Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation

Chen Cao*
The Pennsylvania State University, Behrend College
ccao@psu.edu

Le Guan*
University of Georgia
leguan@cs.uga.edu

Jiang Ming
University of Texas at Arlington
jiang.ming@uta.edu

Peng Liu
The Pennsylvania State University
pxl20@psu.edu

ABSTRACT

With the rapid proliferation of IoT devices, our cyberspace is nowadays dominated by billions of low-cost computing nodes, which are very heterogeneous to each other. Dynamic analysis, one of the most effective approaches to finding software bugs, has become paralyzed due to the lack of a generic emulator capable of running diverse previously-unseen firmware. In recent years, we have witnessed devastating security breaches targeting low-end microcontroller-based IoT devices. These security concerns have significantly hamstrung further evolution of the IoT technology. In this work, we present `Laelaps`, a device emulator specifically designed to run diverse software of microcontroller devices. We do not encode into our emulator any specific information about a device. Instead, `Laelaps` infers the expected behavior of firmware via symbolic-execution-assisted peripheral emulation and generates proper inputs to steer concrete execution on the fly. This unique design feature makes `Laelaps` capable of running diverse firmware with no *a priori* knowledge about the target device. To demonstrate the capabilities of `Laelaps`, we applied dynamic analysis techniques on top of our emulator. We successfully identified both self-injected and real-world vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; Software reverse engineering; • **Computer systems organization → Firmware**.

KEYWORDS

microcontroller, firmware emulation, symbolic execution

ACM Reference Format:
Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427280

*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC 2020, December 7–11, 2020, Austin, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427280

1 INTRODUCTION

Software-based emulation techniques [43] have demonstrated their pivotal roles in dynamically analyzing binary code. Running a program inside an emulator allows analysts to gain semantically insightful run-time information (e.g., execution path and stack layout) and even dynamically instrument the binaries [6, 32, 36, 38]. However, these capabilities are difficult to be deployed to analyze the firmware of low-end microcontroller-based IoT devices. A major obstacle is the absence of a versatile emulator that could execute arbitrary firmware of different microcontroller devices. Such an emulator has to deal with the vast diversity of microcontroller firmware in terms of hardware architecture (e.g., x86, ARM, MIPS, etc), integrated peripherals (e.g., communication interface, DSP, etc.), and the underlying operating system (e.g., bare-metal, mBed OS, FreeRTOS, etc.). Customizing the emulator for every kind of device is nearly impossible.

Dynamically analyzing embedded firmware has been studied for a while. Unfortunately, existing solutions are far from mature in many ways. They are either ad-hoc, tightly coupled with real devices, or rely on an abstraction layer such as the Linux kernel. Existing work [25, 28, 34, 44, 46] forwards peripheral signals to real devices and run the rest of firmware in an emulator. In this way, analysts could execute the firmware and inspect into the inner state of firmware execution. However, this approach is not affordable for testing large-scale firmware images because for every firmware image a real device is needed. Besides, frequent rebooting of the device and signal forwarding are time-consuming. Recent work advances this research direction by modeling the interactions between the original hardware and the firmware [19, 23]. This enables the virtualized execution of any piece of firmware possible without writing a specific back-end peripheral emulator for the hardware. However, existing approaches either require the real hardware to "learn" the peripheral interaction model [23], or cannot handle complex firmware logic [19]. In particular, P^2IM restricts values to be returned by the peripheral in a small subset of all possible values. This is fundamentally limited by the nature of concrete execution. Previous work also leverages the abstraction layer is simplify the problem [9, 14, 48]. However, it relies on the presence of the Linux kernel [9, 48] or the hardware abstraction layer in the firmware [14].

In this work, we demonstrate that the obstacles of device-agnostic firmware execution are not insurmountable. We present `Laelaps`,

a generic emulator for ARM Cortex-M based microcontroller units (MCUs). Instead of implementing peripheral logic for every device, we leverage symbolic execution and satisfiability modulo theories (SMT) [3] to reason about the expected inputs from peripherals and feed them to the being-emulated firmware on the fly. Therefore, our approach aims to achieve the ambitious goal of executing non-Linux firmware without relying on real devices. The design of `Laelaps` combines concrete execution and symbolic execution. Concrete execution runs in a full system emulator, QEMU [4], to provide the inner state of execution for dynamic analysis. However, the state-of-the-art whole system emulators cannot emulate previously-unseen peripherals. If the firmware accesses unimplemented peripherals, the emulation will become paralyzed. Symbolic execution then kicks in to find a proper input for the current peripheral access operation and guides firmware execution. We found that symbolic execution is particularly good at inferring peripheral inputs, because many of them are used in logical or arithmetical calculations to decide a branch target.

In general, `Laelaps`'s concrete execution will be stuck when accessing an unimplemented peripheral, and then it switches to the symbolic execution to find proper inputs that can guide QEMU to a path that is most likely to be identical with a real execution. One significant practical challenge for automatic test generation is how to effectively explore program paths. Various search heuristics have been proposed to mitigate the path explosion problem in PC software [30, 41, 45]. However, peripherals reveal many distinct features that require special treatment, such as very common infinite loops and interrupt requests. At the heart of our technique is a tunable path selection strategy, called Context Preserving Scanning Algorithm, or `CPSA` for short. `CPSA` contains a set of peripheral-specific heuristics to prune the search space and find the most promising path. Peripherals also interact with the firmware through interrupts. In fact, embedded systems are largely driven by interrupts. QEMU has built-in support for interrupt delivering, but it has no knowledge with regard to when to assert an interrupt—this logic should be implemented by peripherals. We address this issue by periodically raising interrupts which have been activated by the firmware. Although our solution may not strictly follow the path on a real device, we demonstrate that it is able to steer the execution to properly initialized and valid points suitable for further analysis.

We have developed `Laelaps` on top of angr [42] and QEMU [4]. Our prototype focuses on ARM Cortex-M MCUs, which dominate the low-end embedded device market, but the design of `Laelaps` is applicable to other architectures as well. We evaluate `Laelaps` by running 30 firmware images built for 4 development boards. The tested firmware spans a wide spectrum of sophistication, including simple synthetic programs as well as real-world IoT programs running Amazon FreeRTOS OS [1]. Our work makes the following main contributions:

- We abstract the system model of ARM Cortex-M based embedded microcontroller devices and distill the missing but essential parts for full system emulation of those devices.
- We fill the missing parts of full system device emulation by designing a symbolically-guided emulator, which is capable of running diverse firmware for ARM MCUs with previously-unknown peripherals.
- We demonstrate the potential of `Laelaps` by using it in combination with advanced dynamic analysis tools, including boofuzz [37], angr [42], and PANDA [17]. `Laelaps` is an open-source tool availabe at https://github.com/dongmu/Laelaps.

2 BACKGROUND

2.1 ARM Cortex-M Microcontroller

Previously, microcontroller units were often considered as specialized computer systems that are embedded into some other devices, as contrary to personal computers or mobile *SoC*. With the emergence of IoT, now they have been central to many of the innovations in the cost-sensitive and power-constrained IoT space.

ARM Cortex-M family is the dominating product in the microcontroller market. These devices support Thumb instruction set for the most efficient code density. From the view point of a programmer, the most remarkable difference between PC/mobile processors and Cortex-M processors is that Cortex-M processors do not support MMU. This means that the application code and the OS code are mingled together in a flat memory address space. For this reason, it does not support the popular Linux kernel.

ARM Cortex-M processors map everything into a single address space, including the ROM, RAM and peripherals. Therefore, peripheral functions are invoked by accessing the corresponding registers in the system memory. For each ARM core, ARM defines the basic functionality and the memory map for its core peripherals, such as the interrupt controller (Nested Vector Interrupt Controller or NVIC), system timer, debugging facilities, etc. Then, ARM sells the licenses of its core design as intellectual property (IP). The licensees produce the physical cores. These participating manufactures are free to customize their implementations as long as they conform to the core ARM design. As a result, manufactures optimize and customize their products in different ways, leading to a vast diversity of Cortex-M processors.

2.2 Firmware Execution

Th MCU firmware execution can be roughly divided into four phases: 1) device setup, 2) base system setup, 3) RTOS initialization, and 4) task execution. In the device setup phase, the hardware components, including RAM and peripherals, are turned on and self-tested. In the base system setup phase, standard libraries such as libc are initialized. That means dynamic memory can be used, and static memory is allocated. Then the code of a RTOS (or bare-metal) image is copied into the allocated memory regions, and core data structures are initialized. If the firmware is powered RTOS, the scheduler is also started. Finally, multiple tasks are executed on the processor in a time-sharing fashion (in case of RTOS design) or a single-purpose task monopolizes the processor (in case of bare-metal design).

