Physical Devices-Agnostic Hybrid Fuzzing of IoT Firmware

Lingyun Situ, Chi Zhang, Le Guan, Zhiqiang Zuo, Linzhang Wang, Xuandong Li, Peng Liu, Jin Shi

***Abstract*—With the rapid expansion of the Internet of Things, a vast number of microcontroller-based IoT devices are now susceptible to attacks through the Internet. Vulnerabilities within the firmware are one of the most important attack surfaces. Fuzzing has emerged as one of the most effective techniques for identifying such vulnerabilities. However, when applied to IoT firmware, several challenges arise, including: (1) the inability of firmware to execute properly in the absence of peripherals, (2) the lack of support for exploring input spaces of multiple peripherals, (3) difficulties in instrumenting and gathering feedback, and (4) the absence of a fault detection mechanism. To address these challenges, we have developed and implemented an innovative peripheral-independent hybrid fuzzing tool called FirmHybirdFuzzer. This tool enables testing of microcontroller-based firmware without reliance on specific peripheral hardware. First, a unified virtual peripheral was integrated to model the behaviors of various peripherals, thus enabling the physical devices-agnostic firmware execution. Then, a hybrid event generation approach was used to generate inputs for different peripheral accesses. Furthermore, two-level coverage feedback was collected to optimize the testcase generation. Finally, a plugin-based fault detection mechanism was implemented to identify typical memory corruption vulnerabilities. A Large-scale experimental evaluation has been performed to show FirmHybirdFuzzer's effectiveness and efficiency.**

***Index Terms*—Internet of Things, Firmware, Hybrid Fuzzing, Vulnerability Detection**

I. Introduction

With the rapid development of communication technologies such as NB-IoT [1] and 5G [2], more and more IoT devices (e.g., IP cameras, smart routers) have been deployed in the security-critical areas including intelligent transportation, smart homes, smart grids and so on. Nowadays, our cyberspace is dominated by billions of low-cost computing nodes. According to the prediction of GSMA [1], the number of IoT devices connected to the Internet will reach 25.2 billion in 2025. When these IoT devices are exposed to attackers via Internet, serious consequences could happen due to loose protection such as missing memory management unit (MMU).

Firmware is the program running on the IoT devices, which is responsible for controlling the hardware, interacting with peripherals, monitoring status, collecting data, and so on. The consequences of a vulnerability in the firmware could be devastating. For example, the Google project zero team disclosed vulnerabilities in the system on a chip (SoC) of Broadcom's wifi, which allows attackers to gain control of a smartphone's main application processor [3]. FreeRTOS is the leading operating system for Amazon's IoT devices, 13 critical vulnerabilities were reported within it, which put a wide range of devices at risk of compromise [4]. Thus, it is essential to detect vulnerabilities within firmware for ensuring the security of IoT devices.

Fuzzing [5], [6], is one of the most effective approaches to exploring security defects in desktop and mobile software systems. Especially, the feedback-based greybox fuzzing [7]–[9] have been widely adopted by many industries including Google [10] and Microsoft [11] to improve the reliability and security of their software products. The core idea of fuzzing is to feed massive inputs to the target program via the interaction interfaces so as to trigger unintended behaviors (e.g., crashes). A lot of techniques such as static analysis [12]–[14], taint analysis [15]–[17], symbolic execution [18]–[21] and machine learning [22]–[25] have been integrated to improve the effectiveness and efficiency of fuzzing.

Different from traditional desktop software and mobile application, IoT firmware has some unique features.

- ***Diverse underlying environments.*** The firmware is usually highly customized to suit the underlying environments, which include the different operating systems (e.g., Bare-metal, FreeRTOS) and various peripherals (e.g., GPS, DSP, communication protocols).
- ***Peripherals based interaction.*** The main task of firmware is usually located in an infinite loop. It continuously interacts with the outside through various peripherals, which are hardware components that handle sensors, actuators, and communication protocols.

Manuscript received January 26, 2022. Revised June 23, 2023. Accepted August 7, 2023

This paper was an extension of the conference paper "Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation" published at the 2020 Annual Computer Security Applications Conference.

Lingyun Situ is with the School of Information Management, Nanjing University, Nanjing, China, 210023. Email: stly@nju.edu.cn

Chi Zhang, Zhiqiang Zuo, Linzhang Wang and Xuandong Li are in the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, 210023.Email: zhangchi@seg.nju.edu.cn, {zqzuo, lzwang, lxd}@nju.edu.cn

Le Guan is with University of Georgia, Athens, GA 30602, USA. Email: leguan@cs.uga.edu

Peng Liu is with Pennsylvania State University, State College, PA 16802, USA. Email: pliu@ist.psu.edu

Jin Shi is with the School of Information Management, Nanjing University, Nanjing, China, 210023. Email: shijin@nju.edu.cn

Corresponding author: Linzhang Wang. Email: lzwang@nju.edu.cn

[1] https://www.gsma.com/

Challenges. These unique features of IoT firmware bring new challenges to fuzzing-based vulnerability detection [26], and existing methods are far from mature.

