Conware: Automated Modeling of Hardware Peripherals

Chad Spensky
Allthenticate
UC Santa Barbara
MIT Lincoln Laboratory
Goleta, CA, USA
chad@allthenticate.net

Aravind Machiry
Purdue University
West Lafayette, IN, USA
amachiry@purdue.edu

Nilo Redini
UC Santa Barbara
Goleta, CA, USA
nredini@cs.ucsb.edu

Colin Unger
UC Santa Barbara
Goleta, CA, USA
colinunger@ucsb.edu

Graham Foster
UC Santa Barbara
Goleta, CA, USA
gmfoster@ucsb.edu

Evan Blasband
Allthenticate
UC Santa Barbara
Goleta, CA, USA
evan@allthenticate.net

Hamed Okhravi
MIT Lincoln Laboratory
Lexington, MA, USA
hamed.okhravi@ll.mit.edu

Christopher Kruegel
UC Santa Barbara
Goleta, CA, USA
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
Goleta, CA, USA
vigna@cs.ucsb.edu

ABSTRACT

Emulation is at the core of many security analyses. However, emulating embedded systems is still not possible in most cases. To facilitate this critical analysis, we present Conware, a hardware emulation framework that can automatically generate models for hardware peripherals, which alleviates one of the major challenges currently hindering embedded systems emulation. Conware enables individual peripherals to be modeled, exported, and combined with other peripherals in a pluggable fashion. Conware achieves this by first obtaining a recording of the low-level hardware interactions between the firmware and the peripheral, using either existing methods or our source-code instrumentation technique. These recordings are then used to create high-fidelity automata representations of the peripheral using novel automata-generation techniques. The various models can then be *merged* to facilitate full-system emulation of *any* embedded firmware that uses *any* of the modeled peripherals, even if that specific firmware or its target hardware was never directly instrumented. Indeed, we demonstrate that Conware is able to successfully emulate a peripheral-heavy firmware binary that was never instrumented, by merging the models of six unique peripherals that were trained on a development board using only the vendor-provided example code.

CCS CONCEPTS

• **Computer systems organization** → **Firmware**; **Embedded hardware**; **Embedded software**.

KEYWORDS

embedded systems, emulation, hardware peripherals

ACM Reference Format:
Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. 2021. Conware: Automated Modeling of Hardware Peripherals. 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/10.1145/3433210.3437532

1 INTRODUCTION

When presented with a system to analyze, the first step usually involves performing dynamic analyses to understand how the system works. Unfortunately, in the world of embedded systems, which are the most prolific type of computing devices today, this simple step is not trivial, and, in many cases, it is prohibitively expensive. In fact, no approach exists today that can emulate embedded systems in the general case. This problem stems from a few key challenges: *1)* embedded systems run on a variety of architectures (*e.g.,* ARM, MIPS, or AVR), *2)* some systems implement custom instruction set architectures (ISAs) with undocumented instructions, *3)* embedded software has strict hardware dependencies with little-to-no hardware abstraction software, *4)* the firmware's code is typically interrupt driven, and *5)* embedded code depends on a multitude of peripherals, which are unique for each system.

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, 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.

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

ACM ISBN 978-1-4503-8287-8/21/06…$15.00
https://doi.org/10.1145/3433210.3437532

While there have been multiple approaches that address various aspects of the emulation problem, it is still far from "solved." For example, researchers have put significant manual effort into systems like the Quick EMUlator (QEMU) [5] to support more architectures and instructions. Similarly, hardware-in-the-loop record and replay systems (*e.g.,* Avatar [37] and Pretender [17]) are capable of replaying hardware interactions that were obtained through a debug interface. Finally, recent proposals that depend on a hardware abstraction layer (HAL) being present (*e.g.,* HALucinator [9] and Firmadyne [8]) are able to satisfy hardware interactions by manually implementing specific routines (*e.g.,* `read_from_uart`). However, no system has addressed the problem of efficiently emulating peripherals, on which the majority of these embedded systems (and their emulation) depends. In theory, these peripherals could be manually implemented for a specific system, but the sheer number of peripherals, and the rate at which new peripherals are being introduced to the market, make manual implementation intractable.

Peripherals interactions are typically implemented directly as memory-mapped input and output (MMIO), where interactions with the peripheral appear as normal memory reads and writes. While traditional operating systems (OSes) implement a HAL to interact with hardware, in the form of drivers and common interfaces, embedded systems typically have their own custom HAL or interact with the peripherals directly. Herein lies the problem — the values returned from an MMIO read are unknown to any analysis that does not have access to the hardware (either directly or indirectly). As such, the analysis must over-approximate the range of possible values, or infer them through best-guess heuristics by analyzing the firmware mounted by the considered system. For static analyses, this over-approximation might introduce a significant overhead, which can make the analysis intractable [10], and cause a loss of precision (*e.g.,* due to numerous spurious program states that would not be possible during any real execution of the system). In the case of dynamic analysis, the inability to return an expected value will likely result in the execution stalling indefinitely or entering some error-handling code. Either way, the analysis would likely fail to delve deep into the firmware code.

To make matters worse, most embedded systems are heavily dependent on interrupts that are issued from external peripherals (*e.g.,* when data has arrived) to advance their execution. In fact, it is common for embedded firmware to be completely interrupt-driven (*i.e.,* the firmware will only perform a simple control loop until an interrupt is issued). An emulator that is incapable of issuing these interrupts will be unable to achieve any realistic functionality, as it will never execute the interrupt handlers.

The goal of this work is to automatically create a software-based version of a hardware peripheral that is capable of *conn*ing the firmware into believing that the actual hard*ware* peripheral is present by returning valid MMIO values and issuing relevant interrupts. Indeed, our current technique can be employed on Linux-based, RTOS-based, and bare-metal [28], but the scope of this paper is limited to bare-metal systems. While our work is currently limited to MMIO-based systems (*e.g.,* ARM, MIPS, SPRAC), we believe the techniques could be extended in future work to include a more diverse set of peripheral interactions (*e.g.,* port-mapped input and output that is seen in PIC and Intel systems). Similarly, our current work requires access to hardware for the training phase, we hope that future work will expand these techniques to also support model-generation from inference-based, software-only techniques.

Conware works by first ingesting logs of real hardware interactions (*i.e.,* interrupts, reads, and writes) of the peripheral in question, which can be obtained by using either our novel source-code instrumentation technique, or existing hardware-in-the-loop techniques (*e.g.,* Avatar [27]). These recordings are converted into directed acyclic graphs (DAGs), where the edges are annotated with MMIO writes, and the interrupts and memory values are encoded as a peripheral *state* in the nodes. Then, the DAGs (one per peripheral) are converted into ω-automata using a novel graph-transformation technique, which serves as a generalized representation of the peripheral. Additionally, multiple recordings can be *merged* to create an automaton that accurately represents and generalizes *all* of the recorded interactions.

By representing peripherals as composable automata, Conware is able to not only combine recordings of the same peripheral, but it can also be used to merge disjoint peripherals to create a complete system that has multiple independent peripherals attached. For example, an Internet of Things (IoT) camera may consist of an ARM processor, a camera, a microphone, and a WiFi controller. To emulate this camera's firmware, one could purchase those same components and connect them to the appropriate ARM processor on a development board. Models for each of these peripherals could then be generated, independently, by running the example source code that is provided with the different peripheral components. These recordings could then be converted into automata, combined, and attached to an emulation environment that can be used to successfully emulate the firmware of the camera, which was never instrumented. To the best of our knowledge, Conware is the only system capable of creating generalized peripheral models that can be used on multiple firmware samples.

In summary, we claim the following contributions:

- an LLVM-based tool to automatically instrument source code to record all peripheral interactions (*i.e.,* MMIO and interrupts) on embedded systems,
- a novel technique for generating ω-automaton models of peripherals,
- a novel technique for *merging* peripheral models to facilitate portable models and full-system emulation,
- Conware: an open-source framework[1] for recording, modeling, and emulating embedded peripherals, and
- an analysis of Conware on popular embedded peripherals, demonstrating its efficacy by successfully modeling 10 peripherals and emulating one peripheral-heavy firmware that was never instrumented.

2 BACKGROUND

Generally, the goal of any analysis tool is to observe the system under analysis (SUA) in a realistic environment to extract features and draw conclusions about its inner workings. Thus, an effective embedded systems analysis framework must create a *realistic* representation of the hardware to ensure that the software will execute correctly and produce useful results. In the literature, this concept is known as *survivability*, which is "the ability for the firmware to

[1] https://github.com/ucsb-seclab/conware

execute the same regions of code as it would if the original hardware were present, without faulting, stalling, or otherwise impeding this process [17]" or "the firmware never ... *crashes, stalls, or skips operations* due to peripheral IO errors. [12]".

There are four general approaches to achieve survivability:

Hardware Debugging. Hardware debugging is capable of running the SUA on the actual hardware and debugging the real system. This can be done by leveraging existing debugging interfaces, *e.g.,* JTAG, or potentially interposing or snooping on the hardware components themselves. On production devices, it is increasingly rare for these interfaces to be exposed. Moreover, in many cases it is prohibitively expensive to acquire the SUA itself, instrument it, and potentially replace it if the analysis go awry — imagine irreversibly damaging an electric car.