Firmware execution highly depends on the underlying hardware, and *such hardware uncertainties have become the biggest barrier to the development of a generic emulator.* An improper emulation leads to failed bootstrap very early in phase 1. We also note that there can be multiple valid execution paths in a firmware execution. In fact, manufacturers often include multiple driver versions to normalize different peripherals. All the valid paths can lead to a successful

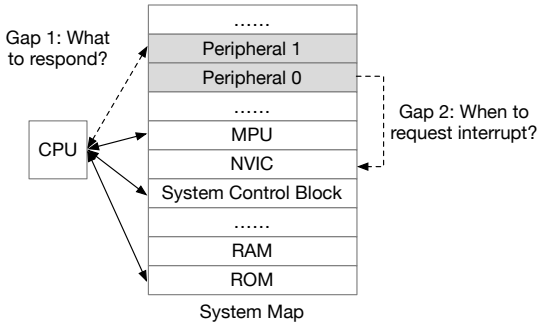

Figure 1: Missing logic in QEMU (shaded memory regions correspond to unimplemented peripherals).

execution. In other words, *the executed driver version, as long as it is valid, does not influence the result of firmware analysis.* This fact grants us a certain level of fault tolerance in firmware emulation. That is, a wrongly selected path can still lead to a successful emulation for analysis.

2.3 Dynamic Symbolic Execution

Symbolic execution, first proposed by King [27], is a powerful automated software testing technique. It treats program inputs as symbolic variables and simulates program execution so that all variables are represented as symbolic expressions. Together with theorem proving technique [20, 33], symbolic execution is able to automatically generate concrete inputs that cover new program paths. Notably, symbolic execution has achieved encouraging results in testing closed-source device drivers [10, 29, 31, 39]. However, existing work mainly focuses on analyzing drivers in the full-fledged OSs. MCU firmware exhibits very differently challenges that require special treatment. Dynamic symbolic execution (a.k.a concolic execution) [7, 8, 21, 40] performs symbolic execution along a concrete execution path, and it combines static and dynamic analysis in a manner that gains the advantages of both. Dynamic symbolic execution has achieved remarkable success in generating high-coverage test suites and finding deep vulnerabilities in commercial software [5, 11, 22, 31]. The core of `Laelaps` is a concolic execution approach for peripheral emulation. One particular challenge for symbolic execution is the path explosion problem [30, 41, 45]. Our study proposes a set of peripheral-specific search heuristics to mitigate the path explosion.

3 OVERVIEW

3.1 Function Gap

QEMU [4], the most popular generic machine emulator, has built-in support for almost all of the functions defined by ARM. We call them *core peripherals/functions* in the remainder of this paper. However, chip manufacturers often integrate custom-made peripherals that are also mapped into the address space of the system. The logic of these peripherals, together with the core peripherals, define the behavior of an ARM MCU device. Therefore, to emulate a real device, an emulator needs to support all the manufacturer-specific peripherals. However, our source code review shows that QEMU, the state-of-the-art emulator, only supports three ARM-based microcontrollers (two TI Stellaris evaluation boards and one ARM SSE-200 subsystem device). For unsupported devices, QEMU only emulates the core peripherals defined by ARM. Figure 1 shows the missing logics in QEMU.

When the processor interacts with an unimplemented peripheral (shown as shaded in Figure 1), QEMU becomes paralyzed due to two unfilled gaps. **Gap 1:** QEMU does not know how to respond when the processor accesses an unimplemented peripheral register. **Gap 2:** QEMU lacks the logic of unimplemented peripherals and therefore cannot know when to send interrupt requests.

3.2 Motivating Observations

QEMU becomes paralyzed when the firmware access an unimplemented peripheral, simply because it cannot provide a suitable value to the firmware. If QEMU provides a random value, the execution is very likely to be stuck. Our in-depth study on the usage of peripheral values leads to three key observations. First, a large amount of peripheral accesses do not significantly influence firmware execution. As shown below, this statement reads a value from peripheral register `base->PCR[pin]` and assigns another value to the same register after some logic calculations. This statement configures the functionality of a pin on the board, but the values being read and written do not influence the firmware emulation at all.

```
base->PCR[pin] = (base->PCR[pin] & ~PORT_PCR_MUX_MASK) |
      PORT_PCR_MUX(mux);
```

Second, the rest of peripheral accesses that do actually influence firmware execution have an important effect on the execution path, and therefore, it is crucial to model them correctly. Third, if we can find a value that drives the execution along a correct path, then QEMU can usually execute the firmware as expected.

To explain this, we list a code snippet for a *UART* driver in Listing 1. It outputs a buffer through the UART interface. In Line 3, it reads from a *UART* register (`base->S1`) in a while loop. Only if the register has certain bits set would the loop be terminated. Then the driver will send out a byte by putting the byte on another register (`base->D`). It is clear that executing line 4 is necessary for the firmware to move forward. To obtain the input leading to line 4, we found symbolic execution a perfect fit. Specifically, if we mark the value in the unknown register (`base->S1`) as a symbol, we can instantly deduce a satisfiable value to reach line 4. Like this example, we found many peripheral drivers use peripheral registers in simple logic or arithmetic calculations, and then the results are used in control-flow decision making.

```
1 void UART_WriteBlocking(UART_Type *base, const uint8_t *data,
       size_t length){
2     while (length--){
3         while (!(base->S1 & UART_S1_TDRE_MASK)){}
4         base->D = *(data++);
5     }
6 }
```

Listing 1: Code snippet from real driver code.

3.3 Laelaps Overview

`Laelaps` combines concrete execution and symbolic execution, namely concolic execution [7, 8, 21, 40]. Neither of them alone could achieve our goal because 1) concrete execution cannot deal

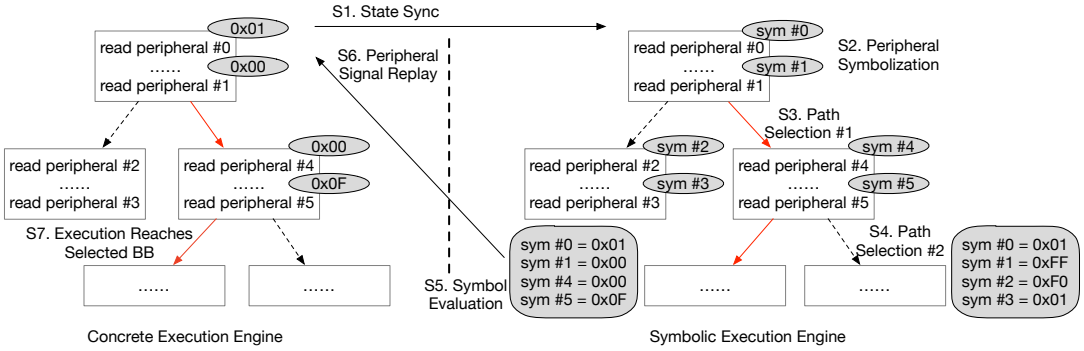

Figure 2: Laelaps's branch exploration with the depth of two. The red color branches are selected by Laelaps.

with unimplemented peripherals; and 2) pure symbolic execution faces the traditional path explosion problem. We design our system based on concrete execution but employ symbolic execution to run small code snippets to calculate suitable values for unimplemented peripheral inputs. In this way, a firmware image runs concretely and symbolically by turns, gaining the advantages of both.

Laelaps only needs basic information of a device to initialize the execution environment. Specifically, it requires the target architecture profile (e.g., ARM Cotex-M0/3/4) and locations of ROM and RAM. Then it loads ARM core peripherals into the system map. Next, memory regions used by chip-specific peripherals are marked as *unimplemented* (e.g., the 0x40000000–0x400FFFFF region is used as peripheral memory map in NXP chips). Accesses to them are intercepted and handled in the symbolic execution engine. All the other memory regions are unmapped and should never be accessed. An access to the unmapped memory indicates a potential memory corruption, which can be used by a fuzzing tool to capture crashes. QEMU translates and emulates each instruction of firmware until there is a read operation to an unimplemented memory. Our goal is to predict a proper response. Peripheral write operations, on the other hand, are ignored because they do not influence program status in any way. As shown in Figure 2, when an unimplemented read operation is detected, the processor context and memory are then synchronized to the symbolic execution engine (S1).

