can be analyzed and various traces extracted to aid diagnostics. These traces may be collected by capturing the execution state at every instruction, system call, or at any other time of interest. Traces from R_i have more diagnostic value if equivalent traces can be collected from the real device, S, via an *output channel* such as hardware debug support (e.g., JTAG) or extractable software logs (e.g., Linux `ftrace`).

Evaluating the fidelity of R_i with comparisons of ground-truth traces or expected behavior from S to observations of R_i is essential for determining when the rehosting process has finished. Access to the physical device can make assessing fidelity easier. Without the physical device, assessments can only be approximated using prior knowledge of the system's expected behavior. For example, the gateway functionality (e.g., DHCP, NAT support) of a router's firmware could be tested, but high-level functional testing is a very coarse measure of fidelity.

If the fidelity of R_i is unsatisfactory, it may be improved by modifying its components to produce R_{i+1}. This might mean correcting instruction decoding errors in cpu_{i+1} or excising irrelevant parts of the firmware in fw_{i+1}. Alternatively, parameters or data in d_{i+1} could be changed or a peripheral model, p_{i+1}^j, may be updated. After making these changes, R_{i+1} should be run so that iterative refinement can continue. After some number of refinements, the fidelity of R_i should become satisfactory if all reasonable system modifications are considered. Once the rehosting process is complete, the final R can be saved to use for dynamic analysis.

7 REHOSTING ROADMAP

In this section, we identify and discuss rehosting roadblocks that require addressing. We organize these into a roadmap for future research and development with the hope of guiding the community toward a future in which rehosting is a well-understood, systematic, and scientific endeavor. We identify the following significant obstacles to the rehosting process that could be improved by future work:

(1) Building VXEs for new CPUs/ISAs;
(2) Widespread adoption of modeling standards;
(3) Handling peripheral behavior;
(4) Quantifying fidelity of a rehosted system; and
(5) Facilitating rehosting for complex systems

7.1 Creating Virtual Execution Engines

Building a VE may require an analyst to implement a new VXE. When documentation or prior knowledge about an ISA is available, building a VXE is a straightforward, but significant, undertaking. Without such information, an in-depth analysis of system binaries may occasionally enable development of an emulator [28, 44], but this is an incredibly challenging task.

The difficulty of building VXEs, a critical piece of rehosting, means that the vast majority of existing work focuses on widely-used and well-documented ISAs. This is reflected in the supported ISAs shown in Table 4 where the majority of tools focus solely on rehosting ARM targets, MIPS targets, or both in their evaluation. PowerPC, MSP430, and 8081 are targeted by only one tool each, and other commonly deployed architectures—such as Xtensa, AVR, RISC-V, and SPARC—are not represented at all.

Despite the current focus on ARM and MIPS on Linux, the embedded systems in the wild are more diverse. While it is difficult to pinpoint the distribution of ISAs and OSs, prior studies provide an estimate by crawling the Internet for firmware images. For instance, in the dataset Chen et al. acquired for Firmadyne [10], 82% of firmware images ran on MIPS, 10% were ARM (neglecting endianness and bitwidth), and approximately 41% of all acquired images were based on Linux. Another large-scale study within the same order of magnitude [19] reports the acquisition of a dataset in which 63% of the firmware is for ARM devices, 7% for MIPS, and 86% for Linux. These numbers differ largely due to changes in source selection and processing of the datasets, but both show that a significant amount of firmware is not written for ARM and MIPS on Linux.

Given the significant number of embedded devices not running Linux on ARM or MIPS, an important research task is to find ways of making it easier to create fast and accurate VXEs. There has been some encouraging recent work on automated synthesis of semantic specifications for specific ISAs such as x86 [34, 39]. TaintInduce [14] shows that higher level semantics (i.e., taint propagation rules) can be dynamically inferred for an ISA. However, these approaches focus on simple instructions, such as arithmetic and basic logical operations. To create fully-fledged VXEs for real-world CPUs, complex instructions that manipulate hidden CPU states (e.g.,

instructions that change privilege levels) or perform complex high-level tasks (e.g., the AES-NI instructions on x86) may need to be handled.