Hardware-in-the-Loop Emulation. Debugging directly on the hardware can be slow, and lack certain features to aid analysis (*e.g.,* the ability to record every instruction or set a large number of breakpoints). Thus, hybrid, hardware-in-the-loop techniques [22, 37] have emerged to address this issue by emulating the main central processing unit (CPU) and intelligently forwarding any interactions with hardware peripherals to the actual hardware. While this approach offers a more feature-rich analysis, and a potentially much faster analysis environment, it does not scale, as the number of analyses executing is still limited by the physical hardware devices that are available. This lack of scale is due to the fact that a *real*, production hardware system is needed for each emulation platform, versus emulation which scales as a factor of computation resources. Similarly, there are numerous hardware constraints that are much more difficult with hardware-in-the-loop approaches (*e.g.,* frequent peripheral-initiated interrupts [15]).

Record-and-Replay. This approach involves removing the hardware dependency by obtaining a high-fidelity recording of the interactions, using the actual hardware, and then using this recording to effectively "replay" the interactions with the hardware. This technique is especially useful when the portion of code being analyzed does not have many hardware dependencies or the interaction in question is being replaced by the analysis itself (*e.g.,* fuzzing [34]). This technique has been implemented by systems like PANDA [11], which record the interactions in real time and then later use this recording to feed "real" data into a more heavyweight analysis. However, record-and-replay techniques do not facilitate any analyses that aim to exercise hardware interactions that were not directly observed in the recording phase.

Full-System Emulation. Full-system emulation executes the firmware in a completely emulated environment, ensuring survivability without the need for any hardware. Ideally, a full-system emulator would also enable the firmware to exercise *all* of its functionality, and interact with *all* of the assumed hardware peripherals. This can either be achieved by implementing the hardware peripherals either manually (*e.g., P^2IM* [12], HALucinator [9], or Simics [26]) or automatically (*e.g.,* Pretender [17]). Despite the various advances in this area, scalable full-system emulation of embedded systems is still an open problem. Pretender is a promising first step, but it requires debugging interfaces to be enabled on the system under test and is only capable of replaying the basic interactions of

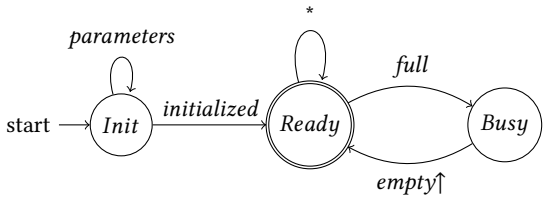


Figure 1: A state-machine representation of a simple universal asynchronous receiver-transmitter (UART) controller, where the peripheral is either awaiting to be initialized, ready to received any (∗) data, or in a busy state, which is transitioned to when the buffer is full and transitioned out of when the buffer has space (potentially triggering an interrupt)

the same hardware and firmware that were recorded in a mostly linear fashion. More precisely, Pretender is incapable of replaying peripherals that were not observed in the same recording nor is it capable of indefinitely emulating stateful peripherals. Conware is capable of creating portable and composable models that can be executed indefinitely, and is a strictly more general technique than Pretender.

Indeed, full-system emulation particularly useful for security analyses since it can provide the ability to analyze the entire firmware with limited false positives, since the identified problems will be in context of the complete system, and can even be used to generate proofs-of-concept that can be directly tested on the real system. Nevertheless, this valuable tool is still currently lacking in the world of embedded system security analyses. Conware has advanced the state-of-the-art in full-system emulation by facilitating *portable* and *composable* peripheral models, which can be leveraged by any emulation framework to enable unbounded execution of arbitrary firmware, even if the specific firmware and hardware being emulated have never been instrumented. Moreover, Conware does not require access to hardware used by the system under test at all, as a similar board could be used to generate the recordings and models, which can then be used to execute the firmware in question.

2.1 Motivation

As a motivating example, we examine the humble UART controller, which provides a simple interface to either read or write a single byte at a time (*e.g.,* a text-based interface).

UART, as is the case with many peripherals, has the ability to operate with or without interrupts. If an analyst uses a tool that does not use interrupts to generate a model of the peripheral, it is unlikely that this model would be useful for emulating a different system that *does* use interrupts. Indeed, the interrupt-based firmware would fail to execute, as it would wait indefinitely for an interrupt to be fired. Similarly, a naïve replay of MMIO values that were recorded with interrupts would not work on a non-interrupt-based firmware because the observed writes (*i.e.,* enabling interrupts for cached values) would differ from the observed values (*i.e.,* busy-waiting for a *ready* bit), requiring the emulation framework to "guess" what to do next. Thus, to ensure the complete functionality of this peripheral, a valid model must account for *at least* two

different scenarios, which are unlikely to ever occur in a single firmware. Even worse, UART is capable of reading and writing *any* byte. If the recorded UART interaction was non-variable (*e.g.,* it always output a fixed string) and the firmware being analyzed had more varied interactions (*i.e.,* exercising more of the peripheral's functionality), simple record-and-reply-based systems would fail to adequately handle these interactions.

Table 1: An recording obtained from instrumenting a simple firmware that prints "ON\r\n" and "off\r\n" repeatedly over UART, without interrupts

Operation	Address	Value
⋮	⋮	⋮
WRITE	0x400E0800 (control)	0x50
READ	0x400E0814 (status)	0x40001A1A (ready)
WRITE	0x400E081C (TX)	0x4F | **O**
READ	0x400E0814 (status)	0x40001818 (busy)
READ	*(repeats 434×)*	*(repeats 434×)*
READ	0x400E0814 (status)	0x4000181A (ready)
WRITE	0x400E081C (TX)	0x4E | **N**
READ	0x400E0814 (status)	0x40001818 (busy)
READ	*(repeats 2,634×)*	*(repeats 2,634×)*
READ	0x400E0814 (status)	0x4000181A (ready)
⋮	⋮	⋮

Despite the difficulties with replaying a UART recording, the actual state-machine of a typical UART peripheral is quite simple (see Figure 1). In fact, almost all embedded systems are implemented as state machines [32]. A high-level UART state machine consists of three states: an initialization state, where the firmware can set parameters like the bits of actual usable data (BAUD) rate; a ready state, where the controller is ready to receive and transmit data; and a busy state for when the controller is currently transmitting or receiving. These states are conveyed to the firmware through MMIO status registers. Thus, the firmware must either continuously check the status register before it can write new data (see Table 1) or ask the controller to trigger an interrupt when the state transitions from *busy* to *ready* (*i.e., empty*↑).

Conware is the first system capable of extracting models of the high-level state-machines that define the peripheral, using only recorded interactions.

3 SYSTEM DESIGN

Conware has three core components (see Figure 2):

- a source-code instrumentation framework capable of recording MMIO interactions and sending the log over any hardware interface (Section 3.1). This module is not necessary for Conware to work, as it can use recordings produced by previous work [17, 27];
- a model generation and optimization framework that converts a raw recording log into a DAG, and then into a ω-automata (Section 3.4); and
- an emulation framework that is capable of using the generated models (Section 3.7).

Indeed, these three components are standalone contributions, as each provides a unique contribution to the field.

Recording. Conware requires a recording of the low-level interactions (*i.e.,* MMIO and interrupts) with the target peripheral, which can be obtained through various methods. Indeed, hardware-based recording has already been explored, and the output of the existing tools can be used by Conware. These recordings can be of *any* firmware interacting with the peripheral; Conware does not require a recording from firmware being emulated. However, hardware-based methods require access to the original hardware *and* a debugging or instrumentation interface and are more likely to result in the Heisenberg effect (*i.e.,* the act of observing the phenomena alters its outcome) because of the timing overhead imposed. Thus, we created a new source-code instrumentation method for obtaining accurate hardware recordings to supplement this work.

Modeling. Conware generates models by first mapping the recordings of observed interactions (*raw recordings*) into directed graphs (one per peripheral), which are then converted into ω-automata. These automata are human-readable and facilitate unbounded execution since they more accurately represent the internal state machine of the real peripheral. Unfortunately, existing state machine minimization techniques (*e.g.,* the Hopcroft minimization algorithm [19], implication tables, or the Moore reduction procedure) are ill-suited for our purposes. This necessitated the creation of a novel automata-generation algorithm (Section 3.5).

Emulation. The ultimate goal is to emulate the firmware for the SUA, which can typically be obtained from the vendor's website or through more invasive techniques (*e.g.,* using a Bus Pirate [36] or advanced hardware hacking techniques [16]). Our automata, which can be generated from any example source code on any similar hardware (*e.g.,* a development board), can be plugged into any popular emulation framework as a stand in for the physical peripheral. More precisely, Conware is able to emulate arbitrary firmware without ever instrumenting the actual firmware or the hardware that it was intended for.

3.1 Source Code Instrumentation

Conware uses the LLVM framework [24] to instrument the firmware's source code. Consequently, our instrumentation works at the LLVM Bitcode level [23] and works on all of the source languages supported by LLVM (currently over 20). The purpose of this instrumentation is to record *all* of the interactions with MMIO peripherals (*i.e.,* reads, writes, and interrupts). This is implemented as a buffered logger that exposes two functions:

- `conware_log(address, value, type)` is used to log the addresses and values that were read from or written to (*i.e.,* `type`).
- `conware_interrupt_log(num)` is used to log the firing of an interrupt of a specific interrupt request line (IRQ) number (*i.e.,* `num`).