During symbolic execution, every unimplemented peripheral access is symbolized (S2), resulting in a list of symbols. Each time a branch is encountered, we run a path selection algorithm (S3/4) that chooses the most promising path (see §4.3). Symbolic execution advances along the path until one of following events is detected:

E1: Synchronous exception (e.g., software interrupt)
E2: Exception return
E3: Long loop (e.g., memcpy)
E4: Reaching the limit of executed branches

E1 and E2 terminate symbolic execution because these system level events cannot be easily modeled by existing symbolic execution engines (§4.2). E3 could consume a lot of time in symbolic execution. Therefore, whenever detected, the execution should be transferred to the concrete engine (§4.2.5). We do not allow emulation to stay in symbolic engine forever due to the path explosion problem. Therefore, we set a limit for the maximum branches to encounter in each symbolic execution (§4.3). In Figure 2, we illustrate a case in which we set this limit as two.

At the time when symbolic execution is terminated, we evaluate the values of the list of symbols that navigate execution to the current path (S5) and feed the solved values to QEMU (S6). Since these values are verified via the constraint solver, they will guide the concrete execution to follow the selected promising path. Specifically, QEMU re-executes the path explored by the symbolic execution engine by following the solved values. In this paper, we call each switching to symbolic engine a symbolic execution pass. Laelaps pushes firmware execution forward by continuously switching between QEMU and symbolic execution passes. In this way, we provide a platform that execute the firmware to a state suitable for further dynamic analysis (e.g., examining a hard-to-reach code logic that was only possible with a real device previously). It leaves to analysts to decide the right time to dig into firmware execution and perform further analysis. How to dynamically analyze the firmware is out of the scope in this paper. We expect many tools can be directly benefited from Laelaps because our design is not specific to a particular tool. Nevertheless, we showcase one of its applications (fuzz testing) in §6.3.

4 LAELAPS SYSTEM DESIGN

In this section, we present the details of Laelaps system design, limitations, and our mitigations.

4.1 State Transfer

Whenever an unimplemented peripheral read is detected, the program state is transferred to the symbolic execution engine. In our current design, Laelaps synchronizes the processor context (general purpose registers, system registers) of the current execution mode to the symbolic execution engine. Since copying all RAM is expensive, we adopt a copy-on-access strategy that only copies required pages on demand. After symbolic execution, modified memory is not synchronized to QEMU. Instead, it is re-constructed in QEMU by following the same path explored by the symbolic execution engine.

4.2 Symbolic Execution

4.2.1 Basic Rule #1. Since the symbolic execution engine is invoked by unimplemented peripheral read operations, the first instruction in the symbolic engine is always a peripheral read. We generate a symbolic variable for this memory access. Likewise, the following peripheral read operations are also assigned with symbols. Note that even if a peripheral address has been accessed earlier, we still assign a new symbol. This is because of the volatile nature of peripheral memory – their values change nondeterministically due to unforeseen events generated externally. In this sense, ***we assign new symbols spatially (different addresses get different symbols) and temporally (different times get different symbols)***.

4.2.2 Basic Rule #2. Firmware may contain OS-level functions that inevitably involve the interaction between tasks and event handlers running in the separated privileged mode. Our current symbolic execution cannot correctly handle complex context switches due to exceptions. Therefore, in each symbolic execution, we set a basic rule that ***the execution should always stick to the original execution mode***. To meet this rule, for each explicit instruction that requires context switch, we immediately terminate symbolic execution and transfer the execution to QEMU. This includes synchronous exception instruction such as supervisor calls (SVC) and exception returns. In an exception return, the processor encounters a specially encoded program counter (PC) value and fetches the real PC and other to-be-restored registers from the stack.

4.2.3 Basic Rule #3. As discussed in §3.3, `Laelaps` holds multiple solved symbols to be replayed. In essence, `Laelaps` expects QEMU to follow exactly the same path explored during symbolic execution. This implies that QEMU should not take any asymmetric exceptions when replaying the buffered symbol values. Otherwise, the QEMU execution is deviated from the expected path, rendering the solved symbols useless. We can certainly discard the remaining solved symbols on a path deviation caused by exceptions. However, since symbolic execution is expensive, we opt to adopt another practical approach. That is, we set a basic rule that ***QEMU resumes replaying without accepting any exceptions until all of the solved symbols are consumed***.

4.2.4 Unrecognized Instructions. Currently, state-of-the-art symbolic execution engines cannot recognize system-level ARM instructions. We take another two strategies to handle this. First, for the unrecognized instructions that do not affect program control flow, we replace them with NOP instructions. This includes many instructions without operands (e.g., DMB, ISB), instruction updating system registers (e.g., MSR), and breakpoint instruction BKPT. Second, for the unrecognized instructions that directly change control flow (e.g., SVC) or update general purpose registers (e.g., MRS), we immediately terminate symbolic execution and switch to QEMU.

4.2.5 Long Loop Detection. Symbolic execution is much slower than concrete execution. Therefore, we need to keep the time spent on symbolic execution as little as possible but at the same time yield similar predicted paths. When encountering long loops controlled by concrete counters, the loop would be executed symbolically until the loop is finished. Unfortunately, there are numerous such long loops in a firmware. Examples include frequently used library functions such as `memcpy` and `memset`. Since those functions usually contain long loops, symbolically executing them is extremely inefficient. `Laelaps` is able to detect long loops at run-time. If a long loop is detected, the execution is forced to be transferred to QEMU. To detect long loops, `Laelaps` maintains the execution trace based on recently executed basic blocks and finds the longest repeated cycle. Whenever the longest repeated cycle is longer than a threshold, symbolic execution will be terminated. Based our empirical study, we set this threshold as 5 in the experiments.

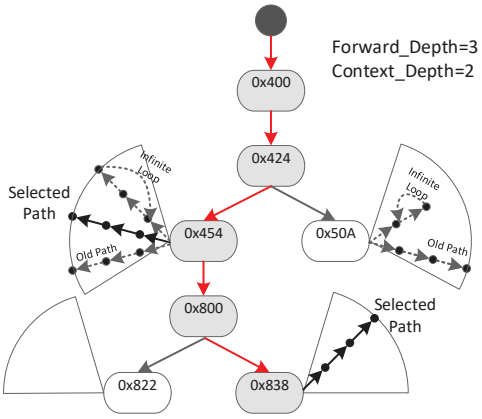

Figure 3: The illustration of `Laelaps`'s path selection strategy: Context Preserving Scanning Algorithm. Executed path is represented by red edges in the CFG. In each sector, CPSA explores all possible paths within Forward_Depth steps. At the node `0x424`, two branches are explored. We choose the left-hand branch because it has the most promising path.

4.3 Path Selection Strategy: Context Preserving Scanning Algorithm

The goal of `Laelaps`'s symbolic execution is to find the most promising path and direct QEMU towards this path. A promising path is one that passes firmware's internal checks and avoids disrupting firmware execution (e.g., assertion failure and infinite loop). Since we lack the high-level semantic information about data structures and control flow, it is particularly challenging. At a high level, we try to use symbolic execution to avoid generating problematic peripheral inputs that may lead to disrupted execution, and this is achieved by a set of firmware specific heuristics.

Figure 3 shows how the proposed path selection strategy – Context Preserving Scanning Algorithm (or `CPSA` for short) – works in general. There are two parameters that can be used to adjust the performance and accuracy of the algorithm. `Context_Depth` specifies the number of branches the symbolic engine has to accumulate before invoking the constraint solver and returning to the QEMU. `Forward_Depth` is the maximum number of basic blocks that the symbolic engine can advance from a branch. With `Context_Depth` set to two, each symbolic execution pass decides the results for two branches (from `0x424` to `0x454` and `0x800` to `0x838`). Note that before reaching a point to decide a branch, there might have been multiple basic blocks executed. These intermediate basic blocks end with a single branch or the corresponding conditions are determined by concrete values. We call an execution leading to a branch

selection as a step, following the naming convention of angr [42]. With `Forward_Depth` set to three, symbolic engine explores as many as three future steps for each branch. When encountering a new branch in a step, both branches are explored. As shown in the Figure 3, there are two branches at the end of basic block `0x424`. The left-hand branch leads to three distinct paths within `Forward_Depth` steps, while the right-hand branch leads two. Our algorithm selects the most promising one among all of the paths. In this figure, we choose a path starting from the left-hand branch. Therefore, we pick the `0x454` branch to follow the `0x424` branch. Similarly, at the node `0x800`, the right-hand branch is selected.