Although various languages for describing and specifying CPU behaviour can create emulators and simulators (e.g., Sleigh [33], Sled [66], and Verilog [40]), the process of creating these specifications is manual and error-prone. We believe research into extending automated synthesis to complex instructions and ISAs found in embedded systems could ease the difficulty of rehosting systems that use proprietary, legacy, or merely unpopular CPUs and architectures.

7.2 Widespread Adoption of Modeling Standards

Of the standard formats commonly used today to describe hardware layouts and encode peripheral metadata (e.g., Device Trees [24], ACPI tables [31], SVD files [2]), none encode hardware behavior. If such a format was widely adopted, VEs could ingest abstract configurations that encode peripheral and CPU behaviors to use as drop-in replacements or shims for full implementations of each.

This lack of widespread adoption is not due to a lack of standardization. The Open Virtual Platforms (OVP) project provides a collection of APIs for modeling peripheral and VEs as well as a repository of generated models [53]. Components of embedded systems can be modeled using OVP's APIs in a standardized fashion and consumed by multiple emulators. OVP has partnered with numerous semiconductor design companies including ARM and MIPS to validate behavior of its models. Another notable standard is SystemC [62], a hardware modeling platform for behavioral and system levels. Although hardware modeled in one of these standards cannot easily be converted into the other, emulators such as QEMU and OVPsim can be integrated with either of these standards [21, 58]. Future rehosting research should leverage these standards to build off existing hardware models and to produce results that could be consumed by subsequent work.

7.3 Handling Peripherals

A critical, yet challenging, component of building a high-fidelity VE is generating an accurate model of all the peripherals with which the firmware will interact. Between the early drafts of this paper in 2018 and our current submission, researchers have started down the path to automated peripheral modeling we describe here (in particular, Pretender [36], and Laelaps [8]). We find it encouraging that the research community has begun to realize the importance of automated emulation of embedded devices, but stress that open problems remain formidable. We identify at least five possible approaches to handle embedded peripherals.

First, if prior knowledge of peripheral behavior or documentation is available, a model could be manually constructed or a pre-built model could be leveraged. Although § 4 shows that this approach cannot scale to the diversity of peripherals found in the wild, it may be possible to at least make the manual effort invested reusable through the use of standardized models described in § 7.2.

A second approach is to attempt to automatically create a model of a peripheral through analysis of its firmware or the behavior of the physical system. Pretender [36] shows that MMIO traces from a physical system can be used to infer some peripheral models, but additional work is necessary to model complex peripherals. Alternatively peripheral models could be generated by analyzing driver code using symbolic execution (e.g., Laelaps [8]), fuzzing, or static analysis.

Third, if sufficient instrumentation capabilities are available on a target system, peripherals could be probed with inputs and models constructed to describe observed outputs. Subramanyan et al. [76] demonstrated that some cryptographic co-processors could automatically be modeled by creating peripheral templates and then using program synthesis from I/O samples to automatically synthesize a working peripheral model. This approach, if extended to other common embedded peripherals, such as UARTs and timers, could greatly ease the burden of peripheral modeling.

A fourth approach is to replace a peripheral with another that is already modeled, remove it entirely or replace it with a symbolic peripheral model. While prior approaches [12, 59, 68] have used symbolic peripheral models, they quickly encounter problems due to state explosion. If analysis indicates that replacing or removing a peripheral will have insignificant effects on a system's behavior, such a change can be an effective option.

A final approach is to intercept requests at higher layers of abstraction that ultimately lead to peripheral interactions and build models there. This approach, known as *high-level emulation*, precludes analysis of potentially vulnerable driver code but enables reuse of peripheral models when common peripheral interfaces can be identified across multiple embedded systems [15].

Even with these approaches, peripherals that provide unavailable, high-entropy data will often be impossible to sufficiently model. For example, a model of a storage controller built without knowledge of any the underlying file system data could lead an RES to an unbounded state if any code from that file system was executed. Regardless of implementation specifics, new capabilities to handle device peripherals are necessary, given the asymmetry between peripheral diversity in the wild and manually developed peripheral models.

7.4 Formalizing Fidelity

The fidelity evaluations and comparisons commonly used in the rehosting literature to date are informal and as such, prevent meaningful comparisons between rehosting techniques. As identifying when two systems are equivalent is an unsolvable problem in the general case, we do not believe future rehosting techniques will ever guarantee flawless fidelity for non-trivial systems. However, we see two distinct scenarios for evaluating rehosting fidelity and suggest more principled approaches to both.