This logging infrastructure maintains an in-memory buffer and also takes care of flushing the buffer to a known interface (*e.g.,* the Joint Test Action Group (JTAG) or UART interface) when it is full or a programmed trigger is hit. This technique even works with high-bandwidth peripherals (*e.g.,* Ethernet) since we disable interrupts while flushing the buffer, which effectively halts the execution of the embedded system, permitting to operate a high-speed in small,

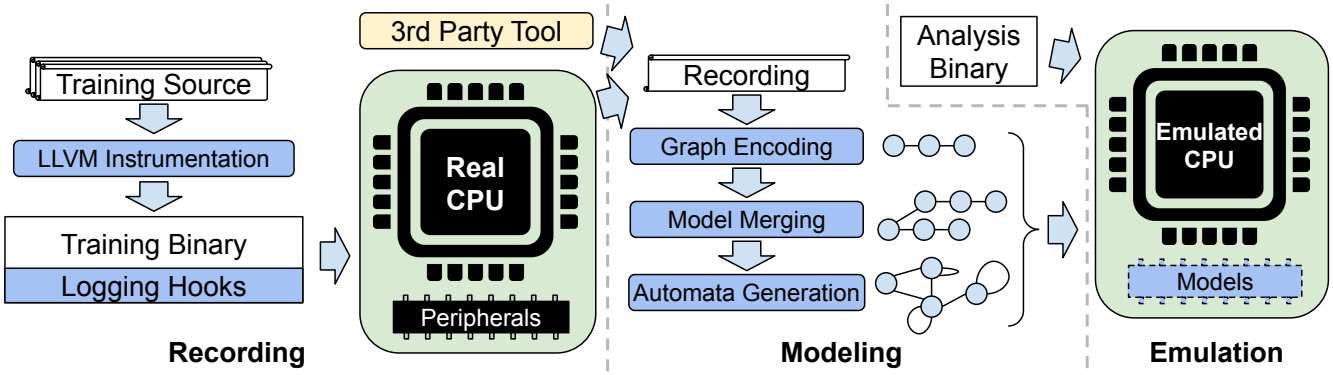


Figure 2: High-level design of ConWare: logging hooks are inserted into the compiled binary, the binary is run on real hardware to extract a detailed log, these logs are then converted into concise state-machine representations, and then both the uninstrumented binary and the models are run on emulated hardware, enabling detailed, scalable analyses.

buffered spurts. By logging directly to memory, ConWare incurs a minimal performance overhead, and thus a minimal Heisenberg effect. In fact, inline binary recording enables ConWare to overcome the challenge of recording frequent interrupts accurately [17]. Furthermore, our logger provides a single place to handle multithreading and reentrancy [13], which is necessary for accurately recording interrupts in practice. In addition to recording the immediately relevant information, we also log the program counter to facilitate debugging and future analysis.

3.2 Recording MMIO accesses

Though, theoretically, it is important to instrument all of the `loads` and `stores` to not miss any MMIO access, previous work [31] has shown that MMIO is usually accessed via hardcoded addresses, which can be retrieved by analyzing the firmware code of an embedded system. Exploiting this insight provides a lower overhead and is more tractable. Thus, all of the accesses that use hardcoded addresses are instrumented by inserting a call to `conware_log` after `loads` and before `stores`, with the value being read, or that is about to be written, respectively. In fact, in our specific examples, we found that this could be optimized even further, as all of the peripheral interactions were represented as `structs` in the source code (greatly reducing the overhead costs).

Due to memory limitations, ConWare stores all of the reads in a compressed array format for each entry, where repeated reads are stored only once, with an associated counter. This counter and log are reset every time a new write is observed, as writes are akin to state transitions of the peripheral. This is a necessary optimization since many MMIO values are read repeatedly until they change (*e.g.,* a status register) and would quickly exhaust the buffer otherwise.

3.3 Recording Interrupts

To record interrupts, we first retrieve all the interrupt service routines (ISRs) along with corresponding interrupt number they service from the interrupt vector table, which is always linked at a static address. ConWare instruments all the identified ISRs by inserting a call to the `conware_log` function at the entry of the function with the corresponding interrupt number. These interrupts are similarly compressed with a repeat counter to save buffer space and optimize our recording. The correlation between interrupts and their associated peripheral are discerned from the data sheet of the microcontroller, which are manually entered once per chipset. For example, page 38 of the datasheet for the SAM3X chipset explicitly lists every peripheral interrupt in the nested vectored interrupt controller (NVIC) [3], and the handlers to these routines are trivially found in the source code or compiled binaries. These memory locations will be constant across all variants of the same processor (*i.e.,* all Cortex-M3 processors will have the same values [25]).

3.4 Encoding Recordings

Hardware peripherals typically only change states when the software *writes* a value to one of the registers on the peripheral [17] (*e.g.,* the firmware writes a *command* to the peripheral). Thus, ConWare first encodes the recordings as simple DAGs where the edges are labeled with MMIO writes and the nodes encode the "state" of the peripheral, which includes the values that each memory region should return when they are read, as well as the interrupts that should be fired (and how many times). We call this graph a *linear model*, which is later converted into a more robust automata.

Formally, the DAG is denoted as (N, E) where E is the set of directed edges and N is the set of nodes. An edge $e_{12} \in E$ from n_1 to n_2 is represented by the tuple (n_1, n_2). Each node n has an associated state, such that $n.state \in S$, where S is the set of all states in a given DAG. This simple linear DAG can only reproduce a verbatim replay of the recorded content, as any out-of-order operations would not have a valid state transition and could only be handled by an educated guess.

As an explicit example, a write to the UART transmit (TX) register would traverse the edge with that specific value, and put the peripheral into a "new" state. This state would then return the following *pattern*: `BUSY` for the first 434 reads, and then `READY` on the 435^{th}. An explicit example of a node in our model for the UART peripheral is shown in Figure 3, where each address has its own

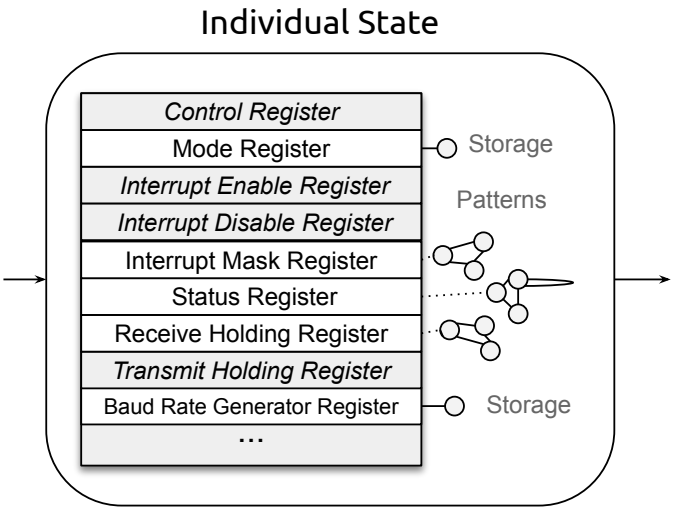


Figure 3: A individual node representation in our graph (each node contains a single peripheral state) of a UART controller, where edges are writes and the state encodes the values to be read from specific addresses. Each address has a sub-model: for example, Storage works as normal memory and Patterns return more complex data.

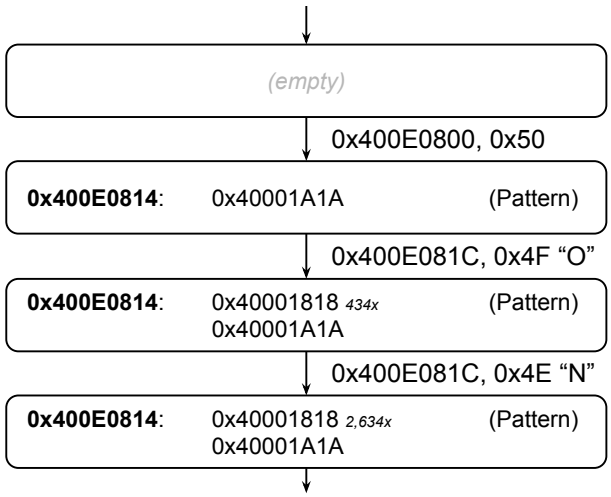


Figure 4: A DAG representation of a simple UART device, where each node represents a state and encodes the address to be read from (*e.g.,* status register) and the values to be returned (*i.e.,* `TXRDY` or `BUSY`). Nodes are traversed when writes are observed (*i.e.,* writing "O" to the TX register).

sub-model, within the overarching peripheral graph. And an example output of converting Table 1 into our DAG representation can be seen in Figure 4.

Within in each state, each memory address is encoded as sub-model to ensure that the appropriate values are returned when that address is read. These memory models are lumped into three general types [17]:

- storage – acts like normal memory,
- pattern – a single or repeated pattern, and
- monotonic – returns a monotonically changing value

For example, if a particular address in the peripheral (*e.g.,* a status register), always returns the same value, we will simply model that as a static pattern, which will always return the same value, regardless of how many reads occur. Conversely, if the register always returns a string of `0xA`s, followed by a `0xB`, our model will keep these semantics. This is currently the state of the art [17].