4.3.1 Explanatory Example. Listing 2 is a code snippet of an Ethernet driver from the NXP device SDK. The function `enet_init` initializes the Ethernet interface, which calls `PHY_Init` to configure the *Network Interface Controller* (NIC) with a *physical layer* (PHY) address. If the invocation fails, the execution will be suspended and lead to calling an assert function in line 5, which is an infinite loop. Inside `PHY_Init`, `PHY_Write` interacts with NIC for actual configuration. Lines 10 and 12 invoke `PHY_Write` twice. If either invocation fails, `PHY_Init` returns with a failed result. If both of them returns `kStatus_Success`, the program checks whether the operations have been successful by reading back peripheral memories using `PHY_Read`, as indicted in lines 16 and 18. Different from `PHY_Write` in lines 10 and 12, there are multiple chances indicted by `counter` for the two `PHY_Read` functions to obtain the expected result. If so (line 19), a short loop is executed to wait until the state is stable (lines 21-24). In a word, a correct execution trace is expected to follow "3-10-11-12-13-15-16-17-18-19-(21-23-24)*-25-34-4-6".

```
1 static void enet_init(...){
2   ...
3   status = PHY_Init(...);
4   if (kStatus_Success != status)
5     LWIP_ASSERT("\r\nCannot initialize PHY.\r\n", 0);
6 }
7
8 status_t PHY_Init(...){
9   ...
10  result = PHY_Write(...);
11  if (result == kStatus_Success) {
12      result = PHY_Write(...);
13      if (result == kStatus_Success) {
14 /* Check auto negotiation complete. */
15          while (counter --) {
16              result = PHY_Read(..., &bssReg);
17              if ( result == kStatus_Success) {
18                  PHY_Read(..., &ctlReg);
19                  if (((bssReg & ...) && (ctlReg & ...)) {
20 /* Wait a moment for Phy status stable. */
21                      for (timeDelay = 0; timeDelay <
        PHY_TIMEOUT_COUNT; timeDelay ++) {
22 /* Must be here to succeed. */
23                          __ASM("nop");
24                      }
25                      break;
26                  }
27              }
28              if (!counter) {
29                return kStatus_PHY_AutoNegotiateFail;
30              }
31          }
32      }
33  }
34  return result;
35 }
```

Listing 2: Source code of a complex Ethernet driver.

4.3.2 Heuristic #1: Context Preservation. `Laelaps` steers firmware execution forward by continuously switching between QEMU and symbolic execution passes. Each symbolic execution pass only makes decision based on the current context instead of a holistic context. Therefore, it cannot make an optimal decision globally. Lines 16-19 in Listing 2 clearly demonstrate this. In line 16 and line 18, there are two `PHY_Read` invocations that read a symbolic value to `bssReg` and `ctlReg` respectively. In line 19, these two symbols are used to determine a branch. If we transfers execution to QEMU after line 16, the condition in line 19 might never be satisfied, because at that time `bssReg` is already a concrete value, which might equal to zero. The root reason is that we concretize `bssReg` too early and it later affects the subsequent path to be taken. We call this "over-constraining".

Inspired by *speculative symbolic execution* [47], we do not invoke the constraint solver when encountering `bssReg`. Instead, our symbolic execution advances along the path and solves `bssReg` together with `ctlReg` in line 19. More generally, we allow analysts to configure a parameter `Context_Depth`, which is the specified number of branches the symbolic engine has to accumulate before invoking the constraint solver. In this way, we preserve the possibilities of future paths and thus yielding more accurate results. The downside is that a larger `Context_Depth` leads more paths to be explored in symbolic execution, and so it consumes more time. Therefore, `Context_Depth` serves as an adjustable parameter for a trade-off between fidelity and performance.

4.3.3 Heuristic #2: Avoiding Infinite Loop. Symbolic execution becomes entangled in an infinite loop. As shown in Listing 2, any failed invocations to `PHY_Write` or `PHY_Read` will trigger the execution of line 5, an infinite loop. We allow analysts to specify a parameter `Forward_Depth`, which is the maximum number of steps that the symbolic engine can advance from a branch. Within `Forward_Depth` steps, a branch could lead to multiple paths. If all of these paths have an infinite loop, this branch is discarded. If `Laelaps` singles out a branch because all the other branches are eliminated due to infinite loop detection, we say `Laelaps` chooses this branch on the basis of **infinite-loop-elimination**. To identify an infinite loop, our symbolic engine maintains the execution traces and states of explored paths. It then compares execution states within each path. If any two states are the same, meaning the processor registers do not change at different times, we regard the corresponding path as an infinite loop.

The **infinite-loop-elimination** heuristic might incorrectly filter out a legitimate path which seems to be a infinite loop. For example, a piece of code may constantly queries a flag in the RAM, which is only changed by an interrupt handler. Since the symbolic execution engine is not interrupt-aware in our design, the legitimate path is filtered out. To address this issue, `CPSA` does not filter out paths with infinite loops completely. Instead, they are selected at the lowest priority. In this way, when the execution is switched back to the QEMU, an interrupt can be raised and handled (§4.4), effectively unlocking the infinite loop.

4.3.4 Heuristic #3: Prioritizing New Paths. We maintain a list of previously executed basic blocks and calculate a similarity measurement between the historical path and each of the explored future paths. We prioritize the candidate path with the lowest similarity,

implying that a new path is more likely to be selected. To illustrate how this heuristic helps us find the correct path of the code in Listing 2, consider how we can advance to line 21. As shown in line 15, there are `counter` chances that `Laelaps` can try to solve the correct values for `bssReg` and `ctlReg`. If an incorrect value is drawn from angr due to under-constrained path selection, the execution starts over from line 16. If our algorithm makes mistakes continuously in the while loop, the same path pattern will be recorded for many times. Eventually, this will activate similarity checking so that a new path (line 21) is selected. If `Laelaps` singles out a branch, we say `Laelaps` chooses this branch on the basis of **similarity**.

4.3.5 Heuristic #4: Fall-back Path. After applying the above-mentioned path selection mechanisms, if we still have multiple candidate paths, we choose the one with the highest address. This is based on two observations. First, programs are designed to execute sequentially. Second, the firmware typically initializes each peripheral one by one. Therefore, our algorithm tends to move forward quickly.

`Laelaps` has to choose a fall-back branch if neither the **infinite-loop-elimination** basis nor the **similarity** basis can single out a branch. In this case, we say `Laelaps` chooses this branch on the basis of **fall-back path**.

4.4 Interrupt Injecting

So far, we have presented how `Laelaps` fills **gap 1** shown in Figure 1. That is, how to support firmware sequential execution even if the firmware access unimplemented peripherals. On the other hand, in addition to generating data for the firmware to fetch, peripherals also notify the firmware when the data are ready through the interrupt mechanism. Typical, a firmware for embedded application just waits in low-power mode, and it only wakes up when receiving an interrupt request. Therefore, without being activated by interrupts (**gap 2**), most firmware logic remains dormant.

To fill **gap 2**, we implement a python interface that periodically delivers activated interrupts. This simple design works fine for two reasons. First, in a real execution, firmware only activates a limited number of interrupts. Therefore, delivering activated interrupts will not introduce too much performance penalty. Second, an interrupt handler can often gracefully deal with unexpected events. Although additional code is executed, they will not cause great impacts on firmware execution.

4.5 Limitations & Mitigations

`Laelaps` is designed to automatically reason about the expected peripheral inputs with only access to the binary code. However, it is impossible to exactly follow the semantic of the firmware in certain circumstances. If the peripheral inputs do not influence control flow, the solution made by symbolic execution would be random. We summarize common pitfalls to complicate automatic firmware execution and how we handle them.

4.5.1 Data Input. `Laelaps` works well when the peripheral inputs only decide control flow. However, the firmware also interacts with the external world by exchanging data. Data exchange is supported by fetching data from a particular data register at the agreed time slots. Obviously, we cannot feed the randomly generated data by symbolic execution to the firmware. Fortunately, in many dynamic analyses, these input channels are intercepted and fed with manually generated test-cases. In other words, `Laelaps` does not need to generate the inputs anyway. In §6, we show how we use `Laelaps` to hook network functions in FreeRTOS and reproduce the vulnerabilities in the `TCP_IP` stack of FreeRTOS [26].

