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

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 IoT devices, especially microcontrollers. These security concerns have significantly hamstrung further evolution of IoT technology. In this work, we present Laelaps, a device emulator specifically designed to run diverse software of low-cost IoT 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 deployed two popular dynamic analysis techniques—fuzzing testing and dynamic symbolic execution—on top of our emulator. We successfully identified both self-injected and real-world vulnerabilities.

1. INTRODUCTION
Software-based emulation techniques \cite{1} 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 \cite{2,3,5}. However, none of these capabilities have been utilized to analyze firmware of low-end embedded devices such as microcontrollers. A major obstacle of utilizing existing analysis capabilities is the absence of a versatile device-agnostic software-based emulator that could execute arbitrary firmware of different devices.

To implement such an emulator, it has to deal with the vast diversity of microcontroller firmware in terms of hardware architecture (e.g., x86, ARM, MIPS, etc), integrated feature (e.g., network function, DSP, etc), and the underlying operating system (e.g., bare-metal, Linux, FreeRTOS, etc.). Customizing the emulator for every kind of equipment is nearly impossible. The vast diversity design dates back to the System-on-Chip (SoC) design methodology of embedded systems, where a single integrated circuit (IC) integrates all components of a computer, including processor, memory, and other hardware logics that interface with a bunch of peripherals. Furthermore, in order to make their products most competitive, manufacturers tend to integrate more and more custom-made functions in peripherals. For example, the NXP FRDM-K66F chip incorporates more than 50 different peripherals \cite{6}. If we zoom in a very simple peripheral – the Universal Asynchronous Receiver/Transmitter (UART) interface, it is controlled by 40 individual registers, let alone other complex peripherals such as Network Interface Controller (NIC).

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, designed for a specific operating system, or they must be tightly coupled with real devices. Implementing a generic emulator that deals with different peripherals has to put tremendous efforts. Therefore, existing work \cite{7,8,9,10,11} 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. A recent work advances this research direction by modeling the interactions between the original hardware and the firmware \cite{12}. This enables the virtualized execution of any piece of firmware possible without writing a specific back-end peripheral emulator for the hardware. However, this approach still requires the real hardware to “learn” the peripheral interaction model. Previous work also explores ways to emulate Linux-based firmware \cite{13,14}. FIRMADYNE \cite{13} extracts the file system of the firmware and mounts it with a generic...
kernel executed in QEMU [15]. FIRM-AFL [14] further proposes a grey-box fuzzing mechanism. However, both of them only work for Linux-based embedded firmware, while a large number of real-world embedded systems run on microcontrollers which only support lightweight RTOS or bare-metal systems.

In this work, we demonstrate that the obstacles of device-agnostic firmware execution are not insurmountable. We present Laelaps\footnote{"Laelaps" was a Greek mythological dog who never failed to hunt his prey. So, in a metaphorical sense, here we use this term to represent the potential versatility of our proposed solution.} as 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)\footnote{Laelaps is open source at (URL omitted for double-blind reviewing). We also release the corresponding} 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 [15], 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 unknown 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\footnote{In general, Laelaps’s concrete execution will be stuck when accessing an unknown 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 [17,19]. However, peripherals reveal many distinct features that require special treatment, such as very common infinite loops and interrupts. 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 designed logic, we demonstrate that it is able to properly initialize the execution context for dynamically analyzing the firmware in practice.}

We have developed Laelaps on top of angr [20] and QEMU [15]. 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 FreeRTOS [21]. We admit that a small portion of peripheral read operations (e.g., receiving network packets) still need necessary human inputs, nevertheless Laelaps takes a step towards scalable, dynamic IoT firmware analysis. It enables existing dynamic analysis techniques to become directly applicable to analyzing embedded firmware. In particular, 1) our tool makes firmware fuzzing more efficient; this is because after instrumenting the QEMU with heuristics about symptoms of crashes, firmware corruptions can be easily captured [22]; 2) our tool assists dynamic symbolic execution; with a properly initialized context, symbolic execution can be constrained and thus avoids symbolizing too many variables; 3) analysts can interactively debug the firmware using debuggers such as GDB; 4) the fully initialized device state can be saved as a snapshot for future replayable analysis. Note that we complete all of these dynamic analysis tasks without purchasing any real device or using any proprietary emulators that specifically work for certain devices. In summary, 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 unknown peripherals.
- We demonstrate the potential of Laelaps by using it in combination with advanced dynamic analysis tools, including boofuzz [23], angr [20], and PANDA [24]. We show that the full emulation environment provided by Laelaps facilitates the execution of these advanced (forms of) dynamic analysis, which enables us to identify both self-injected and real-world bugs.