3.4.1 Interrupts. Conware must not only support interrupts, but must be able to automatically learn *when* to trigger which interrupt. Fortunately, the NVIC is standardized for most architectures (*i.e.,* every NVIC for ARM has the same structure), which permits us to manually hardcode the appropriate actions for various timers etc. by reading the datasheet. For example, our Cortex-M3 has eight timer counters (*i.e.,* Timer Counter Channels 1 through 8), that can be enabled or disabled. Thus, when a specific timer interrupt is enabled, we can programmatically trigger that interrupt periodically until the interrupt is later disabled. Somewhat unintuitively, the actually frequency of triggering the interrupt does not actually matter. For example, if the interrupt is supposed to trigger every 50 *ms* on a real board, deviating from this is unlikely to result in an erroneous emulation. The reason for this is that embedded systems rely on these interrupts for their time, and have no other timekeeping mechanisms. Thus, the code treats the interrupts as a single time unit, but does not make any assumptions as to the actual time that has passed. This assumption is only true of timer-like interrupts.

For peripheral-triggered interrupts (*e.g.,* a *data ready* interrupt), this problem is exacerbated by the fact that interrupts can depend on the context of the peripheral and the firmware. To ensure that interrupts are triggered at the correct time, they are encoded as part of a node's state (*i.e.,* the specific IRQ numbers and how many of each), and triggered when the incoming edge is taken. Thus, interrupts will only be triggered after a write was observed that was also immediately followed by interrupts in the recordings (*i.e.,* that specific state in our model was reached, not just the address/value pair). This is contrary to previous work [17], which would observe a specific write and then begin triggering interrupts indefinitely, which is likely to lead to over-approximation in practice. When observing our UART controller, the interrupt was never explicitly disabled, but one interrupt per write was issued (*i.e.,* each write queues a future interrupt).

An explicit example of this can be seen in Figure 5, which was generated by recording and optimizing a firmware that prints "Knock!\r\n" every time that a piezo transducer is *knocked*. Our model correctly encodes the interrupts into the state where the final byte was written, and all subsequent byte writes to the `TX` register are encoded as a self-loop. In practice this corresponds to our emulator returning *busy* the appropriate number of times, and triggering a *ready* interrupt every time a valid write to the TX register is observed. Indeed, this automata can be used to emulate any firmware that supports interrupts. If the execution deviates from our model (*i.e.,* the execution wrote a value that is not present in our model), we perform a breadth-first search (BFS) to identify an

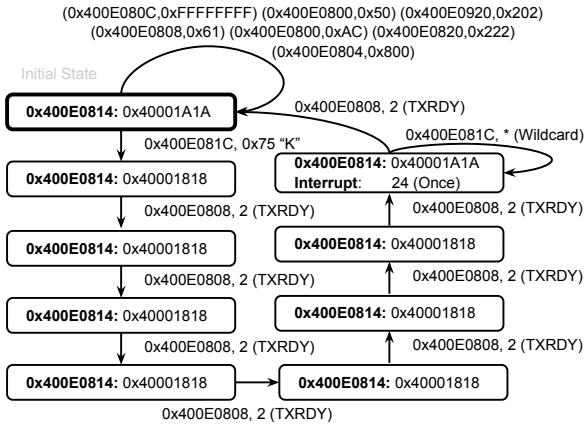


Figure 5: An ω-automata for a UART peripheral generated from a recording that prints "Knock!\r\n" repeatedly. The initial state will accept all configuration parameters, once a "K" is written, the peripheral becomes `BUSY` until the `READY` state is reached and interrupts are thrown.

acceptable state within our model (*e.g.,* any write to the TX register other than "K" would ultimately take the wildcard edge where the buffer is `READY`). This, as shown in Section 4.3, will result in the proper actions ultimately being taken (*e.g.,* when enough buffered writes are observed, it will transition into the *ready* state, and the interrupts will fire until the buffer is emptied).

3.5 Automata Generation

The next step of our approach involves transforming our DAGs, obtained as explained in Section 3.4, into ω-automata. Before generating our ω-automata we must first define what it means for two states to be *mergeable* (*i.e.,* they can be merged into one), or, put another way, which nodes in the graph are in the same equivalence class. Two generic states a and b can be merged, indicated by $a \cup b$, if they are *equivalent*. The states a and b are equivalent, indicated as $a \simeq b$, if they have the same identified type (*e.g.,* storage, constant, or pattern) for every overlapping memory address, and those types are also equivalent (*i.e.,* they encode the same data). Nodes are mergeable if and only if their states are mergeable. Additionally, we consider two edges to be equivalent, also denoted by $\simeq$, if they have the same labels (*i.e.,* the same write address and value).

For example, take states $a, b \in S$ where a has memory models for address $0x100$ (Storage) and $0x200$ (Pattern) and b has memory models for $0x100$ (Storage) and $0x300$ (Pattern). These states would be considered to be mergeable, since there is no risk of returning a wrong value. The returned values would be the same for both a or b and $a \cup b$. Indeed, $a \cup b$ is strictly more verbose than either of the individual states. Patterns are only considered equal if they are identical. While this could potentially be relaxed, there are numerous cases where the exact values are critical, *e.g.,* the NEC infrared (IR) encoding protocol.

While this definition of *mergeability* is intuitive, its lack of transitivity does slightly complicate the state-reduction phase. More precisely, it is possible for $A \simeq B$ and $B \simeq C$ but $A \cup B \not\simeq C$ (*e.g.,* A:[$0x100$:Storage], B:[$0x200$:Pattern], C:[$0x100$:Pattern]). Thus, we must be diligent when merging equivalence classes to make sure that all of the merges will succeed without violating the soundness of our model.

The goal of the automata-generation phase is to combine all of the mergeable states to create a more general representation that can be used for indefinite execution and handle out-of-order operations. For example, a linear DAG, which is the current state of the art [17], is incapable of handling execution beyond the last node in the graph, and would completely fail if firmware were to execute functionality in a different order. A generalized automata, in theory, should not suffer any of these shortcomings.

Our automata-generation algorithm starts at the initial node (*i.e.,* the node that corresponds to the *first* action in the recording) in the graph (i), and traverses the graph using a nested depth-first search (DFS) such that i is compared to every node that is reachable from i. This search is then repeated for every subsequent node in the graph, until a set of nodes that can be merged (*i.e.,* in the same equivalence class, C) is successfully identified. If two nodes are mergeable, the algorithm then traverses all of the equivalent outgoing edges (*i.e.,* the edges have the same labeled memory write address and value) recursively to ensure that after the two nodes are merged all of the edges will remain valid. More precisely, for two nodes to be merged, they must be mergeable, and all of the nodes on the same outgoing edges must also be equivalent. This recursive comparison is done by first identifying equivalent edges for the two initial nodes, and then recursively identifying all of the equivalent edges for any other identified nodes, until a cycle is completed or the two nodes in question share no common outgoing edges. If two unequal nodes are found, the nodes are marked as *unmergeable*. Finally, because our equivalence comparison is non-transitive, we confirm the equivalence of the cross product of the various node equivalence classes before merging the equivalence class into a single node and combining the relevant edges.

After a successful merge, the algorithm is then run again, starting at the initial node (i). This process is repeated until the algorithm reaches a fixed point, which is defined by reaching the end of the nested DFS for every node in the graph without any nodes being merged. The entire algorithm is shown more formally in Algorithm 1. This algorithm guarantees that we have obtained *an* ω-automaton, but not necessarily the *best* or *smallest* representation, as the order of operations could impact the outcome. Nevertheless, this automaton is more than sufficient for the purpose of emulating and understanding the general structure of the peripheral's internals. The results of this algorithm on a UART recording which prints "Knock!\r\n" indefinitely is show in Figure 5.

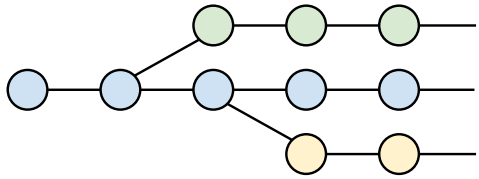
To achieve more general automata, we consider an edge to be a wildcard (*i.e.,* any value is an acceptable state transition for that address), and merge the associated edges, if and only if all of the outgoing edges with that address have the same destination node and the number of similar edges is above a threshold (*e.g.,* five). Indeed, this merging of edges greatly increases Conware's ability emulate unobserved code branches in our training data, as well as never-before-seen firmware (see Figure 5).

```
Function GetEdges(n_1, n_2):
    , _1 ← n_1.state;
    , _2 ← n_2.state;
    if , _1 ≃ , _2 then
        if n_1 ∈ C ∧ n_2 ∈ C then
            return;
        end
        C ← C ∪ {, _1, , _2};
        EC ← ConnectedComponents(C, n_1);
        O ← OutgoingEdges(EC, n_1);
        R ← ∅;
        forall e_1 ∈ O do
            forall e_2 ∈ n_1.edges ∪ n_2.edges do
                if e_1 ≃ e_2 then
                    R ← R ∪ (e_1.dest, e_2.dest);
                end
            end
        end
        return R;
    else
        return ⊥;
    end
Function GenerateAutomata(N):
    forall n_1, n_2 ∈ N | n_1 ≠ n_2 do
        C ← ∅;
        M ← GetEdges(n_1, n_2);
        while x, y ← pop(M) do
            M ← M ∪ GetEdges(x, y);
        end
        if M ≠⊥ then
            Merge(C);
        end
    end
    return;
```

Algorithm 1: Functions for determining if two nodes are equal and can be merged, which will ultimately update the graph by merging all "equal" nodes, and all of their annotations, into a single node.