4.5.2 Lack of Holistic Analysis. `Laelaps` preserves context information by staying in the symbolic engine for up to `Context_Depth` branches. However, `Context_Depth` cannot be set too large as it will slow down performance significantly. If a sub-optimal solution is generated under a low `Context_Depth`, the execution could go wrong. To overcome this limitation, we design several interfaces that analysts can leverage to override the solution from the symbolic execution engine and thus avoid unwanted execution. Analysts usually identify a false or unexpected execution when the firmware goes into an infinite loop or a crash. Then based on the execution trace, analysts override the solution accordingly. As shown in our evaluation, with necessary human inputs, `Laelaps` succeeds in dynamically running very complex firmware images.

5 IMPLEMENTATION

We developed the prototype of `Laelaps` based on QEMU [4] and angr [42], which are concrete execution engine and symbolic execution engine, respectively. To facilitate state transfer between the two execution engines, we integrate Avatar [34, 46], a Python framework for seamlessly orchestrating multiple dynamic analysis platforms, including QEMU, real device, angr, PANDA [17], etc. Our tool inherits the state transfer interface of Avatar, enhances Avatar's capability to handle Cortex-M devices, implements a memory synchronization mechanism between QEMU and angr, develops the proposed `CPSA` on top of angr, and exports to firmware analysts an easy-to-use Python interface. Our tool emulates a generic Cortex-M device on which firmware analysts can load and execute the firmware that interacts with unimplemented peripherals. These are implemented by 854 lines of Python code and 209 lines of C code (QEMU modification).

5.1 Configuration

Although `Laelaps` does not need prior knowledge about peripherals, some essential information about the chip is required. This information includes 1) the core being used (e.g., Cortex-M0, M3 or M4), 2) the mapping range of ROM/RAM, 3) the mapping ranges of chip-specific peripherals, and 4) how the firmware should be loaded (i.e., how each section of a firmware image corresponds to the memory map). The chip information can be oftentimes obtained from the official product description page, third-party forums, or the Federal Communications Commission (FCC) ID webpage [18]. But we acknowledge that there is a small portion of devices that use custom chips or non-publicly documented microcontrollers. To get information about how the firmware is loaded, moderate static analysis is required. In the simplest form, a raw firmware image as a whole is directly mapped from the beginning of the address space. This kind of image can be easily identified based on some characteristics (e.g., it starts with an initial stack pointer and an exception table) [2].

5.2 Peripheral Access Interception

When firmware accesses an unimplemented address specified in the configuration stage, the memory request is forwarded to the angr for symbolic execution. Our implementation is largely inherited from Avatar. In particular, Avatar implements a remote memory mechanism in which accesses to an unmapped memory region in QEMU are forwarded to a Python script. The Python script then emulates the behavior of a real peripheral and feeds the result to QEMU. Note that to symbolically execute the firmware, angr needs the current processor status (i.e., register values) and memory contents. Avatar fetches the processor status through a customized inter-process protocol and memory contents through the GDB interface. Unfortunately, in `Laelaps`, we cannot use the GDB interface for memory synchronization (explained below). We made modifications to Avatar so that additional Cortex-M specific registers (e.g., *Program Status Register* (PSR)) are synchronized to angr, and implemented our own memory synchronization interface.

5.3 Memory Synchronization

As mentioned earlier, Avatar uses the GDB interface to synchronize memory. The Avatar authors demonstrate this feature by synchronizing the state of a Firefox process from QEMU to angr and continuing executing it symbolically. Note that to invoke GDB for memory access, the target must be in the stopped state. However, in `Laelaps`, we cannot predict the program counters that access unimplemented peripherals and make breakpoints beforehand. An alternative to this issue is to invoke QEMU's internal function to stop the firmware execution at the time of unimplemented peripheral access. Unfortunately, due to the design model of QEMU, this idea cannot be achieved without significant modifications to QEMU.

We address this problem by exporting all RAM regions through IPC. Specifically, in QEMU, when a RAM region is created, we create a POSIX shared memory object and bind it with the RAM region using `mmap`. As a result, angr is able to directly address the firmware RAM by reading the exported shared memory object. Our solution significantly outperforms Avatar in memory synchronization. As with Avatar, the actual memory transfer is issued on demand at page granularity. All memory modifications are kept locally and never forwarded back to QEMU. By design, `Laelaps` forwards peripheral inputs to QEMU and lets QEMU re-execute the explored path. Therefore, there is no need to transfer memory back to QEMU.

5.4 Interrupt Injection

`Laelaps` randomly injects activated interrupts to QEMU. This is implemented on top of QEMU Machine Protocol (QMP) interface. We added three new QMP commands: "active-irqs", "inject-irq", and "inject-irq-all". They are able to get the current activated interrupt numbers, inject an interrupt, and inject all the activated interrupt numbers in one go, respectively. QMP is a JSON based protocol. `Laelaps` connects to the QMP port of the QEMU instance and randomly sends QMP commands to inject interrupts. For example, to inject an interrupt with number 10, `Laelaps` sends the following QMP message.

```
{"execute": "inject-irq", "arguments": {"irq": 10}}
```

To assert an interrupt, the added QMP command emulates a hardware interrupt assertion by setting the corresponding bit of the *interrupt status pending register (ISPR)*. It is worth noting that the injected QMP commands can never be executed in QEMU in our initial implementation. It turned out the threads handling QMP commands and I/O cannot be executed concurrently. In particular, QEMU listens for QMP messages and handles I/O in separate threads. Each thread must acquire a global lock by invoking the function `qemu_mutex_lock_iothread()` to grab CPU. We observed that QMP thread can never win in acquiring the lock when I/O thread is actively invoked. In fact, the default `Pthread` mutex does not implement FIFO protocol. Therefore, OS cannot guarantee QMP can ever acquire the lock. We made a workaround by delaying 100μs in each I/O loop.

Table 1: Emulation summary of 30 firmware images.

Board	RTOS	FW # w/o Human Intervention	FW # w/ Human Intervention	Failed #
NXP_FRDM-K66F	FreeRTOS, Bare-metal	14	2	6
NXP_FRDM-KW41Z	FreeRTOS, Bare-metal	3	0	1
STM32100E_EVAL	FreeRTOS, Bare-metal	2	0	0
STM32 Nucleo-L152RE	ChibiOS, Mbed OS	1	1	0

6 EVALUATION

6.1 Firmware Emulation

6.1.1 Firmware Collection. To test how `Laelaps` deals with diverse firmware, we collected/built 30 firmware images from/for four ARM Cortex-M based development boards. They are NXP FRDM-K66F development board, NXP FRDM-KW41Z development board, STMicroelectronics Nucleo-L152RE development board, and STM32100E evaluation board. The reason why we chose development boards is that we could run the firmware on real devices. Therefore, the execution traces captured on real devices (see §6.2) form a ground truth for evaluating the fidelity of firmware execution in `Laelaps`. All the evaluated firmware images were built from the SDKs and demonstration programs provided by the corresponding chip vendors. We note that chip vendors are investing significant resources into the development of SDKs to attract developers. Also, to reduce time to market, more developers are willing to adopt the low-level SDK codes from the vendors.

In terms of software architecture, we tested three popular open-source real-time operating systems (FreeRTOS, Mbed OS, and ChibiOS/RT) as well as bare-metal firmware. In terms of peripheral diversity, these firmware images contain drivers for a large number of different peripherals, ranging from basic sensors to complex network interfaces. Depending on the sophistication of the SDK, the drivers work either in polling mode or interrupt mode. Therefore, the collected images resemble the functionality and complexity of real-world firmware images. We put detailed information about each firmware image in Appendix B.

6.1.2 Results. We tested each of the collected firmware images using `Laelaps`. The result is promising. As shown in Table 1, among all 30 images, `Laelaps` is able to successfully emulate 20 images without any human intervention. All the emulations advance to the core logic of the tasks correctly. At this point, the environment has been properly initialized, allowing for close inspection of interesting code points. For three very complex firmware images

(Column 4), Laelaps is able to emulate them with some human interventions. Among these three images, two of them need data input. We manually redirected the input stream (see §6.3.2).