Some rehosting techniques, particularly those that model peripherals with symbolic abstractions, are designed to over-approximate behavior of physical systems. These techniques aim to reduce false negatives by analyzing every possible state; however, this increases the rate of false positives as infeasible states will also be analyzed. As such, the fidelity evaluation for these rehosting techniques must differ from those that aim to precisely replicate a system's behavior. Quantifying the fidelity of an over-approximated RES should compare concrete states from the physical system to reachable states in the RES. Such states could be memory snapshots or traces of instructions executed. A high-fidelity, over-approximated RES should be able to reproduce every state from a large collection of states.

Working with an RES that aims to precisely reproduce a physical system's behavior could be evaluated in terms of *input-output equivalence*. Providing an identical set of input states to the two versions of the system and comparing the outputs will reveal information about the fidelity of the RES. The confidence in this fidelity evaluation would depend on the scale and diversity of input states tested as well as the precision of the output comparison. The output comparison could be made more precise by modifying the systems to produce intermediate outputs through techniques such as binary rewriting (e.g., Ramblr [81]), or enabling non-standard logging. Perhaps an ideal technique for collecting intermediate outputs would build on program slicing techniques. Although current whole-system slicing techniques (i.e., Virtuoso [26]) fail to handle inter-process communication, peripheral behavior, or process creation, solutions to these shortcomings would enable precise capture of every intermediate output. If slices on output buffers of interest could be extracted from and compared between an RES and its physical counterpart, differences may reveal inequivalencies.

7.5 Rehosting of Complex Embedded Systems

The current state-of-the-art of rehosting revolves around firmware executed on a single CPU and is closely tied to the peripherals associated to that CPU. Yet, embedded systems usually consist of more than one processing unit and cyber-physical systems can easily comprise multiple different CPUs, specialized Digital Signal Processors (DSPs), custom Application-Specific Integrated Circuits (ASICs) and configurable FPGAs [6, 55, 56].

Existing rehosting systems either ignore these components or model them as peripherals. However, similar to multi-layer vulnerabilities discussed in § 2.1, some vulnerabilities may only be observable when the interactions between the components are captured thoroughly in a rehosted system. Hence, we believe future rehosting approaches will need to investigate computing units beyond traditional general purpose processors, as well as the interaction between multiple rehosted components.

8 CONCLUSION

Rehosting is an important capability that enables the application of powerful dynamic analysis techniques such as fuzzing and symbolic execution to embedded systems. While prior work has attempted to develop ad-hoc solutions to rehosting in the pursuit of other research goals, we argue that rehosting is a research problem in its own right and, as such, should be approached systematically.

In this paper, we disambiguate the field of rehosting from emulation and show that building complete hardware emulation systems is both unnecessary to enable dynamic analysis of firmware and impossible to scale. We propose a taxonomy of rehosting strategies, highlighting the differences between preliminary approaches in a systematic fashion. We identify the essential steps in the rehosting process and a high-level, iterative process for rehosting embedded systems. Finally, we describe unsolved rehosting challenges and propose a roadmap for future research in this space.

By improving the rehosting process, the security community will finally be able to apply decades of dynamic analysis research and mature tooling to the world of embedded systems. We hope that this systematization, together with our suggested future research directions, will spawn new lines of rehosting research, well-equipped to provide the foundation of successful security analysis platforms for current and future embedded systems.

ACKNOWLEDGMENTS

The authors wish to thank the following individuals for their contributions and support: Lindsey Wang, John Wilkinson, Douglas E. Stetson, William Hedberg, and Greta Lepore. This work was in part funded by ONR Awards N00014-15-1-2180 and N00014-19-1-2364; the National Science Foundation under Grants No. CNS-1916398 and CNS-1942793; NWO 628.001.030 "Tropics" and NWO NWA-ORC InterSect; and a research contract with Siemens AG. DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering, Office of Naval Research, or the National Science Foundation.

REFERENCES

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *USENIX Security*, 2017.