Laelaps is open source at (URL omitted for double-blind reviewing). We also release the corresponding
demonstration firmware samples analyzed in our experiments.

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. More than 22 billion units of Cortex-M based devices have been shipped [25]. Cortex-M family devices include a wide range of cores including Cortex-M0, Cortex-M0+, Cortex-M3, and Cortex-M4, which build on one another with more features. Cortex-M cores are based on the 32-bit ARMv6-M, ARMv7-M or ARMv8-M architectures. All of them support Thumb instructions 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 operating system code are mingled together in a flat memory address space. For this reason, it does not support the popular Linux kernel. Instead, around it, many other ecosystems have been developed. Examples include Amazon FreeRTOS [21] and Arm Mbed OS [26].

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 (called 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 manufacturers are free to customize their implementation as long as it conforms to the design defined by Arm. As a result, different manufacturers optimize and customize their products in different ways, leading to a vast diversity of Cortex-M processors.

2.2 Firmware Execution

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 execution. In other words, the executed driver version, as long as it is valid, does not even 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 [28, 29], 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 [30, 32]. Dynamic symbolic execution (a.k.a concolic execution) [33, 36] 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 [37, 39]. The core of Laelaps is a concolic execution approach for peripheral emulation. One particular challenge for concrete execution is the path explosion problem [17, 19]. Our study proposes a set of peripheral-specific search heuristics to mitigate the path explosion, and they work well in practice.

3. OVERVIEW

3.1 Function Gap

QEMU [15], the most popular generic machine emulator, has built-in support for almost all of the functions defined by Arm. We call them core peripherals/func-
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 indefinitely. Our in-depth study on the usage of peripheral values leads to three key observations.

First, most peripheral accesses are in fact not critical to 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.

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

Second, excluding the non-critical peripheral accesses, many of the rest are involved in firmware control flow logic so that they have direct influence on the execution path. Third, if we can find a value that drives the execution along a correct path, then QEMU can usually execute the firmware as expected.

```c
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)){}  //wait
4         base->D = *(data++);
5     }
6 }
```

Listing 1: Code snippet from real driver code.

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. To demonstrate this, we manually followed the execution of a simple SDK sample that prints out a “hello world” message via UART interface, and counted the usage of each peripheral read operation. In total, there are 134 read operations to custom-made peripherals. Among them, 104 (77.6%) are not critical, and 30 (22.4%) directly affect the control flow. Therefore, if we can correctly infer a correct path, we have a high chance to succeed in emulating this firmware execution.

Admittedly, there are still a small portion of peripheral read operations that are not amenable to symbolic execution. For example, Ethernet driver fetches Internet packets from another endpoint. Obviously, randomly generated data from symbolic execution may not obey the program logic. Another example is peripheral-assisted cryptographic computations. We will discuss appropriate mitigations in §4.5.

3.3 Laelaps Overview
Laelaps combines concrete execution and symbolic execution, namely concolic execution \(33-36\). Neither of them alone could achieve our goal because 1) concrete execution cannot deal with unknown 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 unknown 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 Cortex-M0/3/4) and locations of ROM and RAM. Then it loads ARM core peripherals into the system map. All the other memory regions are marked unimplemented, and the accesses to them are intercepted. QEMU translates and emulates each instruction of firmware until there is a read operation to an unknown memory. Our goal is to predict a proper read values. 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 unknown read operation is detected, the processor context and memory are then synchronized to the symbolic execution engine (S1). During symbolic execution, every unknown 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 the 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
- **E5**: User defined program points

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 (\(4.2.6\)). 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. Lastly, for E5, assuming analysts have some prior knowledge about the firmware via static analysis or Laelaps itself, we provide an interface allowing them to configure some program points that should terminate symbolical execution.

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. 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., passing the device setup phase). It leaves to analysts to decide the right time to dig into firmware execution and perform further analysis.