On the other hand, there exist seven images that Laelaps cannot handle even with human efforts (Column 5). We analyzed the execution traces and attributed these failed emulations to the following reasons. First, sometimes the firmware reads a peripheral register and stores the value in a global variable, but only uses that value after a long time. From time to access to time to use, there could have been multiple switches between symbolic execution engine and concrete execution engine. It is obviously that the peripheral value cannot stay symbolized at the time of use. As a result, symbolic engine cannot execute CPSA algorithm holistically. Second, some firmware depends on custom-made peripherals to implement complex computations such as hash or cryptographic operations, which anger failed to handle. All the details, including the Context_depth and Forward_Depth needed for successful emulations, can be found in Appendix B

6.2 Fidelity Analysis

Although our experiments shows that Laelaps is able to boot a variety of firmware images and reach a point suitable for dynamic analysis, we have no idea as to whether the execution traces in Laelaps resemble ones in real device execution. Therefore, we collected two firmware execution traces of the same firmware image on both Laelaps and real devices, and compared the similarity between them. This firmware simply boots the FreeRTOS kernel and prints out a "hello world" message through the UART interface.

6.2.1 Trace Collection. We collected the firmware execution trace on a real NXP FRDM-K66F development board using the built-in hardware-based trace collection unit called *Embedded Trace Macrocel (ETM)* [2]. ETM is an optional debug component to trace instructions, and it enables the transparent reconstruction of program execution. We directly leveraged the on-board OpenSDA interface to enable the ETM and access the traced data in a buffer called ETB. We do not have the ETM component in Laelaps to collect traces. However, QEMU provides us with great logging facility which allows us to transparently print out execution traces. In particular, we passed the option `"-d exec,nochain"` to QEMU so that it printed out the firmware address before each executed translation block. When mapping the start of each translation block to the firmware code, we can recover the full execution trace.

6.2.2 Execution Trace Comparison. Figure 4 shows a visualized comparison between the traces of the same firmware image collected on Laelaps and real device. We showed the traces collected from system power-on to the start of the first task, corresponding to a full system execution described in §2.2. Figure 4 is a bitmap for the two instruction traces. The top of the figure represents low addresses of the code, while bottom represents high addresses. When an instruction is executed, the corresponding pixel is highlighted. In the figure, the trace collected on Laelaps is in red, and the trace collected on real device is in blue. We observed a large number of overlapped regions labeled in purple, implying that the two traces have similar path coverage. In the figure, we also marked the end of

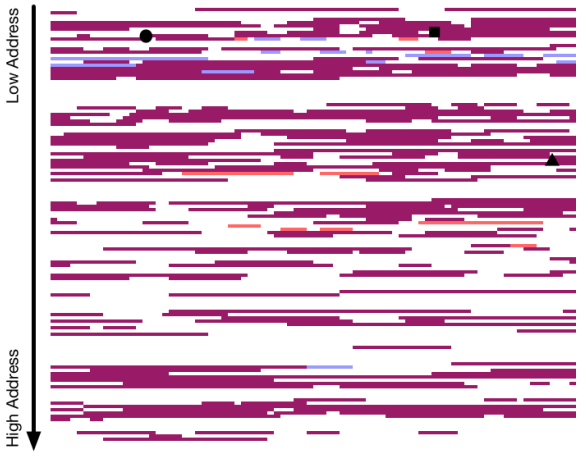

Figure 4: Bitmap of instruction traces collected on Laelaps and the real device. Purple color represents overlapped trace segments. • marks the end of device setup (phase 1). ■ marks the end of base system setup (phase 2). ▲ marks the end of RTOS initialization and the start of the first task (phase 3).

Table 2: Jaccard indexes between the traces collected on Laelaps and real devices when applying only the fall-back path heuristic and all the heuristics.

	FW #1	FW #2	FW #3	FW #4	FW #5	FW #6	FW #7
fall-back path	37.43	39.17	92.96	56.67	48.41	44.02	87.80
all	96.54	92.02	94.26	79.78	95.40	95.42	92.15
	FW #8	FW #9	FW #10	FW #11	FW #12	FW #13	FW #14
fall-back path	35.85	45.34	46.26	51.79	40.54	32.74	45.47
all	96.54	92.02	94.26	79.78	95.40	95.42	92.15

the first three execution phases, which are essential milestones during firmware execution. The figure clearly shows that both traces reach all of them.

Note that having even exactly the same path coverage does not mean the two execution traces are the same. For example, a real device execution may encounter a long loop waiting for a signal, while Laelaps can directly pass through the loop, leading to different execution traces but the same coverage (the same set of control flow transitions occur in both traces). However, many of these deviations are not important. In fact, our emulation does not need to faithfully honor the real execution path in this case.

Coverage similarity measurement visualized in Figure 4 is only an intuitive demonstration of the fidelity achieved by Laelaps. To be able to quantitatively measure the similarity of collected traces, we also calculated Jaccard index (i.e., the number of common instructions between two traces divided by the number of total instructions in the union of the two traces) to measure the common instructions between the collected traces. Since we cannot control the interrupts to be delivered at exactly the same pace on two targets, we did an alignment to the raw traces so that the comparison starts from the same address. In particular, interrupt processing intrusions are extracted and compared separately. Then the results were combined together. Table 2 shows the Jaccard indexes when only applying the fall-back path heuristic and applying all the heuristics. The fall-back path heuristic represents a

Table 3: Corruption detection under different probabilities for corrupting inputs.

	Pc	# of Test-cases	# of Corruptions	Detection Ratio
Test 1	0.10	840	88	10.48%
Test 2	0.05	936	47	5.02%
Test 3	0.01	939	9	0.96%

straightforward path selection strategy that can be developed with reasonable effort, while combining them all is only possible with the proposed sophisticated symbolic guidance. We list the results of 14 firmware images that `Laelaps` is able to emulate without any human interventions. When all the heuristics are applied, the calculated Jaccard indexes are higher than 90% in most cases, which agrees with the visualized result.

6.3 Application to Dynamic Analysis

Based on the positive results we got in firmware emulation, we further explored the possibility of using `Laelaps` to perform actual dynamic analysis.

6.3.1 Fuzzing Mbed OS Firmware. Muench et al. observed that the effectiveness of traditional dynamic testing techniques on embedded devices is greatly jeopardized due to the invisibility of memory bugs on embedded devices [35]. They came up with an idea that leverages six live analysis heuristics to aid fuzzing test in QEMU. These heuristics help make "silent" memory bugs to be easily observable. In their proof-of-concept prototype, they used PANDA [17] which is a dynamic analysis platform built on top of QEMU. Its plug-in system facilitates efficient hooking of various system events. To do the experiments, their approach relied on a real device to initialize the memory and then used Avatar [46] to transfer the initialized state from a real device to PANDA.

To demonstrate `Laelaps`'s device-agnostic property, we ported `Laelaps` to PANDA and tested the same firmware image used in the paper [35]. In addition, we reproduced the same fuzzing experiments. We did not use the real device but were still able to emulate the firmware. After the device was booted, we took a snapshot. During fuzzing, if the device crashed, the fuzzer instructed the emulator to restart from the snapshot.

The firmware image is empowered by the *Mbed OS* and integrates the *Expat* [13] library for parsing incoming XML files. The used Expat library has five types of common memory corruption vulnerabilities. The firmware image took input from the UART interface. As in the paper [35], we instrumented the fuzzer to forcefully generate inputs which trigger one of the five kinds of memory corruption vulnerabilities with a given probability *Pc*. We ran the experiment for 1 hour under probabilities $Pc = 0.1$, $Pc = 0.05$ and $Pc = 0.01$, respectively. The result is shown in Table 3. We can see that there is roughly a linear relationship between *Pc* and detection ratio. Also, the less corrupting inputs were given, the more test-cases could be tested within one hour. This is because the PANDA instance can persist on multiple valid inputs, but it has to take time to restore when receiving malformed inputs. This experiment demonstrates that `Laelaps` is able to do dynamic analysis without relying on a real device.

6.3.2 Analyzing FreeRTOS Firmware. We also tested the capability of `Laelaps` in helping analyze real-world vulnerabilities in FreeRTOS-powered firmware. These vulnerabilities locate in the FreeRTOS+TCP network stack, which were reported in AWS FreeRTOS with version 1.3.1. Without `Laelaps`, the traditional dynamic analysis of these vulnerabilities is very expensive, as it has to rely on real devices and hardware debuggers. We prepared our testing in two steps. First, since the reported vulnerabilities occur in the FreeRTOS+TCP TCP/IP stack, we replaced *lwip*, the default TCP/IP implementation shipped with the SDK of NXP FRDM-K66F, with FreeRTOS+TCP. Second, we identified the location of the network input buffer and wrote a PANDA plugin to redirect the memory read operations from the buffer to a file. We began our testing from the function `prvHandleEthernetPacket`, which is the gateway function processing incoming network packets. In the end, we succeeded in triggering four TCP and IP layer vulnerabilities (CVE-2018-16601, CVE-2018-16603, CVE-2018-16523, and CVE-2018-16524). Note that these vulnerabilities were all caused by improper implementation at IP or TCP/UDP layers. We had not been able to identify vulnerabilities residing at higher levels of network stack because triggering them needs highly structured inputs.