3.6 Combining Models

Because of the way in which our automata generation is implemented, merging recordings is relatively straightforward. First, we combine the initial linear DAGs by starting at the initial nodes and merging every mergeable state until there is a convergence (*i.e.,* the current node is equivalent but the outgoing edges are not equivalent). Given that peripherals are expected to power on into a known state, it is unlikely to ever have a read value differ without first seeing a deviation in the written value (*i.e.,* putting the peripheral in a different state). The resulting merged graph will then have only a few nodes with multiple outgoing edges and no cycles (*i.e.,* a tree).



Regardless of the number of recordings that are merged, the automata-generation step proceeds as it did in the single-recording scenario – iteratively merging equivalent states until no more nodes can be merged. The result of this step is a model that is generalized *and* satisfies the constraints of every input model. Said another way, this model can be used to successfully emulate *any* of the original firmware, and likely many other firmware that use the same peripherals.

3.7 Hardware Emulation

Our current emulation framework is built in Python as an extension of Avatar[2], and is loosely based on Pretender. Conware implements a custom `AvatarPeripheral` that encompasses the entire MMIO memory region, where the reads and writes interact directly with the generated models, advancing states on writes and returning appropriate values on reads. Within this class, the memory is split into individual peripherals, which can either be identified manually (*e.g.,* by reading the data sheet) or automatically [17]. Each individual peripheral has its own disjoint automaton that is actuated in isolation. Once the emulator is running, the `write` command will result in the model advancing, either to the next state if that forward edge exists, or by performing a BFS in the case that is it does not. If the BFS fails, the first fallback is to pick the node in the entire graph that has the most observed incoming edges with that address value. If a write to this address was never observed, we simply stay in the same state and create a `Storage` model for that address.

If a state is entered that has interrupts, a thread is started for each interrupt that will trigger it the appropriate number of times. This is done to ensure that the firmware still executes seamlessly, without waiting until the interrupts are handled. Similarly, this permits Conware to trigger continuous interrupts (*e.g.,* a counter that is triggered every *Xms*). As previously mentioned, known interrupts are hard-coded in our emulation framework and triggered when the appropriate *enable* bit is written to a specific address, while peripheral-specific interrupts are triggered during state transitions.

3.8 Porting Models

Conware models are also portable across different chipsets and memory maps. For example, if the training device had the peripheral mapped at memory addresses `X - Y` and the emulated peripheral mapped the device at `X' - Y'`, the Conware model can be updated appropriately. In the Conware framework each peripheral is stored as a self-contained object, which is mapped to the specific memory region. More precisely, each memory address has a direct mapping to the corresponding peripheral object. Moreover, within the peripheral object, each memory address is also mapped to a specific model for memory reads to ensure that the proper value is always returned. Thus, porting a model to a different region of memory is

as easy and remapping each of these objects in the nested dictionary object.

In fact, these memory models are readily available [29] for many embedded processes as System View Description (SVD) files. These files explicitly list the MMIO peripherals, and even which registers are mapped to which address. Therefore, a simple transformation tool that changes the address locations between device models could be trivially constructed to move a peripheral model between chipsets.

4 EVALUATION

The Arduino platform proved to be a perfect testing ground for ConWare — it is open-source, is compatible with a large array of peripherals, and has well-documented example code for the supported peripherals. Indeed, analysis aside, ConWare provides the ability to fully emulate Arduino firmware *with* arbitrary peripherals, making it the first system capable of this feat. To ensure the applicability of our evaluation to real-world systems, we opted to use the Arduino Due, which has a 32 bit ARM Cortex-M3 processor (the Atmel SMART SAM3X/A [3]). To instrument the Arduino code, we modified the build environment to instrument both the Arduino environment and the program that was being compiled on top of that environment (*i.e.,* the `.ino` file). This instrumentation is capable of automatically injecting logging into any Adruino program, including library packages, and outputting the recorded log over any standard interface (*e.g.,* UART) after the specified buffer has been exhausted (*e.g.,* 2,000 entries with compression) or a triggered event was detected (*e.g.,* a button press). This buffer can be filled, emptied, and recorded indefinitely, which enables the recording of long-running or MMIO-intensive interactions.

For our evaluation we ran a spread of unique experiments to demonstrate the practicality our modeling framework:

- Recording and replaying the same firmware, with a ConWare model replacing the hardware (Section 4.2);
- Recording and replaying the same firmware with a *merged* and generalized automaton — multiple recordings where merged into one model and the original firmware was run against it (Section 4.3);
- Recording and replaying an "unseen" firmware with a merged and generalized automaton — the model was generated using different recordings of individual peripherals using the example source code that was provided with those peripherals (Section 4.4).

The purpose of these experiments was to demonstrate that ConWare is a viable solution for emulating hardware peripherals, and that it is capable of handling real-world peripherals. All of these experiments were run in a fully-automated fashion (*i.e.,* a single script was executed to generate all of the models and execute the emulator). We do not claim that these findings indicate that ConWare is *the* solution or that emulating embedded systems is *solved*, but instead advocate graph-based, automata modeling of peripherals as a viable technique for the research community to continue to address this critically important problem (*i.e.,* the full-system emulation of embedded systems).

Before delving into the results, we want to first emphasize the complexity involved with emulating an embedded system. We first instrumented an Arduino program that simply blinks the on-board light emitting diode (LED) and prints text over UART (see Listing 1). With a 2,000 entry recording buffer (with compression), our instrumentation logged 88,287 MMIO accesses, which consisted of 65 unique addresses across 11 peripherals. The breakdown of these accesses were as follows: 52 unique addresses were written to (159 unique address-value combinations), 21 unique address were read from (37 unique address-value combinations), and zero interrupts (excluding `SysTick`) were observed. To represent these logs in our linear DAG it requires 11 separate graphs (one for each peripheral), which in total contain 1,014 nodes, 1,003 edges, and encode 87,284 values in their states (*i.e.,* values to return when certain memory addresses are read). The UART peripheral accounts for 367 of those nodes and the platform input and output B (PIOB) peripheral, which is used to control the LED, accounted for 506 nodes. Emulating even a simple firmware such as this one is no trivial task. After applying our automata-generation technique, the graphs contained a combined 26 nodes (a 22× reduction) and 45 edges, 21 of which are self loops. This reduction can be made arbitrarily high by recording for longer, as the number of equivalence classes is static.

```
void loop() {
  printf("ON\n\r");
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  printf("off\n\r");
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}
```

Listing 1: Simple Arduino program that blinks an LED and prints the lumination status over UART

4.1 Dataset and Experimental Setup

To demonstrate the breadth of devices, and interactions, we strategically choose a few indicative peripherals, many of which are used by a hobbyist smart door lock firmware that we emulate in Section 4.4:

- `IR`: an IR remote controller [14] and an IR receiver,
- `LCD`: a standard HD44780 liquid crystal display (LCD) display,
- `Knock`: a piezo transducer to detect a "knock,"
- `UART`: various UART interactions, both with and without interrupts,
- `Color`: a Cadmium-Sulfide (CdS) photo resistor used to detect the color of incoming light,
- `Servo` [4]: a 180 degree servo motor,
- `LED`: both onboard and external LEDs,
- `Ethernet`: an Ethernet board capable of 100 Mbit communication,
- `Button`: an external button, and
- `RF TX`: a 433 MHz radio frequency (RF) receiver.

The source code for all of the experiments in this section is available in our GitHub repository hosted at https://github.com/ucsb-seclab/conware.

The IR remote is particularly interesting, as it works by starting a timer (`TC5`), which will fire interrupts indefinitely. The interrupt controller reads the value from the IR receiver (*e.g.,* high or low), and

will continue to receive data, according to the NEC IR transmission protocol [2], until an entire data unit was received. At this point, the individual bits in the buffer are decoded into their respective byte values (*e.g.,* the number "1" is encoded as `0xFF30CF`), which can then be parsed and handled by the application. This means that in order to properly emulate this peripheral, the interrupts must be triggered appropriately, the individual bits must be fed in correctly, and in order, and the subsequent actions must also be supported (*e.g.,* flashing an LED, or printing values over UART, which also uses interrupts).

The `LCD` code is high-bandwidth, and indicative of more complicated peripherals that display detailed information to users (*e.g.,* an alarm clock, weather app, or smart electronic). The `Knock` sensor is representative of any analog sensor (*e.g.,* temperature, acceleration, or humidity) that has a range of values, of which the firmware is typically concerned with some "threshold" value. `UART` is still one of the most popular protocols for interacting with embedded systems, and presents an interesting case because, while its actual functionality is simple, the implemented functionality is unbounded (*e.g.,* complete shell interfaces). `Color` is an analog sensor that also depends on actuating nearby LEDs (*e.g.,* red, green, and blue) to detect the reflected light. Buttons, servos, and LEDs are common interfaces for most embedded systems that need to communicate with the user efficiently or actuate some external motor. And, finally, to ensure that our peripherals were indicative of popular IoT devices, which communicate with external devices, we also included an Ethernet controller and a popular 433 MHz wireless radio. Both of these communication peripherals were running echo servers, and were appropriately actuated in training.

All of these sensors came with accompanying libraries and example code, *i.e.,* `File|Examples` in the Arduino integrated development environment (IDE), that was used in our experiments to remove any biases. These examples programs were used "as is" to generate our recordings. Emulations were run on a laptop with an Intel® Core™ i7-8550U CPU @ 1.80 GHz and 16GB of memory.