### 4. LAELAPS SYSTEM DESIGN

This section details the design of core components of Laelaps. This includes the state transfer module, the symbolic execution module, and the path selection algorithm. We also discuss how we emulate device interrupts to move execution forward. Finally, we discuss limitations of our design and the mitigation in practice.

#### 4.1 State Transfer
Whenever an unknown peripheral read is detected, the program state is transferred to our 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. During symbolic execution, QEMU is suspended, and symbolic execution engine works on its own RAM copy.

## 4.2 Symbolic Execution

### 4.2.1 Basic Rule #1

Since the symbolic execution engine is invoked by unknown 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. In this sense, we assign new symbols spatially (different addresses get different symbols) and temporally (different times get different symbols). Consider an example shown in Listing 2. `base->CNT` is a peripheral register that keeps an increasing counter. Line 1 and line 3 read the current values. Although they are accessing the same memory address, since they are accessed at different times, we assign two different symbols. Otherwise, line 5 can never be reached (subtracting a symbol from itself always gets zero).

1. `cnt0 = (uint32_t)(base->CNT & FTM_CNT_COUNT_MASK);`
2. `do_stuff();`
3. `cnt1 = (uint32_t)(base->CNT & FTM_CNT_COUNT_MASK);`
4. `if((cnt1 - cnt0) > 0xFF)`
5. `...`

Listing 2: Source code snippet using peripheral timer.

### 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. Unfortunately, QEMU has full-fledged emulation capability. Any exception could cause a deviation from the expected path and thus render 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 for concrete execution.

### 4.2.5 Solver

§3 has discussed several events that indicate the execution will go back to QEMU. When one of such predefined events arrives, Laelaps has accumulated a list of symbolic formulas during symbolic execution. Laelaps utilizes an SMT solver to find concrete inputs to these symbols and steers QEMU’s concrete execution towards the selected path (§4.3).

### 4.2.6 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`, `memset`, and `strcpy`. Since those functions usually contain long loops, symbolically executing them is extremely inefficient. Lae-
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. Since the left-hand branch has the most promising path, we choose the left-hand branch. Similarly, at the node 0x800, the right-hand branch is selected. CPSA selects the most promising path on each branching. It avoids paths with infinite loops. It also avoiding re-executing old paths. See our heuristics for more details.

Laelaps is able to automatically detect long loops. 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 (say 5), symbolic execution will be terminated.

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. Since we lack the high-level semantic information about data structures and control flow, it is particularly challenging to choose the right branch. In the following, we start with an overview of our path selection strategy – Context Preserving Scanning Algorithm, or CPSA for short. Then we interpret a representative SDK code snippet. It intuitively explains our main search heuristics to prioritize a “right” branch.

Figure 3 shows how CPSA works in general. 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 [20]. 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. We apply an algorithm to selecting 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.

4.3.1 Explanatory Example

Listing 3 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 actually 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 indicated in lines 16 and 18. Different from PHY_Write in lines 10 and 12, there are multiple chances indicated 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). Obviously, 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”. In what follows, we present our heuristics that automatically guide the execution to this right path.

```c
static void enet_init(...){
    ...
    status = PHY_Init(...);
    if (kStatus_Success != status) {
        LWIP_ASSERT("\n\nCannot initialize PHY.\r\n", 0);
        LWIP_ASSERT("\n\nThe program is not initialized, trying again.
\r\n", 0);
    }
    status_t PHY_Init(...){
        ...
        result = PHY_Write(...);
        if (result == kStatus_Success) {
            result = PHY_Read(..., kbsReg);
            if (result == kStatus_Success) {
                PHY_Write(..., ctlReg);
                PHY_Read(..., ctlReg);
            }
        }
    }
    ...
}```
Listing 3: Source code of a complex Ethernet driver (some parameters are omitted due to space limit).

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 3 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 [40], 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 3, 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 basic blocks 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, we do not apply sophisticated fixed-point theorems [41]. Instead, our symbolic engine maintains the execution traces and states of explored paths, and it compares execution states within each path. If any two states are the same, we regard this path as an infinite loop. To determine whether the two execution states are the same, we only consider registers (including program counter) with concrete values. This design choice is proven quite efficient and accurate in our evaluation. In contrast, a finite loop must have a concrete counter register that controls loop termination condition, which will be changed in each iteration.

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 chooses a path with infinite loop at the lowest priority. When 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 path. 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 3, 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 booting code
of firmware typically initializes each peripheral one by one. Therefore, our algorithm tends to move forward quickly.

However, the fall-back path heuristic may not yield an optimal result in choosing right branches regarding exception handler decision. Consider the following code snippet of a UART interrupt handler.