[2] ARM. System view description. https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[3] N. Artenstein. Broadpwn: Remotely compromising android and ios via a bug in the broadcom wi-fi chipset, 2017.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[5] N. Brown. Device trees i: Are we having fun yet? https://lwn.net/Articles/572692/.

[6] P. Burgio, C. Alvarez, E. Ayguadé, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, and R. Giorgi. Simulating next-generation cyber-physical computing platforms. *Ada User Journal*, 37, 2016.

[7] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[8] C. Cao, L. Guan, J. Ming, and P. Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *ACSAC*. ACM, 2020.

[9] A. Caraceni, F. De Cristofaro, F. Ferrara, S. Scala, and O. Philipp. Benefits of using a real-time engine model during engine ecu development. Technical report, SAE Technical Paper, 2003.

[10] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016.

[11] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *IEEE/IFIP DSN*, 2018.

[12] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *ACM EUROSYS*.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, 2011.

[14] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su. One engine to serve'em all: Inferring taint rules without architectural semantics. In *NDSS*, 2019.

[15] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, et al. HALucinator: Firmware re-hosting through abstraction layer emulation. In *USENIX Security*, 2020.

[16] A. A. Clements, L. Carpenter, W. A. Moeglein, and C. Wright. Is your firmware real or re-hosted? a case study in re-hosting vxworks control system firmware. In *BAR*, 2021.

[17] Comsecuris. Luaqemu. https://github.com/comsecuris/luaqemu.

[18] N. Corteggiani, G. Camurati, and A. Francillon. Inception: system-wide security testing of real-world embedded systems software. In *USENIX Security*, 2018.

[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[20] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ACM ASIA CCS*, 2016.

[21] F. Cucchetto, A. Lonardi, and G. Pravadelli. A common architecture for co-simulation of systemc models in qemu and ovp virtual platforms. In *IEEE VLSI-SoC*, 2014.

[22] Y. David, N. Partush, and E. Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 2018.

[23] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.

[24] Devicetree.org. Device tree specification v0.2. https://www.devicetree.org/specifications/, 2017.

[25] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[26] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *IEEE SP*, 2011.

[27] M. D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *ACM SIGPLAN-SIGSOFT PASTE*, 2004.

[28] fail0verflow. Unprogramming: Intro. https://fail0verflow.com/blog/2012/unprogramming-intro/, 2012.

[29] B. Feng, A. Mera, and L. Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *USENIX Security*, 2020.

[30] S. Fleming. Accessing pci express configuration registers using intel chipsets. *Intel White Paper*, (321090), 2008.

[31] U. E. F. I. Forum. Advanced configuration and powerinterface specification v6.2. https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf, 2017.

[32] FTDI. Simplified description of usb device enumeration. https://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_113_SimplifiedDescriptionofUSBDeviceEnumeration.pdf, 2009.

[33] Ghidra. SLEIGH - A Language for Rapid Processor Specification.

[34] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *ACM SIGPLAN PLDI*, 2012.

[35] Z. Gui, H. Shu, F. Kang, and X. Xiong. Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution. *IEEE Access*, 2020.

[36] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *RAID*, 2019.

[37] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *USENIX Security*, 2020.

[38] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *ACM SIGSAC*, 2017.

[39] S. Heule, E. Schkufza, R. Sharma, and A. Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *ACM SIGNPLAN PLDI*, 2016.

[40] IEEE Computer Society. Std 1364: IEEE Standard for Verilog Hardware Description Language. 1995.

[41] M. J. Jung and T. Ballo. Stm-based introspection. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[42] M. Kammerstetter, D. Burian, and W. Kastner. Embedded security testing with peripheral device caching and runtime program state approximation. In *SECURWARE*, 2016.

[43] M. Kammerstetter, C. Platzer, and W. Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM ASIA CCS*, 2014.

[44] P.-H. Kamp. The crypto-cs-seti challenge: An un-programming challenge. http://web.archive.org/web/20160304030848/http://queue.acm.org/unprogramming.cfm, 2012.

[45] U. Kargén and N. Shahmehri. Speeding up bug finding using focused fuzzing. In *ACM ARES*, 2018.

[46] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *ACSAC*. ACM, 2020.

[47] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Usenix WOOT*, 2015.

[48] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996.