7 RELATED WORK

Several approaches have applied symbolic execution to addressing security problems in firmware [15, 16, 24]. Like `Laelaps`, Inception [15] aims at testing a complete firmware image. It builds an Inception Symbolic Virtual Machine on top of KLEE [7], which symbolically executes LLVM-IR merged from source code, assembly, and binary libraries. To handle peripherals, it either models read from peripheral as unconstrained symbolic values or redirects the read operation to a real device. However, this approach relies on the availabilities of source code to retain semantic information during LLVM merging. FIE [16] modifies KLEE to target a specific kind of device (MSP430). It requires source code and ignores the interactions with peripheral. S^2E is a concolic testing platform based on full system emulation [12]. Combining QEMU and KLEE, S^2E enables symbolic variable tracking across privilege boundary. Mousse proposes selective symbolic execution of programs with untamed Android environment [31]. S^2E, Mousse and `Laelaps` are all concolic execution platforms. However, "selective symbolic execution" in S^2E and Mousse mainly apply to applications that run on top of a standard OS. By contrast, `Laelaps` works on lightweight RTOS and bare-metal systems. We offer a set of peripheral-specific search heuristics to mitigate the path explosion for peripheral emulation.

To be able to execute firmware in an emulated environment, many previous work forwards the peripheral access requests to the real hardware [25, 28, 34, 44, 46]. However, a real device does not always have an interface for exchanging data with the emulator. Furthermore, this approach is not scalable for testing large-scale firmware images because for every firmware image a real device is needed. Instead of relying on real devices, our approach infers proper inputs from peripherals on-the-fly using symbolic execution. Our approach inherits many benefits of a traditional emulator. For example, we can store a snapshot at any time and replay it for repeated analyses.

A very related work [23] to ours was recently proposed by Eric Gustafson et. al. The authors proposed to "learn" the interactions between the original hardware and the firmware from the real hardware. As a result, analysts do not need to program a specific back-end peripheral emulator for every target hardware. This approach achieves similar dynamic analysis capability with ours, however, it still needs the real hardware in the "learning" process. P^{2}IM removes the dependence on real hardware by automatically instantiating the abstract machine model with the firmware-specific information by learning the access pattern of the peripheral [19]. As such, P^{2}IM is oblivious to peripheral designs and generic to firmware implementations. However, this approach restricts values to be returned by status registers in a small subset of all possible values. In particular, P^{2}IM employes explorative execution to find proper responses from status registers. However, a search space of 2^{32} candidates is impracticle for concrete execution. P^{2}IM explictly narrows down the search space by only investigating candidates with a single bit set, meaning that only 32 plus 1 candidates are checked. This fundamentally prevents P^{2}IM from generating correct values for status registers. In `Laelaps`, we use to symbolic execution to directly caculate the expected values for peripheral registers.

Finally, previous work has made tremendous progress in analyzing firmware that relies on an abstraction layer, particularly Linux-based firmware [9, 48]. The high-level idea is to design a generic kernel for all the devices. This approach leverages the abstraction layer offered by the Linux kernel. For example, the WiFi interface can be easily supported by providing a standard emulated Ethernet interface, since the POSIX API is being used. However, for microcontroller firmware, there is no such a unified interface between the tasks and the kernel. Even if some MCU OSs provide hardware abstraction layer (HALs), the ecosystem is severely segmented. Therefore, a database for matching HAL libraries needs to be specifically built [14]. Our work does not rely on any abstraction layers and directly interacts with the previously-unseen hardware. Therefore, our approach can deal with more kinds of firmware.

8 CONCLUSION & FUTURE WORK

We present `Laelaps`, a device-agnostic emulator for ARM microcontroller. The high-level idea is to leverage concolic execution to generate proper peripheral inputs to steer device emulator on the fly. Dynamic symbolic execution is a perfect fit for this task based on our observations and experimental validations. To find a right input, the key is to identify the most promising branch. We designed a path selection algorithm based on a set of generally applicable heuristics. We have implemented this idea on top of QEMU and angr, and have conducted extensive experiments. Of all the collected 30 firmware images from different manufacturers, we found that our prototype can successfully execute 20 of them without any human intervention. We also tested fuzzing testing on top of `Laelaps`. The results showed that `Laelaps` is able to correctly boot the system into an analyzable state. As a result, `Laelaps` can identify both self-injected and real-world bugs. In the future, we plan to extend our prototype to support a border spectrum of devices including ARM Cortex-A and MIPS devices.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. This research was supported in part by NSF CNS-1850434, NSF CNS-1814679, NSF CNS-2019340, ARO W911NF-13-1-0421 (MURI), and JFSG from the University of Georgia Research Foundation, Inc.

REFERENCES

[1] Amazon Web Services. 2018. The FreeRTO Kernel. https://www.freertos.org/.
[2] Arm Holdings. 2015. ARM Cortex-M3 Processor Technical Reference Manual. https://developer.arm.com/docs/100165/0201.
[3] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories.* Springer International Publishing.
[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05).*
[5] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *ICSE'13.*
[6] Derek Bruening, Chris Adeniyi-Jones, Edmund Grimley-Evans, and Kevin Zhou. 2017. Building Dynamic Tools with DynamoRIO on x86 and ARMv8. 2017 International Symposium on Code Generation and Optimization Tutorial.
[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08).*
[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS'06).*
[9] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS'16).*
[10] Vitaly Chipounov and George Candea. 2010. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10).*
[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *SIGPLAN Not.* 47, 4 (March 2011), 265–278. https://doi.org/10.1145/2248487.1950396
[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S^2E: A Platform for In-Vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*
[13] Clark, James. 2018. Expat XML parser. https://libexpat.github.io/.
[14] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Giovanni Vigna Christopher Kruegel, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Boston, MA.
[15] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18).*
[16] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13).* USENIX, Washington, D.C., 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
[17] 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 (PPREW-5).*
[18] Federal Communications Commission. [n.d.]. FCC ID Search. https://www.fcc.gov/oet/ea/fccid.
[19] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of Usenix Security Symposium.*
[20] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 2007 International Conference in Computer Aided Verification (CAV'07).*
[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI'05.*
[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS'08.*
[23] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Proceedings of the 22nd International*

Symposium on Research in Attacks, Intrusions and Defenses (RAID'19).
[24] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. 2017. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2245–2262. https://doi.org/10.1145/3133956.3134050
[25] 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 (AsiaCCS'14)*. ACM.
[26] Ori Karliner. 2018. FreeRTOS TCP/IP Stack Vulnerabilities – The Details. https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/.
[27] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.
[28] 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)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher
[29] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing Closed-source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*.
[30] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*.
[31] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. 2020. Mousse: a system for selective symbolic execution of programs with untamed environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.
[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*.
[33] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*.
[34] Marius Muench, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research*.
[35] 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 2018 Network and Distributed System Security Symposium (NDSS'18)*.
[36] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*.
[37] Joshua Pereyda. 2016. boofuzz: Network Protocol Fuzzing for Humans. https://github.com/jtpereyda/boofuzz.
[38] NGUYEN Anh Quynh and DANG Hoang Vu. 2015. Unicorn: Next Generation CPU Emulator Framework. Black Hat USA.
[39] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers Without Devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.
[40] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13) (ESEC/FSE-13)*.
[41] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.
[42] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*.
[43] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[44] 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 *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*.
[45] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* 44, 1 (January 2018).
[46] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*.
[47] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. S2PF: Speculative Symbolic PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (Nov. 2012).
[48] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*.

A IMPLEMENATION MISCELLANEOUS

In this section, we provide supplementary information about implementation details for interested readers.

A.1 Precise PC in QEMU

When transferring processor state from QEMU to angr, we found that the PC register always points to the start of the current translated block, instead of the real PC. We borrow the code from PANDA [17] to address this problem. In particular, we injected into the intermediate language some instructions so that the PC can be updated together with each translated guest instruction.

A.2 Extending Interrupt in QEMU

The official QEMU supports 16 system exceptions and 64 hardware interrupts. A real device often uses more interrupts. Therefore, we extended the supported number of interrupt to 140 in our prototype.