```c
1 if ((UART_S1_FE_MASK & base->S1)) //RX framing error
2 ...
3 if ((UART_S1_PF_MASK & base->S1)) //RX parity error
4 ...
5 if ((UART_S1_RDRF_MASK & base->S1) && ...) 
6 //Receive data register full
7 ...
8 if ((base->S1 & UART_S1_TDRE_MASK) && ...) 
9 //Send data register empty and interrupt is enabled
```

Due to limited resource, an interrupt number is often shared by many related events. The handler checks the occurrence of each possible event and handles it if the corresponding event really happens. Each if statement often reads a value from the peripheral. If our algorithm must choose a fall-back branch at this conditional statement, based on the above discussion, our algorithm skips the statements in the if block. This is because the next if statement typically has a higher address than the statements inside the previous if block (e.g., line 3 has a higher address than line 2). As a result, none of these events can be handled. To address this issue, if Laelaps detects that the execution is in the context of an exception, we change our fall-back path to be the one with the lowest address. Of course this design could lead to additional code being executed. However, based on your evaluation, interrupt handlers can often gracefully deal with unexpected events. On the other hand, if there does come this event, our design can make the execution move forward.

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.

### 4.3.6 Summary

To sum up, CPSA preserves multiple symbols across Context_Depth branches. For each branch, within Forward_Depth steps, CPSA goes through three bases trying to single out a branch. Specifically, CPSA first eliminates the paths with obvious infinite loops. Then it gives a score to the rest paths based on the similarity with previously recorded paths. Our search strategy favors new paths. If there are still multiple choices, a fall-back path is selected depending on the execution context.

### 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

As discussed in §3.2, Laelaps works well when the peripheral inputs only decide control flow. However, the firmware also interacts with the external world by data exchange. From simple UART channels to complex Ethernet channels, they are typically implemented by fetching data from a particular data register at the agreed time slots. Obviously, we cannot feed the randomly generated data 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 §4.4, we show how we use Laelaps to hook network functions in FreeRTOS and analyze the TCP/IP stack of FreeRTOS to reproduce the vulnerabilities disclosed by Zimperium zLabs in Dec 2018 [42].

#### 4.5.2 Cryptographic Computation

To speed up cryptographic computation, many embedded chips are integrated with a dedicated hardware-accelerated unit. Cryptographic peripherals not only generate data for firmware but also influence critical program logic. For example, an encrypted buffer from the Internet should be deciphered first. Then, the packet is parsed byte by byte, during which multiple branches could be encountered. Although significant progress has been made in simulating cryptographic primitives [43], no practical solution can be easily integrated with Lae-
Laelaps, which hamstrings the capability of Laelaps.

4.5.3 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. In our evaluation, we demonstrate that with necessary human inputs, Laelaps succeeds in dynamically running very complex firmware images.

4.5.4 Mitigation #1: Infinite Loop Heads-up

The CPSA algorithm scans all the paths starting from each branch and eliminates paths with infinite loops. In a program, some paths are bound to run into an infinite loop. If the analysts can provide these information, the symbolic engine can just avoid diving into potential infinite loops from the very beginning. For example, firmware usually provides an assertion mechanism to detect its own defects or for debugging purpose. It offers handy information for our symbolic execution module — an assertion error is not the expected path. If the execution goes to a failed assertion, there will most likely be an infinite loop. Therefore, we provide an interface for analysts to provide entry points for such assertion-failure functions, and the CPSA algorithm will avoid these paths in the first place.

4.5.5 Mitigation #2: Decision Overriding

When the symbolic engine makes a wrong decision in path selection, we provide an interface for analysts to override the result. In particular, analysts can provide branching locations (in the form of addresses), the corresponding processor context (register values), and the expected branch. If a match is found, the result of branch selection drawn from Laelaps is overridden by the provided one. In addition to the coarse-grained branch selection overriding, the analysts can further rewrite the concrete values to be provided with QEMU.

5. IMPLEMENTATION