[49] H. Li, D. Tong, K. Huang, and X. Cheng. Femu: A firmware-based emulation framework for soc verification. In *IEEE/ACM/IFIP CODES+ISSS*, 2010.

[50] W. Li, L. Guan, J. Lin, J. Shi, and F. Li. From library portability to para-rehosting:natively executing microcontroller softwareon commodity hardware. In *NDSS*, 2021.

[51] G. Likely. Linux and the device tree: The linux usage model for device tree data.

[52] Y. Liu, H.-W. Hung, and A. A. Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In *ACM EuroSys*, 2020.

[53] I. S. Ltd. Openvirtualplatforms. http://www.ovpworld.org/, 2019.

[54] D. Maier, L. Seidel, and S. Park. Basesafe: Baseband sanitized fuzzing through emulation. In *ACM WiSec*, 2020.

[55] A. Malinowski and H. Yu. Comparison of embedded system design for industrial applications. *IEEE transactions on industrial informatics*, 7, 2011.

[56] P. Marwedel. Embedded and cyber-physical systems in a nutshell. *DAC.COM Knowledge Center Article*, 2010.

[57] A. Mera, B. Feng, L. Lu, E. Kirda, and W. Robertson. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. *To appear at IEEE SP*, 2021.

[58] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed sw/systemc soc emulation framework. In *IEEE ISIE*, 2007.

[59] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti. Avatar²: A Multi-target Orchestration Platform. In *BAR*, 2018.

[60] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[61] J. Obermaier and S. Tatschner. Shedding too much light on a microcontroller's firmware protection. In *USENIX WOOT*, 2017.

[62] P. R. Panda. Systemc: a modeling platform supporting multiple design abstractions. In *ACM ISSS*, 2001.

[63] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: force-executing binary programs for security applications. In *USENIX Security*, 2014.

[64] S. E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, et al. A survey on chip to system reverse engineering. *ACM JETC*, 2016.

[65] N. A. Quynh and D. H. Vu. Unicorn: Next generation cpu emulator framework. *BlackHat USA*, 2015.

[66] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *Transactions on Programming Languages and Systems*, 1997.

[67] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE SP*, 2020.

[68] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: testing drivers without devices. In *USENIX OSDI*, 2012.

[69] J. Ruge, J. Classen, F. Gringoli, and M. Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *USENIX Security*, 2020.

[70] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *SSTIC*, 2015.

[71] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE SP*, 2010.

[72] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[73] O. Shwartz, A. Cohen, A. Shabtai, and Y. Oren. Shattered trust: When replacement smartphone components attack. In *USENIX WOOT*, 2017.

[74] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 NDSS*, 2019.

[75] P. Stewin and I. Bystrov. Understanding dma malware. In *DIMVA*, 2013.

[76] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based synthesis of instruction-level abstractions for soc verification. In *FMCAD*, 2015.

[77] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, et al. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security*, 2018.

[78] O. Thomas. Integrated circuit reverse engineering and code dumping, 2019.

[79] R. Torrance and D. James. The state-of-the-art in ic reverse engineering. In *CHES*. Springer, 2009.

[80] S. Vasile, D. Oswald, and T. Chothia. Breaking all the things-a systematic survey of firmware extraction techniques for iot devices. In *Springer CARDIS*, 2018.

[81] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, et al. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[82] J. Wetzels. The rtos exploit mitigation blues. https://hardwear.io/document/rtos-exploit-mitigation-blues-hardwear-io.pdf, 2017.

[83] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM CSUR*, 2021.

[84] T.-C. Yeh, G.-F. Tseng, and M.-C. Chiang. A fast cycle-accurate instruction set simulator based on qemu and systemc for soc development. In *IEEE MELECON*, 2010.

[85] H. Ying, Y. Zhang, L. Han, Y. Cheng, J. Li, X. Ji, and W. Xu. Detecting buffer-overflow vulnerabilities in smart grid devices via automatic static analysis. In *IEEE ITNEC*, 2019.

[86] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[87] V. A. Zakharov. The equivalence problem for computational models: decidable and undecidable cases. In *Springer MCU*, 2001.

[88] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security*, 2019.

Table 5: The embedded systems zoo.