Real-world Relevance. In addition to the chosen peripherals being popular in IoT devices, the Arduino platform is also used by many rapid prototyping companies [7]. Thus Arduino-based peripherals and their interactions in Arduino products should be indicative of real-world applications, as the major difference from prototype to production is typically cost reduction by choosing smaller, less expensive parts and creating a custom printed circuit board (PCB) that only includes the necessary components [6]. In our smart door lock firmware, the peripheral interactions are also non-trivial. The interrupts for the IR sensor are constantly firing to accept user input. If the "knock" command is received, it then enters a loop that reads the piezo transducer for a fixed amount of time (using a hardware timer), and then will actuate the servo appropriately based on the correctness of the knock pattern. Similarly, if the "color" command is received, the firmware will enter a function that illuminates three LEDs in sequence (red, green, and blue), while reading the value of the photo resistor to determine the color of the object that is near the sensor. If the correct color is detected, the servo is actuated accordingly. These peripheral dependencies, which are typical of IoT devices necessitate a high-fidelity emulation framework — a simple replay of these peripherals would not suffice in exercising any of the interesting functionality of this firmware (*i.e.,* unlocking the door).

4.2 Record and Replay

First, we wanted to demonstrate that Conware is able to achieve the basic record-and-replay functionality on which existing system have focused. In these experiments, we took the example code for our test cases and compiled both an instrumented (*i.e.,* logging enabled) and uninstrumented version of the firmware for each. The instrumented version was then executed on the real hardware, in combination with manual interaction with the peripheral (*e.g.,* pressing buttons on the IR remote or knocking the piezo transducer) until the record buffer was full and the recording was dumped over UART. The recording was then converted into a linear model, and then an automaton. Replaying the linear model is effectively equivalent to the current state of the art (*i.e.,* Pretender [17]).

For each of these direct record-and-replay cases, both the linear graphs and generalized graphs were able to replay the originally recorded firmware. However, after the logs were exhausted (*i.e.,* the emulation ran for more time than the original recording), the differences were clear. In fact, without a technique like Conware, there is currently no proposal (aside from guessing) for how to handle future execution. Nevertheless, Conware's state-machine-like models were able to successfully execute indefinitely. We used the generated models to run each of the samples for 10 minutes in our emulation framework. To enable a straightforward comparison, our emulation framework outputs logs in the same format as our recordings. Thus, we are able to compare the accesses to each peripheral, in order. Since our replays are deterministic and will return the same recording every time, there is no value in running the experiments more than once.

This comparison is done by first splitting the output of each log into its respective peripheral. For example, if 11 peripherals were observed, the log would be split into 11 separate logs where the entries for each peripheral are in sequential order. The logs for each peripheral (*i.e.,* the recorded log and the emulated log) are then compared directly using sequence matching, where duplicates are treated as a single value. More precisely, any repeats reads are effectively collated into a single entry, which ensures that the same sequence, but not necessarily the same exact observation. This ensures that we do not unnecessarily punish ourselves for things like status registers, which can return the same valuable a variable number of times without impacting the code execution, but still enforces strict order, which should only be the same if the states are advancing correctly.

The results of executing each example firmware against its own automaton is shown in Table 2. All but five of the firmware samples replayed *exactly* as they did in the recording. Four of them (*i.e.,* `IR`, `Color`, `RF RX`, and `Ethernet` had a few missing entries due to UART buffer inconsistencies) and LCD had some executions appear out of order, due to interrupts arriving in a different order. Indeed, the differences indicate, more than the identical comparisons, that our automaton is better than a simple replay. Somewhat more interesting than the order of the accesses is the total number of MMIO accesses that were observed. All but two (*i.e.,* `IR` and `Servo`) actuated *far more* MMIO accesses than were observed in the initial

Table 2: A comparison of the in-order MMIO access logs of both the recorded and emulated firmware

Firmware	Conflicts	Additional (%)	Missing (%)	Total (Emu.)	Total (Rec.)
Knock	0 (0.000)	0 (0.000)	0 (0.000)	34,028	5,607
UART	0 (0.000)	0 (0.000)	0 (0.000)	653,793	222,123
Servo	0 (0.000)	0 (0.000)	0 (0.000)	949	4,571
Blink2	0 (0.000)	0 (0.000)	0 (0.000)	15,393	2,606
Blink	0 (0.000)	0 (0.000)	0 (0.000)	212,594	88,286
IR	0 (0.000)	0 (0.000)	1 (0.002)	53,955	205,977
LCD	10 (0.221)	56 (1.237)	136 (3.005)	533,997	4,506
Ethernet	0 (0.000)	1 (0.022)	16 (0.356)	153,170	4,491
Button	0 (0.000)	0 (0.000)	0 (0.000)	614,354	4,603
Color	0 (0.000)	1 (0.039)	1 (0.039)	17,237	2,570
RF RX	1 (0.001)	2 (0.002)	3 (0.004)	82,478	124,807

recording, emphasizing the power of Conware. IR, RF RX, and Servo are very MMIO-heavy, which accounted for the lower number of observed accesses since the MMIO accesses incur a larger overhead in emulation. The same one-to-one correlation was observed beyond 10 minutes, and the values observed over UART were also identical.

As a sanity check, we attempted to execute the firmware for Blink, which does not use interrupts, using the model that was generated from Knock, which does, to see if the UART peripheral would work correctly. Unsurprisingly, this emulation failed to print any characters after the first one, since the UART status register would continually return BUSY, expecting that the firmware would buffer them, and then request interrupts. This experiment demonstrates the subtlety that must be accounted for when emulating embedded systems.

4.3 One Model to Emulate Them All

To demonstrate the efficacy of our technique and to quantify the compression that is achieved by our automata-generation phase, we examined the resulting graphs at each step in our process. Table 4 shows the number of states (*i.e.,* nodes), edges, and self-loops for each peripheral in the case of a linear graph (*i.e.,* the current state of the art) and our automata graphs (*i.e.,* after our automata-generation step), which are denoted with a $_G$ subscript. Looking at this table, it is clear to see that our state-reduction is highly effective, reducing the number of required states by more than 10 fold in every instance. Again, these reductions can be made arbitrarily large by inputting longer recordings. Moreover, this table again demonstrates the complexity of the "re-hosting problem" (*i.e.,* emulating embedded systems). While these examples are objectively simple, they still require the proper emulation of multiple peripherals to execute successfully, even if they do not explicitly use them.

The true value of our automata-generation is not to create models that can be used beyond the recorded execution, but to create portable models by merging the recordings from various firmware samples to capture the full gamut of peripheral interactions, and enabled the emulation of *any* firmware. As a first step toward this goal, we show that Conware can generate merged models that are at least able to emulate their original recordings. This would not be possible with simple linear models alone (*i.e.,* Pretender) — at least a tree would be required to encode the divergence point. Moreover, merging multiple recordings with a simple tree would result in very large models, and would be incapable of handling a firmware sample that actuates a mixture of the functionality observed in multiple training recordings.

Table 3: Summary of executing the various firmware samples on a merged model that is a composition of their individual recordings. The example firmware samples are emulated for 10 minutes, while the smart-lock firmware executed for 60 minutes. MMIO writes, reads and peripheral-specific interrupts are reported, as well as graph traversal statistics: long jumps (took a non-existent edge), wildcards (took a wildcard edge), and BFS (performed a BFS to find the appropriate next state).

Firmware	Writes	Reads	Interrupts	Long Jumps	Wildcards	BFS
Knock	14,112	15,562	3,288	6	427	13
Servo	185,067	111,791	0	3,681	16	191
Button	344,876	173,157	0	0	16	3,985
IR	228	90,179	24	0	19	0
Blink2	10,718	4,305	0	16	6,947	81
Color	1,601	1,744	226	0	36	22
Lock	795,597	1,598,900	397	136	17	26,895

Our Knock and IR examples both read sensor values and report the value over UART. However, these sensors are very different, and the UART output is completely divergent: more precisely, the text "Knock" versus a hexadecimal representation of the encode button press. This made these two samples an ideal ground for testing the portability of our models. When the models were merged, they were both able to emulate successfully, using *the same automaton.*

With the basic functionality confirmed, we merged the models for multiple non-overlapping peripherals (*i.e.,* they all use different physical pins) in an attempt to create a full-system emulation model, capable of handling *any* of the modeled peripherals. Specifically, we created a single model using the recordings from Color, IR, Knock, Blink2 (which blinks external LEDs), Servo, and Button. These peripherals were chosen because they are all used by the smart lock firmware that is our ultimate emulation target, and thus we will refer to this automaton as Lock.

Indeed, we were able to use the Lock model to successfully emulate *all* of the original firmware samples. Given the added complexity of these graphs, it is reasonable to assume that some state transitions may no longer be as straightforward. To investigate exactly "how" the model was emulating these firmware samples, we kept track of every MMIO interaction and the effect that it had on the graph traversal. Table 3 enumerates the various non-standard transitions (*i.e.,* state transitions that did not have an immediately available edge from the current state). We define long jumps as a state transition that had to temporarily create a new transition (*i.e.,* the destination node was not reachable from the current node). The edge selection process is prioritized by locality and the number of edges that were merged to create the selected edge (*i.e.,* how many times that specific state transition was observed). Wildcards are edges that our algorithm deemed safe to accept *any* value (*e.g.,* the TX buffer in a UART controller). Finally, BFS transitions occur when the existing transition is not valid but a BFS through the graph was able to locate an acceptable edge. Fallback transitions were uncommon when emulating any of the initial firmware samples, as their recordings were used to generate the automata.