We developed the prototype of Laelaps based on QEMU [15] and angr [20], which are concrete execution engine and symbolic execution engine, respectively. To facilitate state transfer between the two execution engines, we integrate Avatar [7][8], a Python framework for seamlessly orchestrating multiple dynamic analysis platforms, including QEMU, real device, angr, PANDA [24], 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 unknown 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) 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 obtained from the official product description page, third-party forums, or the Federal Communications Commission (FCC) ID webpage [14]. 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) [15]. On the other hand, some firmware relies on another piece of code (bootloader), in which case additional analysis is required.

5.2 Peripheral Access Interception

When firmware accesses an unknown address (i.e., not a part of ROM/RAM or ARM-defined core peripherals), 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, which we explain in the next section. We made modifications to Avatar so that additional Cortex-M specific registers (e.g., Program Status Register (PSR)) are synchronized to angr,
and we implemented our own memory synchronization interface as well.

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 unknown 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 unknown 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 let 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.

Due to the space limit, we put many other implementation details in Appendix A for interested readers.

6. EVALUATION

We conducted empirical evaluations to demonstrate how Laelaps enables device-agnostic firmware emulation and how such capability benefits firmware analysis.

6.1 Firmware Emulation

6.1.1 Firmware Collection

To test how Laelaps deals with diverse firmware, we collected/built 30 firmware images from 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 STMicroelectronics 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. In other words, by comparing the traces collected on real devices and emulator, we were able to measure how Laelaps deviates from a “perfect” emulator that faithfully implements all the peripherals.

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. FreeRTOS is a market leader in the IoT and embedded platforms market, being ported to over 40 hardware platforms over the last 14 years. Mbed OS is the official embedded OS for ARM Cortex-M based IoT devices. ChibiOS/RT is another compact and efficient RTOS supporting multiple architectures, especially for STM32 devices. In addition, when the SDK includes bare-metal demonstrations, we also tested Laelaps with bare-metal compilations.

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. Each SDK sample was designed to test one type of peripheral, although sometimes multiple peripherals were used in a sample. Depending on the sophistication of the SDK, the drivers work either in polling mode or interrupt mode.

6.1.2 Results
Table 1: Emulation summary of 30 firmware images.

<table>
<thead>
<tr>
<th>Firmware Image</th>
<th>RTOS</th>
<th>FW ≠ w/ Human Intervention</th>
<th>FW ≠ w/ Human Intervention</th>
<th>Failed #</th>
</tr>
</thead>
<tbody>
<tr>
<td>XE2FRDM-K66F</td>
<td>FreeRTOS, Bare-metal</td>
<td>24 (64.9%)</td>
<td>4 (1.1%)</td>
<td>16m07s</td>
</tr>
<tr>
<td>XE2FRDM-K61Z</td>
<td>FreeRTOS, Bare-metal</td>
<td>37 (98.3%)</td>
<td>1 (1.1%)</td>
<td>17m26s</td>
</tr>
<tr>
<td>NXP FRDM-K66F</td>
<td>FreeRTOS, Bare-metal</td>
<td>3 (10.0%)</td>
<td>2 (6.7%)</td>
<td>18m11s</td>
</tr>
<tr>
<td>NXP FRDM-K61Z</td>
<td>FreeRTOS, Bare-metal</td>
<td>3 (10.0%)</td>
<td>2 (6.7%)</td>
<td>18m11s</td>
</tr>
<tr>
<td>STM32D103QVAL</td>
<td>FreeRTOS, Bare-metal</td>
<td>3 (10.0%)</td>
<td>2 (6.7%)</td>
<td>18m11s</td>
</tr>
<tr>
<td>STM32L152RE</td>
<td>ChibiOS, Mbed OS</td>
<td>3 (10.0%)</td>
<td>2 (6.7%)</td>
<td>18m11s</td>
</tr>
</tbody>
</table>

Table 2: Statistics on the three bases to choose branches.

<table>
<thead>
<tr>
<th>Forward Depth</th>
<th>Total Branches</th>
<th>Infinite Loop Elimination</th>
<th>Similarity</th>
<th>Fall-back</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FW #12</td>
<td>3</td>
<td>36</td>
<td>6 (16.7%)</td>
<td>7 (19.4%)</td>
<td>16m07s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>37</td>
<td>6 (16.2%)</td>
<td>7 (18.9%)</td>
<td>18m11s</td>
</tr>
<tr>
<td>FW #22</td>
<td>3</td>
<td>84</td>
<td>78 (92.6%)</td>
<td>7 (8.0%)</td>
<td>17m26s</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>87</td>
<td>79 (90.5%)</td>
<td>7 (8.4%)</td>
<td>20m21s</td>
</tr>
</tbody>
</table>

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. We defined a successful execution to be the one that advances to a point where the environment has been properly initialized, and the core task are running correctly. At this point, analysts can perform actual examination to the execution state. 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, as demonstrated in 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 checksum, hash, or cryptographic operations, which anger failed to handle.