System	CPU	OS	Notable Peripherals
Canon Powershot G11	ARM	DryOS	Image sensor, HDMI
Gen. 1 Apple Airport Express	PPC	VxWorks	WiFi, Ethernet, USB
Google Nest Thermostat E	ARM	Linux	WiFi, Temp. sensor
HP M551dn Printer	ARM	WinCE	Ethernet, motor, daughterboard
Microtik RouterBoard 192	MIPS	RouterOS	Ethernet, speaker
Philips Hue Lightbulb	AVR	Custom	ZigBee and Bluetooth radios

A EXAMPLE EMBEDDED SYSTEMS

Table 5 describes embedded systems, their CPU architecture, operating system, and notable peripherals from each system.

B DETAILED SURVEY METHODOLOGY

B.1 Availability: Execution Engines

Our DTB corpus contains "manufacturer,model" tuples describing the CPU used by each hardware platform. For example, the DTB for the exynos5250 SoC indicates that it uses an ARM Cortex-A15 CPU (`compatible = "arm,cortex-a15";`). After we strip manufacturer name, this corresponds to QEMU's `cortex-a15` CPU.

Because strings describing the same CPU model may differ slightly, we used fuzzy string matching (Levenshtein distance) after stripping manufacturer name from compatible strings to make processor support determinations, e.g., QEMU's `mpc8541e_v11` processor name matches the manufacturer-less PPC compatible string `8541`. Matches were manually reviewed for accuracy. Due to QEMU's extremely limited support for ARM Cortex-M processors and SoCs (only 2 SoCs), we do not consider the systems described by our SVD corpus for this analysis.

We never assume that a core can be safely swapped for another of the same ISA version, e.g. replacing a `cortex-a9` with a `cortex-a15` - both `ARMv7-A` CPUs. Despite QEMU's lack of micro-architecture behavior modeling, such a substitution can lead to a range of errors - including illegal instructions, differing sets of configuration registers, and variations in MMU features.

B.2 Diversity: Unique Peripherals

For our DTB corpus, we extract compatible strings from each system's description. The Device Tree Standard indicates that the `compatible` property of a node, a key whose value is a list of precedence-ordered strings in the format `"manufacturer,model"`, should be used for device driver selection. For example, when the Linux kernel parses a device tree node with property `compatible = "samsung,exynos3250-pmu"` it determines that it must load the device driver implemented in `drivers/soc/samsung/exynos-pmu.c`. If a node's `compatible` property lists multiple strings, we consider only the first. This reflects the kernel's selection precedence and ensures no physical peripheral is counted more than once. DTB nodes that do not contain a `compatible` property are ignored. This conservative approach ensures we only count peripherals we are certain an OS can interact with directly.

For the SVD corpus identifying unique peripherals is less straightforward as peripheral names need not directly correspond to driver code. We consider two peripherals to be the same if they have the same register layout within the memory mapped I/O (MMIO) interface to the peripheral, raise the same interrupt signals, and have similar names (normalized Levenshtein distance ≤ 0.20). Again, this is a fairly conservative approach as divergences in register layouts and interrupts necessitate different peripheral behavior but similarities do not guarantee that two configurations refer to the same peripheral.

Because QEMU has no central table of supported peripherals, we programmatically collect and clean the output of the `-device help` flag for all board definitions for every surveyed architecture. OVPsim peripheral counts are determined by available models advertised on the project's homepage [53].

B.3 Complexity: Driver SLOC as a Proxy

Note that submission of Device Tree files to the Linux kernel does not obligate submission of source code for drivers named therein; we observed that 20-58% of drivers were open source, depending on architecture. When totaling SLOC per SoC, we use precise counts for open-source drivers and the average SLOC per driver count (architecture-specific) for closed-source drivers.

Manual analysis of a small subset ($n = 20$) of QEMU-implemented peripherals demonstrated a positive correlation between driver and peripheral implementation SLOC as shown in Fig. 4.

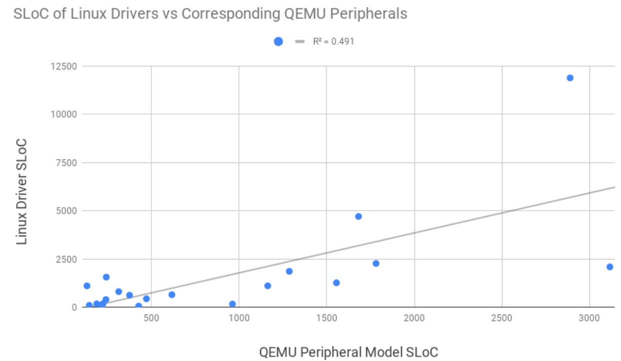


Figure 4: SLOC of Peripheral Model vs Device Driver.