Table 4: Summary of the complexity of both the linear and generalized graphs for our five indicative firmware samples, showing edges, E, self-loops, L, and nodes, N, for each peripheral. Models were generated from recordings with a 2,000 item buffer (with compression). The columns relate to the peripheral controller that the ARM processor interfaces with (*i.e.,* the actual peripheral is behind by these standard interfaces)

	UART			PIOA			PIOB			PIOC			PIOD			UOTGHS			TC1			EEFC0			ADC			PMC			WDT			EEFC1			Total		
Name	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N	E	L	N
IR	304	0	305	50	0	51	25	0	26	27	0	28	22	0	23	13	0	14	8	0	9	1	0	2	10	0	11	15	0	16	1	0	2	1	0	2	477	0	489
IR_G	128	10	103	7	3	3	4	2	2	4	2	2	4	2	2	4	2	2	3	2	2	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	171	30	128
Knock	963	0	964	50	0	51	391	0	392	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	714	0	715	12	0	13	1	0	2	1	0	2	2,190	0	2,201
$Knock_G$	11	2	9	7	3	3	126	62	62	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	705	352	352	9	3	6	1	1	1	1	1	1	873	431	441
UART	552	0	553	50	0	51	31	0	32	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	715	0	726
$UART_G$	44	22	23	7	3	3	6	3	3	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	85	41	47
Servo	7	0	8	50	0	51	25	0	26	1,842	0	1,843	17	0	18	13	0	14	459	0	460	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	2,439	0	2,451
$Servo_G$	1	1	1	7	3	3	4	2	2	274	137	137	4	2	2	4	2	2	203	4	69	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	514	158	228
Button	7	0	8	50	0	51	2,451	0	2,452	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	2,591	0	2,602
$Button_G$	1	1	1	7	3	3	813	406	406	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	850	423	428
Blink	366	0	367	50	0	51	505	0	506	27	0	28	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	1,003	0	1,014
$Blink_G$	4	2	2	7	3	3	6	3	3	4	2	2	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	45	21	26
RF RX	97	0	98	50	0	51	25	0	26	57	0	58	52	0	53	13	0	14	-	-	-	1	0	2	10	0	11	15	0	16	1	0	2	1	0	2	328	0	340
RF RX_G	32	2	27	7	3	3	4	2	2	12	6	6	15	7	7	4	2	2	-	-	-	1	1	1	4	1	3	11	4	6	1	1	1	1	1	1	95	32	61
LCD	7	0	8	50	0	51	25	0	26	2,029	0	2,030	759	0	760	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	2,908	0	2,919
LCD_G	1	1	1	7	3	3	4	2	2	135	39	39	103	50	50	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	270	104	109
Ethernet	74	0	75	65	0	66	25	0	26	33	0	34	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	13	0	14	1	0	2	1	0	2	1,189	0	1,201
$Ethernet_G$	32	2	26	7	3	3	4	2	2	4	2	3	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	10	3	6	1	1	1	1	1	1	524	21	395
Blink2	7	0	8	50	0	51	205	0	206	1,080	0	1,081	17	0	18	13	0	14	-	-	-	1	0	2	10	0	11	12	0	13	1	0	2	1	0	2	1,397	0	1,408
$Blink2_G$	1	1	1	7	3	3	64	31	31	72	36	36	4	2	2	4	2	2	-	-	-	1	1	1	4	1	3	9	3	6	1	1	1	1	1	1	168	82	87
Color	375	0	376	50	0	51	30	0	31	477	0	478	17	0	18	13	0	14	-	-	-	1	0	2	56	0	57	13	0	14	1	0	2	1	0	2	1,034	0	1,045
$Color_G$	11	2	9	7	3	3	4	2	2	55	18	19	4	2	2	4	2	2	-	-	-	1	1	1	47	23	23	10	3	6	1	1	1	1	1	1	145	58	69
K+ir	1,260	0	1,261	50	0	51	391	0	392	27	0	28	22	0	23	13	0	14	8	0	9	1	0	2	1	0	2	15	0	16	1	0	2	714	0	715	2,503	0	2,515
$K+ir_G$	122	11	105	7	3	3	126	62	62	58	29	29	4	2	2	4	2	2	3	2	2	1	1	1	1	1	1	10	3	6	1	1	1	704	352	352	1,041	469	566
lock	1,628	0	1,629	50	0	51	396	0	397	3,333	0	3,334	22	0	23	13	0	14	467	0	468	1	0	2	1	0	2	18	0	19	1	0	2	760	0	761	6,690	0	6,702
$lock_G$	151	11	111	7	3	3	126	62	62	723	354	354	4	2	2	4	2	2	329	38	132	1	1	1	1	1	1	12	4	6	1	1	1	751	375	375	2,110	854	1,050

PIO - Parallel Input/Outputs **UOTGHS** - USB OTG High Speed **TC** - Timer Counter **EEFC** - Enhanced Embedded Flash Controller
ADC - Analog-to-Digital Converter **PMC** - Power Management Controller **WDT** - Watchdog Timer

In Table 3, both `Blink2` and `Knock` observed multiple wildcard traversals due to their heavy usage of UART, which lends itself well to this. The 3,000+ long jumps in `Servo` are due to an interrupt handler accessing a memory address that was not available in the current state (*i.e.,* there was no sub-model for it). This is due to the fact that the emulated interrupts do not happen at the *exact* time that they were observed in the recording. Nevertheless, the long jump selects a satisfactory node every time, and the execution continues correctly. This same phenomena occurred in the UART controller for `Blink2` and `Knock`, since the UART automaton is capable of supporting *any* of the interactions that were previously observed. Likewise for the multiple BFS traversals that were required.

4.4 Emulating Arbitrary Firmware

Finally, we exhibit Conware's ability to emulate a complete new firmware that was never seen in the training data. The hobbyist smart door lock program that we selected permits users to unlock the door by a knock pattern, a personal identification number (PIN) entered on the IR remote, or by presenting a specific color. In our recordings with the initial peripherals, we input sequences that would be accepted by the smart door firmware. However, these inputs could be replaced by a fuzzer, for example, in a straightforward way [9, 17, 28]. This particular firmware would require 13 wires to be connected to the Arduino, and has 11 different physical peripherals, making it a non-trivial emulation target. Nevertheless, we were able to emulate the firmware using our `Lock` model (*i.e.,* the automaton that was created from the individual recordings of the various peripherals using Arduino's included example code). Surprisingly, we observed *zero* failed reads or writes (*i.e.,* there were no reads or writes that our model was not able to handle). Verifying that our models worked "correctly" is not straightforward, since our goal is survivability of execution as opposed to a *perfect* representation. Thus, we first used high-level metrics, such as UART output, which this firmware had, and the distribution of MMIO accesses. In fact, the UART output was identical as when we built and ran the real firmware.

To ensure that the peripherals were actually facilitating this interaction, and that our models were not just getting "lucky," we logged every MMIO accesses that the `Lock` firmware exercised, and compared them to the peripheral recordings. Indeed, after aggregating *all* of the MMIO accesses from the recordings of each peripheral on the real hardware (using example code) and comparing the accesses to the execution of the `Lock` firmware, we found that the

same interrupts were fired, and only five MMIO addresses (out of 94) were in the training data that were not observed in the recording. Indeed, these five addresses where all associated with the servo — specifically the PIOB controller and a PIO Pull Up Register. This makes sense, because the servo example code would increment the servo one degree at a time to move the motor slowly between *every* position, while the `Lock` only had an "on" and "off" position, which requires far less interaction. Moreover, we found that 348 of the address-value pairs that were observed in the recording were also observed during the emulation. In fact, 35 *new* unique address-value pairs where observed, while 375 pairs were never exercised by `Lock`.

Finally, we wanted to ensure that our peripherals were actually causing the firmware to execute most of its code (versus an error handler or a simple surface-level function). To measure this, we used QEMU's trace feature to record every basic block and function that was executed in the emulator. While the emulated firmware executed 738 unique basic blocks in the firmware, it was unclear if these were "interesting" basic blocks (*i.e.,* executing notable functionality). Thus, we used `angr` [33], a popular binary analysis platform, to identify every basic block (609 in total) and function entry point (68 in total) that was reachable from the `loop()` function (the main function in Arduino firmware). This list was then compared against the execution trace from QEMU, revealing that the emulation executed 362 (59%) of those basic blocks and 58 (85%) of the functions. Indeed, these were not superficial functions either. The maximum depth of the call stack originating from `loop()`, as discerned by `angr`, was six. The emulation results are shown in Table 5.

Table 5: Depth of call graph executed in emulation for `Lock` (the maximum possible is 6).

Depth of Function	0	1	2	3	4	5	6
Number of Functions Executed	1	6	21	17	21	7	4

These measurements demonstrate that Conware is not only able to emulate firmware so that it survives, but that our models are successfully *coning* the firmware into executing its full functionality.

5 RELATED WORK

The handling of peripheral interactions is one of the linchpins of the dynamic analysis for embedded firmware.