Due to space limit, we put detailed information about each firmware image in Appendix 2. In the table, we list the main peripheral tested by each image. Note that each SDK sample was designed to test one type of peripheral, although sometimes multiple peripherals were used in a sample. We also show the minimal Context_depth and Forward_Depth needed for successful emulations.

6.1.3 Dissecting the CPSA Algorithm

As mentioned in 3.1.3, the CPSA algorithm selects an optimal branch by going through three heuristic rules. They are infinite-loop-elimination, similarity, and fall-back. If anyone of them can determine a single path, the rest of steps are skipped. To show how each rule influences the decision making, we counted the number of each rule that uniquely determined a branch. We also tuned Forward_Depth, which influences the capability to foresee an infinite loop. In Table 2, we show the results of the two most significant cases (firmware images #12 and #22) that all of these heuristic rules work. The descriptions for the firmware images can be found in Appendix 2. As shown in the table, the proportion of each rule highly depends on the firmware image and the value of Forward_Depth. As Forward_Depth increases, we also observed an increase in execution time, meaning that more time is spent on inefficient symbolic execution. Note that although the time required to complete a firmware execution appears to be long, we argue that we can save the fully booted instant as a snapshot and perform analyses based on the snapshots at any time.

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 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) [45]. 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. The traced data include rich information about each execution including timestamps, instruction addresses, instructions being executed, etc. 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,nowchain” to QEMU so that it printed out the firmware address before each executed translation block (a translation block is a basic block variant used in QEMU). When mapping the address of first instruction of each translation block to the firmware code, we can recover the complete 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 the first three execution phases, which are essential milestones during firmware execution. Reaching these points indicates the successful execution at these stages of 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 paths but the same coverage. 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 3 shows the Jaccard index when the three heuristics are applied cumulatively.

Table 3: Jaccard indexes between the traces collected on Laelaps and real devices when the following three heuristics are applied cumulatively: 1) avoiding infinite loop; 2) prioritizing new paths; 3) fall-back path.

<table>
<thead>
<tr>
<th></th>
<th>(1, 2, 3)</th>
<th>(2, 3)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FW #1</td>
<td>94.17%</td>
<td>94.91%</td>
<td>37.43%</td>
</tr>
<tr>
<td>FW #2</td>
<td>90.93%</td>
<td>90.93%</td>
<td>39.17%</td>
</tr>
<tr>
<td>FW #3</td>
<td>92.96%</td>
<td>92.96%</td>
<td>92.96%</td>
</tr>
<tr>
<td>FW #4</td>
<td>95.10%</td>
<td>95.10%</td>
<td>56.67%</td>
</tr>
<tr>
<td>FW #5</td>
<td>94.39%</td>
<td>94.39%</td>
<td>48.41%</td>
</tr>
<tr>
<td>FW #6</td>
<td>93.20%</td>
<td>93.20%</td>
<td>44.02%</td>
</tr>
<tr>
<td>FW #7</td>
<td>87.80%</td>
<td>87.80%</td>
<td>87.80%</td>
</tr>
<tr>
<td>FW #8</td>
<td>96.54%</td>
<td>96.54%</td>
<td>35.85%</td>
</tr>
<tr>
<td>FW #9</td>
<td>92.02%</td>
<td>92.02%</td>
<td>45.34%</td>
</tr>
<tr>
<td>FW #10</td>
<td>94.26%</td>
<td>94.26%</td>
<td>46.26%</td>
</tr>
<tr>
<td>FW #11</td>
<td>79.78%</td>
<td>79.78%</td>
<td>51.79%</td>
</tr>
<tr>
<td>FW #12</td>
<td>95.40%</td>
<td>95.40%</td>
<td>40.54%</td>
</tr>
<tr>
<td>FW #13</td>
<td>95.42%</td>
<td>95.42%</td>
<td>32.74%</td>
</tr>
<tr>
<td>FW #14</td>
<td>92.15%</td>
<td>92.15%</td>
<td>45.47%</td>
</tr>
</tbody>
</table>

We also compared a trace of a FreeRTOS firmware image. The Jaccard indexes were calculated for each of the four bootstrap phases, with the three heuristics applied cumulatively. As shown in Table 4, when all the heuristics were applied, Laelaps achieved a high level of similarity with the real device in all the phases. However, if only heuristic 3 was enabled, the firmware image failed to boot, which is indicated by a low similarity in phase 1 (26.77%) and zero similarity in the following phases.

Table 4: Jaccard indexes of a FreeRTOS trace broken down by the bootstrap phases with the three heuristics applied cumulatively.

<table>
<thead>
<tr>
<th></th>
<th>(1, 2, 3)</th>
<th>(2, 3)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>99.39%</td>
<td>99.39%</td>
<td>26.77%</td>
</tr>
<tr>
<td>Phase 2</td>
<td>96.31%</td>
<td>96.31%</td>
<td>00.00%</td>
</tr>
<tr>
<td>Phase 3</td>
<td>100.00%</td>
<td>91.11%</td>
<td>00.00%</td>
</tr>
<tr>
<td>Phase 4</td>
<td>91.12%</td>
<td>91.12%</td>
<td>00.00%</td>
</tr>
</tbody>
</table>
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%