Unlike our tabulation of peripheral diversity, here we consider all compatible strings present, not just the first (preferred) per node. This aids completeness, as we do not miss the opportunity to count SLOC for any open drivers. SLOC is computed with `pygount`, a Python library that supports C syntax and does not count comments or empty lines.

Automating SLOC measurement for QEMU peripheral implementations is infeasible as they can be incomplete, tightly coupled with QEMU-internal objects, or spread across hierarchical source files. For example, *hw/cpu/a9mpcore.c* extends QEMU's CPU class to

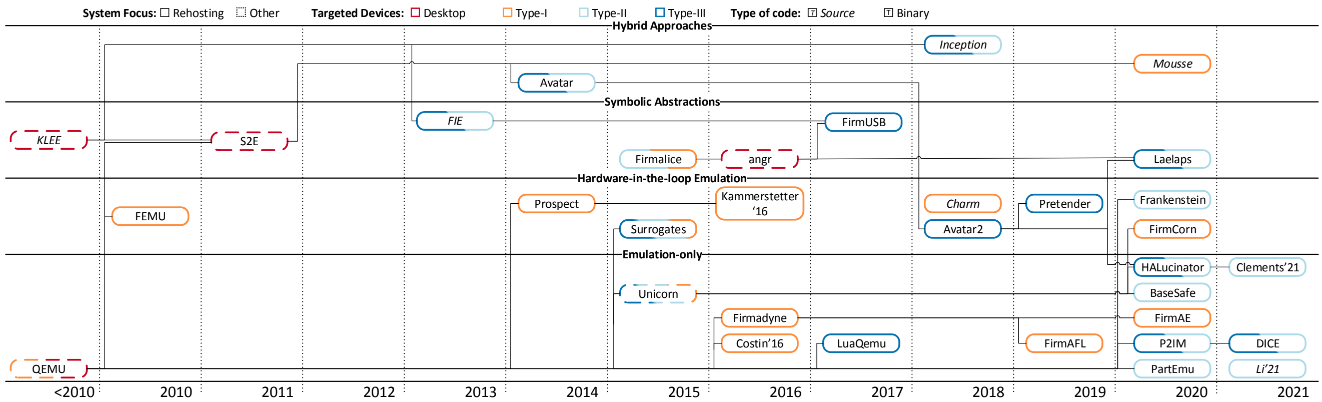


Figure 5: Timeline of Rehosting Systems.

Table 6: Total Peripherals Supported by QEMU.

Arch	v2.11.1 Total	v4.2.0 Total	v5.2.0 Total
ARM	227	321	337
ARM64	279	322	338
MIPS	153	186	192
PPC	160	210	216

add support for multiple Cortex-A9 peripherals (GIC, SCU, timers, etc.).

B.4 Aside: Peripheral Support across QEMU Versions

Looking at three major versions of QEMU approximately a year apart, v2.11.1 (February 2018, latest in Ubuntu 18.04 repositories), v4.2.0 (December 2019), and v5.2.0 (December 2020), Table 6 shows very little increase relative to corpus peripheral diversity (see Table 2 in § 4).

Regardless of which QEMU version we contrast § 4 results against, the outcome is the same. Modern QEMU is not meaningfully more capable of emulating embedded systems than it was 2.5 years ago. Despite the increasing attention the research community has given rehosting, on the HES front there has been no meaningful change. Our Monte Carlo simulation demonstrated the problem of robust peripheral support is intractable going forward, historical data complements this conclusion by demonstrating an insignificant rate of support increase.

C HISTORICAL TAXONOMY OF PRIOR WORK

Table 4 does not capture temporal or evolutionary relationships between prior work. Toward this end, we present a timeline of rehosting solutions and rehosting-related work in Fig. 5. Note that target system type, source dependecy, and primary goal (rehosting or other) are encoded in the figure.