Initial dynamic analysis techniques leveraged hardware-in-the-loop analysis, where all of the interactions with the peripherals are forwarded to the real device. Avatar [37] was the first such emulation framework. Similarly, Charm [35] targeted smartphone drivers. However, Charm is designed for kernel drivers rather than arbitrary peripherals. Prospect [21] forwards peripheral accesses at the `syscall` layer. However, the `syscall` interface does not exist in most of the bare-metal firmware.

Several optimizations have been made over naïvely forwarding all of the peripheral accesses to the hardware. Surrogates [22] significantly improves the forwarding performance of Avatar via customized hardware. Kammerstetter *et al.* [20] uses cached peripheral accesses to minimize the interaction with real hardware. And Avatar^2 [27] generalized caching by allowing the replaying of forwarded peripheral input and output without using the hardware.

Recently, several works proposed domain-specific models to handle peripherals. Here, effective peripherals models are carefully engineered, mostly manually, for a specific set of firmware. HALucinator [9] uses a model of a known HAL implementation to emulate embedded systems with the HAL. Similarly, PartEmu [18] and Ex-vivo [30] create peripheral models that can handle ARM-based, Trusted OSes, and Android kernel drivers, respectively.

Few works try to handle peripheral interactions in a generic manner automatically. Pretender [17] is a record-and-model approach that first records peripheral accesses, generates access models, and then tries to intelligently replay. However, Pretender needs debug access to the chip being modeled and only supports simplistic peripheral models. P^{2}IM is a fuzzing-based approach that generates *acceptable* inputs by randomly fuzzing the firmware [12]. P^{2}IM only considers on-chip peripherals and requires their abstract models to be generated manually and offline by a domain expert. This expert knowledge may be hard to provide for new peripherals. However, Conware automatically generates appropriate models for arbitrary peripherals without expert knowledge. Furthermore, Conware supports both on- and off-chip peripherals.

6 LIMITATIONS AND FUTURE WORK

While this work makes many advances in peripheral emulation, there are still many avenues that we believe will provide interesting future research. First, it is likely possible to make even more concise automata by relaxing the comparison of `Pattern` sub-models to permit more generalized states. More precisely, we currently only consider two "patterns" equal if they are identical to ensure that we do not lose any encoded information. However, it is conceivable, and realistic, that the actual number of items in the pattern does not matter, but only the order. For example, a status register that reads busy, and then ready would operation the same with 10 busy reads or 1,000. By constructing and automated reinforcement-learning based technique, we believe that identifying these instances in an automated-fashion is possible.

Second, we made a simplifying assumption that state transitions are strictly based on MMIO memory writes (*i.e.,* a write will instantly transition the state). However, in practice state transitions within peripherals can happen in ways that violate this model. For example, a state transition may occur based on a read value instead of a write value. In one scenario, the peripheral may not depend on the status register, by may instead transition after a certain value was read to indicate the *end*. In another scenario, the very act of reading a value, may alter the state, which is the case in it does not matter how many times `BUSY` is returned, but it does matter how many times a 1 is returned in an encoding scheme. While Conware does not currently support these cases, it could be easily extended to transition on specific reads by appending a superfluous write in the proper place during model construction and then annotating the appropriate read to force the state transition during emulation. However, detecting these instances remains unsolved. Expert knowledge or differential analysis techniques would be required to faithfully emulate a peripheral with these interactions.

Finally, accurately correlating interrupts to the specific write that triggered them, versus the state transition, makes more sense for some peripherals (*e.g.,* UART). More precisely, since interrupts

on encoded on state transition edges, Conware currently relies on the fact that the proper state will *eventually* be reached. However, in the case of the UART controller, each write to the `Interrupt Enable Register` register will trigger exactly one interrupt once the buffer is available. Thus, extending Conware to identify these interacts and trigger the interrupts accordingly would result in a more accurate emulation. To demonstrate the viability of this technique, we wrote a script to simply ($\approx$ 10 lines of Python) to *disable* buffering on our emulated UART controller for debugging purposes by always returning `TXRDY` in the `Status Register`. Again, this could be done through expert knowledge or more advanced pattern recognition techniques, which would require significant testing to avoid false positives. Methods for automatically detecting when these correlations do and do not hold will likely lead to more accurate emulation.

Finally, while we do not see never-before-seen reads and writes in our evaluation, this scenario is inevitable. Employing a method that leverages static analysis to deduce appropriate reactions, and modifying the automata (*i.e.,* perform on-the-fly static analysis to construct a suitable response) appropriately sounds particularly fruitful [1].

7 CONCLUSION

The ability to emulate a system for analysis is critical for most security analyses, and yet this tool is currently absent form the world of embedded systems, despite importance of securing these prolific systems. In this work, we present Conware, a system that is capable of automatically modeling hardware peripherals used by embedded systems, and uses these models to facilitate full-system emulation. Conware is a complete suite of software that facilitates recording peripheral interactions on real hardware, generating high-fidelity models from these recordings, and emulating firmware using popular emulation frameworks (*e.g.,* QEMU and Avatar^2). Conware's differentiator is that it is able to *merge* recordings in a pluggable way, enabling analysts to generate models based on one (or more) firmware recording and then use those models to execute a complete different firmware. This enables Conware to emulate the hardware for systems without access to the original source code or hardware, in an automated way. Conware was tested against various popular peripherals, and was able to successfully emulate all of them. Moreover, we demonstrated Conware's ability to emulate a black-box firmware sample by merging six independent models, which were generated using the sample code that accompanies each peripheral, to create an emulation environment that was suitable for the new, never-before-seen, firmware.

ACKNOWLEDGEMENTS

We would like to acknowledge Fabian Monrose for his invaluable feedback on this work, Noah Spahn for supporting our development efforts, and the various reviewers that helped to focus and strengthen this work through their comments. Similarly, we would like to thank Eric Gustafson and Marius Muench for their seminal work in the area and their support on this work.

This material is based upon work supported by the Office of Naval Research under Award No. N00014-17-1-2011, and by the Department of Homeland Security under Award No. FA8750-19-2-0005. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research or the Department of Homeland Security.

REFERENCES

[1] Open Review Mobicomm 2020. [n.d.]. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. https://openreview.net/pdf?id=rylaZ6iIDr.
[2] Altium. 2017. NEC Infrared Transmission Protocol. https://techdocs.altium.com/display/FPGA/NEC+Infrared+Transmission+Protocol.
[3] Atmel. 2015. SAM3X/ SAM3A Series (DATASHEET). https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf.
[4] BARRAGAN. 2013. Sweep. https://www.arduino.cc/en/Tutorial/Sweep.
[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.
[6] Jacob Beningo. 2016. Prototype to production: Arduino for the professional. https://www.edn.com/prototype-to-production-arduino-for-the-professional/.
[7] Duane Benson. 2015. Arduino as a rapid prototyping system. https://www.embedded.com/arduino-as-a-rapid-prototyping-system/.
[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16.
[9] 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. *Proceedings of the 29th USENIX Security Symposium (USENIX '20)* (2020).
[10] Furkan Comert and Tolga Ovatman. 2015. Attacking state space explosion problem in model checking embedded TV software. *IEEE Transactions on Consumer Electronics* 61, 4 (2015), 572–579.
[11] 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*. 1–11.
[12] Bo Feng, Alejandro Mera, and Long Lu. 2020. P^{2}IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling (extended version). *Proceedings of the 29th USENIX Security Symposium (USENIX '20)* (2020).
[13] Jack Ganssle. 2004. Reentrancy. In *The Firmware Handbook*. Elsevier, 231–244.
[14] Geeetech. 2012. Arduino IR Remote Control. http://www.geeetech.com/wiki/index.php/Arduino_IR_Remote_Control.
[15] Giovani Gracioli and Sebastian Fischmeister. 2012. Tracing and recording interrupts in embedded software. *Journal of Systems Architecture* 58, 9 (2012), 372–385.
[16] Joe Grand and July Friday. 2004. Advanced hardware hacking techniques. *DEFCON* 12 (2004), 59.
[17] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 135–150.
[18] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. 2020. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX '20)*.
[19] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.
[20] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*.
[21] 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*. 329–340.
[22] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*.
[23] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.
[24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[25] ARM Limited. 2010. Cortex-M3 Technical Reference Manual (Revision r2p1). http://users.ece.utexas.edu/~valvano/EE345L/Labs/Fall2011/CortexM3_TRM_r2p1.pdf.

[26] 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.

[27] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (Colocated with NDSS Symposium)*, Vol. 18. 1–11.

[28] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien 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 (NDSS)*.

[29] Osbourne, Paul. [n.d.]. CMSIS-SVD Repository and Parsers. https://github.com/posborne/cmsis-svd.

[30] Ivan Pustogarov, Qian Wu, and David Lie. [n.d.]. Ex-vivo dynamic analysis framework for Android device drivers. ([n. d.]).

[31] 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 *2020 IEEE Symposium on Security and Privacy (SP)*. 431–448.

[32] Miro Samek. 2016. State Machines for Event-Driven Systems. https://barrgroup.com/embedded-systems/how-to/state-machines-event-driven-systems.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[34] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[35] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX '18)*. 291–307.

[36] LLC. Where Labs. 2019. Bus Pirate. http://dangerousprototypes.com/docs/Bus_Pirate.

[37] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Vol. 14. 1–16.