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 [22]. They came up with an idea that leverages six live analysis heuristics to aid fuzzing test. These heuristics help make “silent” memory bugs to be easily observable. In their proof-of-concept prototype, they used PANDA [24] to implement all these heuristics. PANDA is a dynamic analysis platform built on top of QEMU. Its plug-in system facilitates efficient hooking of various events, such as physical accesses to memory, translation, and execution of translated blocks. To validate these heuristics, their approach relied on a real device to initialize the memory and then used Avatar [2] 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 [22]. In addition, we reproduced the same fuzzing experiments.

Specifically, we configured Laelaps to have the same ROM/RAM configuration as the STM32 Nucleo-L152RE development board and used boofuzz [23] to fuzz the firmware. Note that we reused the UART peripheral emulator used in the original paper [22]. This is because Laelaps cannot handle data input as explained in [15]. Nonetheless, we did not take any other manual efforts for Laelaps to go through other peripheral initialization. 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 [47] 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 [22], we instrumented the fuzzer to forcefully generate inputs which trigger one of the five kinds of memory corruption vulnerabilities with a given probability $P_c$. We ran the experiment for 1 hour under probabilities $P_c = 0.1$, $P_c = 0.05$ and $P_c = 0.01$, respectively. The result is shown in Table 5. We can see that there is roughly a linear relationship between $P_c$ 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 proves that Laelaps is capable of booting firmware to an analyzable state for repeatable 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, including 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 have not been able to identify vulnerabilities residing at higher levels of network stack because triggering them needs highly structured inputs.

6.3.3 Using Laelaps to Constrain Symbolic Execution

Laelaps integrates a symbolic execution engine. As a matter of fact, symbolic execution itself is an effective technique to find software bugs. Combined with the concrete context generated by QEMU, Laelaps can start symbolic execution from interesting points with valid concrete contexts. This directly mitigates the well-known path explosion problem in traditional symbolic execution. Inspired by a Cyber Grand Challenge problem [48], we manually injected a simple stack buffer overflow to the main task of FreeRTOS image. Then we started symbolic execution from the main task. As expected, the symbolic executor encountered an “unconstrained” state in which the instruction pointer can...
be any value, indicating a stack buffer overflow.

7. RELATED WORK

7.1 Firmware Analysis

Several approaches have applied symbolic execution to addressing security problems in firmware [49, 51]. Like Laelaps, Inception [19] aims at testing a complete firmware image. It builds an Inception Symbolic Virtual Machine on top of KLEE [36], 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 availability of source code to retain semantic information during LLVM merging. $S^2E$ is a concolic testing platform based on full system emulation [52]. Combining QEMU and KLEE, $S^2E$ enables symbolic variable tracking across privilege boundary. However, currently it mainly focuses on PC environment without support for ARM. FIE [50] modifies KLEE to target a specific kind of device (MSP430). It requires source code and ignores the interactions with peripheral. FirmUSB [51] analyzes embedded USB devices and uses domain knowledge to speed up the symbolic execution of firmware. Compared to unconstrained symbolic execution, FirmUSB can improve the performance by a factor of seven. To be able to execute firmware in an emulated environment, most of the previous work forwards the peripheral access requests to the real hardware [7, 11]. 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 fetching data from 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 [12] 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. Finally, previous work has made tremendous progress in analyzing Linux-based firmware [13, 14]. The high-level idea is to design a generic kernel for all the devices. This approach leverages the abstract layer offered by the Linux kernel, but cannot work for the firmware of embedded systems where the kernel and tasks are mixed.

7.2 Forced Execution

Emulating binary execution without real input or proper environmental setup has been studied. Forced execution [53] is one of such techniques aiming at discovering different execution paths inside the binary. Researchers also extended this approach for forced execution of JavaScript code [54], mobile binaries [55, 56], and kernel rootkits [57]. This technique has demonstrated its application in control flow graph construction, malicious behavior detection, and API abuse. However, the forced execution of binary focuses on a crash-free execution model, and it rarely reasons and honors the intended control flow. Therefore, the results are often random. This low level of fidelity makes it less appealing when analysts require an authentic execution context.