A.3 Bit-banding

Bit-banding is an optional feature in many ARM-based microcontrollers [2] . It maps a complete word of memory onto a single bit in the corresponding bit-banding region. Writing to a word sets or clears the corresponding bit in the bit-banding region. Therefore, it enables efficient atomic access of a bit in memory. In particular, a read-modify-write sequence can be replaced by a single write operation. QEMU has already perfectly supported this feature while angr has not. We extended the memory model of angr to honor the defined behavior when writing to a bit-band region. This augmentation has been used by `Laelaps` to successfully emulate STM32 devices in our experiments.

A.4 CBZ/CBNZ Instruction

A CBZ instruction causes a branch if the operand is zero, while CBNZ does the opposite. By definition, these instructions mark the end of basic blocks because they branch to new basic blocks. However, in the default implementation of angr, due to optimization, they are not treated as basic block terminators. In fact, angr uses a basic block variant called *IRSB (Intermediate Representation Super-Block)* which can have multiple exits. This results in abnormal behaviors when `Laelaps` selects a branch. Fortunately, angr provides a configuration option that enables using strict basic blocks. Therefore, we enable this option throughout the use of angr.

A.5 Memory Alias

Some STM32 boards heavily depend on memory alias during booting. We extended the memory model of angr to redirect memory accesses when encountering memory regions configured to be an alias to others.

B DETATILS OF COLLECTED FIRMWARE IMAGES

We briefly describe the functionality of each firmware image. Also, we summarize the details of execution results of these firmware images in Table 4.

(1) It sets up the RTC hardware block to trigger an alarm after a user specified time period. The test will set the current date and time to a predefined value. The alarm will be set with reference to this predefined date and time.
(2) User should indicate a channel to provide a voltage signal (can be controlled by user) as the ADC16's sample input. When running the project, typing any key into debug console would trigger the conversion. The execution would check the conversion completed flag in loop until the flag is asserted, which means the conversion is completed. Then read the conversion result value and print it to debug console.
(3) It uses the systick interrupt to realize the function of timing delay. The example takes turns to shine the LED.
(4) It uses notification mechanism and prints the power mode menu through the debug console, where the user can set the MCU to a specific power mode. The user can also set the wakeup source by following the debug console prompts.
(5) It shows how to use DAC module simply as the general DAC converter.
(6) It sets up the PIT hardware block to trigger a periodic interrupt every 1 second. When the PIT interrupt is triggered a message a printed on the UART terminal and an LED is toggled on the board.
(7) In the example, you can send characters to the console back and they will be printed out onto console instantly using lpuart.
(8) The TPM project is a demonstration program of generating a combined PWM signal by the SDK TPM driver.
(9) User should indicate an input channel to capture a voltage signal (can be controlled by user) as the CMP's positive channel input. On the negative side, the internal 6-bit DAC is used to generate the fixed voltage about half value of reference voltage.
(10) EWM counter is continuously refreshed until button is pressed. Once the button is pressed, EWM counter will expire and interrupt will be generated. After the first pressing, another interrupt can be triggered by pressing button again.
(11) Quick test is first implemented to test the wdog. And then after 10 times of refreshing the watchdog in None-window mode, a timeout reset is generated.
(12) The CMT is worked as Time mode and used to modulation 11 bit numbers of data. The CMT is configured to generate a 40000hz carrier generator signal through a modulator gate configured with different mark/space time period to represent bit 1 and bit 0.
(13) It sets up the FTM hardware block to trigger an interrupt every 1 millisecond. When the FTM interrupt is triggered a message a printed on the UART terminal.
(14) It sets up the LPTMR hardware block to trigger a periodic interrupt after every 1 second. When the LPTMR interrupt is triggered a message a printed on the UART terminal and an LED is toggled on the board.
(15) The example configures one FlexCAN Message Buffer to Rx Message Buffer and the other FlexCAN message buffer to Tx message buffer with same ID. After that, the example will send a CAN message from the Tx message buffer to the Rx message buffer through internal loopback interconnect and print out the Message payload to terminal.
(16) It uses the RNGA to generate 32-bit random numbers and prints them to the terminal.
(17) It excuates one shot transfer from source buffer to destination buffer using the SDK EDMA drivers.
(18) One sai instance records the audio data from input and playbacks the audio data.
(19) It uses the KSDK software to generate checksums for an ASCII string.
(20) The SYSMPU example defines protected/unprotected memory region for the core access and tested whether memory protection interrupt events can be delivered if memory violations are detected.
(21) The ENET example tests FreeRTOS+TCP network stack.
(22) This example introduces simple logging mechanism based on message passing.
(23) It prints the "Hello World" string to the terminal using the SDK UART drivers.
(24) The RTC demo application demonstrates the important features of the RTC Module by using the RTC Peripheral Driver. It tested the calendar, alarm and seconds interrupt.
(25) The bubble application demonstrates basic usage of the onboard accelerometer to implement a bubble level. It uses the FTM/TPM to modulate the duty cycle of green and blue colors of onboard RGB LED to gradually increase intensity of the colors as the board deviates from a level state.
(26) It is a simple demonstration program that uses the SDK UART driver in combination with FreeRTOS.
(27) It outputs the printf message on the Hyperterminal using USARTx.
(28) It coordinates two tasks with the help of semaphore in FreeRTOS.
(29) It flashes the board LED using a thread, by pressing the button located on the board and output a string on the serial port SD2 (USART2).
(30) It is the same image used in paper [35]. It reads XML files from UART and uses expat to parse them.

Table 4: Details of Firmware Samples

	RTOS	#	Peripheral	Success	Need Human Intervention	Minimal Context_Depth	Minimal Forward_Depth	Image Size (KB)
NXP_FRDM-K66F	Bare-metal	1	RTC [1]	Y	N	2	5	20.0
		2	ADC [2]	Y	N	1	4	17.0
		3	GPIO [3]	Y	N	2	3	5.1
		4	SMC [4]	Y	N	4	5	26.0
		5	DAC [5]	Y	N	3	4	17.0
		6	PIT [6]	Y	N	1	3	18.0
		7	LPUART [7]	Y	N	1	2	12.0
		8	TPM [8]	Y	N	1	3	20.0
		9	CMP [9]	Y	N	2	3	17.0
		10	EWM [10]	Y	Y	2	3	17.0
		11	WDOG [11]	Y	N	3	5	17.0
		12	CMT [12]	Y	N	2	3	18.0
		13	FTM [13]	Y	N	2	3	19.0
		14	LPTMR [14]	Y	N	2	3	18.0
		15	FLEXCAN [15]	N	–	–	–	22.0
		16	RNGA [16]	N	–	–	–	16.0
		17	EDMA [17]	N	–	–	–	24.0
		18	SAI [18]	N	–	–	–	38.0
		19	CRC [19]	N	–	–	–	17.0
		20	MPU [20]	N	–	–	–	19.0
	FreeRTOS	21	ENET [21]	Y	Y	2	3	65.0
		22	UART [22]	Y	N	1	3	29.0
NXP_FRDM-KW41Z	Bare-metal	23	UART	Y	N	1	2	8.7
		24	RTC	Y	N	1	3	22.0
		25	I2C [23]	N	–	–	–	22.0
	FreeRTOS	26	UART	Y	N	1	3	22.0
STM32100E_EVAL	Bare-metal	27	USART [24]	Y	N	1	2	4.5
	FreeRTOS	28	USART	Y	N	1	2	13.0
STM32 Nucleo-L152RE	ChibiOS	29	USART	Y	N	4	4	5.5
	Mbed OS	30	UART	Y	N	5	6	92.0

1. Real Time Clock
2. Analog-to-Digital Converter
3. General-Purpose Input/Output
4. System Mode Controller
5. Digital-to-Analog Converter
6. Periodic Interrupt Timer
7. Low Power Universal Asynchronous Receiver/Transmitter
8. Timer/PWM Module
9. Comparator
10. External Watchdog Monitor
11. Watchdog Timer
12. Carrier Modulator Transmitter
13. FlexTimer Module
14. Low-Power Timer
15. 10/100-Mbps Ethernet MAC
16. Random Number Generator Accelerator
17. Enhanced Direct Memory Access
18. Synchronous Audio Interface
19. Cyclic Redundancy Check
20. Memory Protection Unit
21. Flexible Controller Area Network
22. Universal Asynchronous Receiver/Transmitter
23. Inter-Integrated Circuit
24. Universal Synchronous/Asynchronous Receiver/Transmitter