The resulting analysis result inevitably incurs high false positives because the results may not be reproduced in a real execution context. In contrast, our proposed approach explores multiple paths and prioritizes the most promising one that honors the intended behavior of the binary, thus providing a high fidelity environment. In addition, the existing works on forced execution focus on user-mode execution rather than the whole system execution. How to deal with system-level events such as privileged instructions and interrupts remains unknown.

8. CONCLUSION

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 boot 20 of them without any human intervention. We also tested fuzzing testing and symbolic execution 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. Although our prototype only works for ARM Cortex-M based devices, our design is general. In the future, we plan to extend our prototype to support a border spectrum of devices including ARM Cortex-A and MIPS devices.

9. REFERENCES


the Construction and Analysis of Systems (TACAS’08), 2008.


APPENDIX

.1 Implementation Miscellaneous

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

.1.1 Precise PC in QEMU

In this section, we provide supplementary information about implementation details for interested readers.
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 \[24\] 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.

### 1.2 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.

### 1.3 Bit-banding

Bit-banding is an optional feature in many ARM-based microcontrollers \[45\]. 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 \textit{Laelaps} to successfully emulate STM32 devices in our experiments.

### 1.4 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 \textit{IRSB} (Intermediate Representation Super-Block) which can have multiple exits. This results in abnormal behaviors when \textit{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.

### 1.5 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.

### 2 Details of Collected Firmware Images

Table 6 summarizes details of collected firmware images, including the used peripherals and their full names. We also briefly describe the functionality of each firmware image.

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 watchdog. 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 executes 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 on-board 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.
<table>
<thead>
<tr>
<th>RTOS</th>
<th>Peripheral</th>
<th>Success</th>
<th>Need Human Intervention</th>
<th>Minimal Context_Depth</th>
<th>Minimal Forward_Depth</th>
<th>Image Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NXP_FRDM-K66F</td>
<td>Bare-metal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>RTC</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>ADC</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>GPIO</td>
<td>Y</td>
<td>N</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>SMIC</td>
<td>Y</td>
<td>N</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>LXC</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>PIT</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>LPUART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>TPM</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>CMP</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>EWM</td>
<td>Y</td>
<td>Y</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>WDOG</td>
<td>Y</td>
<td>N</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>CMT</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>FTM</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>LPTMR</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>FLEXCAN</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>RNGA</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>EDMA</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>SAI</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>CRC</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>MPU</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>ENET</td>
<td>Y</td>
<td>Y</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>UART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>NXP_FRDM-KW41Z</td>
<td>Bare-metal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>UART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>RTC</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>T2C</td>
<td>N</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>UART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>STM32100E_EVAL</td>
<td>Bare-metal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>USART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>USART</td>
<td>Y</td>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>STM32 Nucleo-L152RE</td>
<td>Bare-metal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>USART</td>
<td>Y</td>
<td>N</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>UART</td>
<td>Y</td>
<td>N</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

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