C1: Inability of firmware execution without hardware dependence. Executing firmware is the premise of using the fuzzing approach. However, it is difficult due to the diverse underlying operating systems and various interactive peripherals. Some works utilize emulators such as QEMU [27] or PANDA [28] to run a firmware binary. For instance, FIRMADYNE [29] enables the QEMU's full system emulation of firmware using a built-in abstraction of a modified Linux kernel. This approach has been utilized to perform dynamic analysis of Linux-based firmware [30]. However, this "built-in kernel abstraction" approach will not be feasible when dynamically analyzing microcontroller-based (MCU) firmware which usually runs on a lightweight real-time operating system (RTOS) or a bare-metal system. One reason is that there are many kinds of RTOSes, and the built-in abstraction provided by one kind of RTOS is often incompatible with another RTOS. Another reason it is infeasible is that the existing mainstream emulator, QEMU, could only emulate the core peripherals (e.g., NVIC, SysTick) of microcontrollers. When accessing unknown peripherals, QEMU will become paralyzed. To deal with this issue, some works such as avatar [31] [32] emulate firmware execution by leveraging a real device or implementing a specific piece of "acting as a peripheral" software for each unknown peripheral [26]. But physically acquiring real devices or manually implementing software-version peripherals are expensive. SymDrive [33] is a system to test Linux and FreeBSD drivers without devices based on symbolic execution engine (i.e., S2E). Our previous work Laelaps [34] models the legitimate behaviors of unknown peripherals using symbolic execution, but aims to finish the peripheral initialization along one promising path. DICE [35] proposes a drop-in solution for firmware analyzers to emulate DMA input channels. HALucinator [36] enables the re-hosting and analysis of firmware by providing generic implementations of identified library functions in a full-system emulator. But, it needs the source code of the corresponding HAL SDKs in the first place, which is usually not available by the analyzers from third parities.

C2: Nonsupport for exploring multiple peripheral input spaces. The IoT firmware running on a device continuously interacts with the environment through a variety of peripherals. In contrast, the program-environment interfaces handled by mainstream fuzzing approaches are a single stdin, file, or network interface. For instance, blackbox fuzzing tools such as Sulley [37], Peach [38] and boofuzz [39] support feeding inputs through a single network interface, while greybox fuzzing tools like AFL [7] AFL++ [40] and Libfuzzer [8] could only deal with a single stdin or file interface. However, the interfaces of the IoT firmware include the network communication and various sensors such as proximity sensor, temperature sensor and so on. In other words, the IoT firmware has multiple input spaces for exploration, but the conventional fuzzers could only support exploration of single input space. Besides, the inputs associated with different peripherals could have big differences in type, syntax, semantics, etc. None of the existing fuzzing tools could handle them well, and it is difficult to upgrade the existing fuzzing tools to support exploring multiple input spaces without significant modification.

C3: Inability of instrumenting and collecting feedback to guide fuzzing. The "feedback-based guidance and "genetic optimization" are the keys of the greybox fuzzing technology. It is best practice for x86 software to adopt a mix of compile-time or run-time instrumentation to collect coverage information of the executed input, therefore guiding fuzzing to optimize the input generation using a genetic algorithm. However, compile-time instrumentation requires the source code which is not typically available for firmware within IoT devices. Existing binary dynamic instrumentation tools, such as Pin [41] and Valgrind [42], are closely tied to the target operating system and CPU architecture. So far, none of the existing binary instrumentation tools could support bare-metal or RTOS firmware images [26]. That is why existing firmware fuzzing works such as RPFuzzer [43] and IoTFuzzer [44] are all black-box fuzzing techniques.

In addition, traditional fuzzing techniques rely on observable crashes as immediate consequences of run-time faults. The Linux- and Windows- based systems usually facilitate various crash detection mechanisms that a fuzzing tool can be leverage. These mechanisms can be triggered through software instrumentation (e.g., stack canaries) or hardware protection (e.g., segment fault). Unfortunately, these mechanisms are rarely present (or limited) in microcontroller-based firmware, resulting in lots of silent crashes [26]. Missing fault detection mechanism greatly limits a fuzzing tool's ability to detect vulnerabilities. In order to alleviate the problem, RPFuzzer [43] detects DoS and router reboot by watching CPU utilization and checking system logs. IoTFuzzer [44] identifies potential vulnerabilities by performing a liveness check. It could identify if the system hangs, but can not report the types and details of the detected vulnerability. Marius Muench [26] presents a set of heuristics to detect memory corruption of firmware, they are useful to a certain, but not systematic and universal.

Our Work. We established a physical devices-agnostic hybrid fuzzing tool for microcontroller-based IoT firmware. It can test mainstream bare-metal and RTOS firmware binaries without peripheral hardware dependence, and detect common C/C++ vulnerabilities. The key solutions integrated in the system to overcome the challenges aforementioned are as follows.

- ***S1:*** A virtual peripheral component has been integrated into the tool, allowing it to emulate the behaviors of various peripherals. This ensures firmware execution without being reliant on specific physical devices. (Section III-A)
- ***S2:*** A hybrid event generation method was used to generate inputs for different peripheral accesses. By combining constraint-based and mutation-based generation method, we can effectively explore the input spaces associated with multiple peripherals. (Section III-B)
- ***S3:*** A two-level coverage feedback mechanism to opti-

mize the generation of test cases. This enables comprehensive coverage of the firmware code and enhances the effectiveness of the hybrid fuzzing. (Section III-C)

Furthermore, a fault detection mechanism was implemented and integrated to identify common vulnerabilities such as stack/heap corruption, integer overflow and division by zero.This modular approach enhances the tool's ability to detect and report vulnerabilities accurately. We performed large-scale experimental evaluation to demonstrate the effectiveness and efficiency of proposed approach and tool.

Contributions. We summarized the key contributions as follows.

- ***Approach.*** We proposed a physical devices-agnostic hybrid fuzzing approach, which enabled the hybrid fuzzing of microcontroller-based IoT firmware without peripheral-hardware dependence.
- ***Tool.*** We implemented a multi-dimensional coverage feedback guided hybrid fuzzing tool named *FirmHybirdFuzzer*, which was specifically designed to fuzz microcontroller-based firmware for vulnerability detection.
- ***Evaluation.*** A large-scale experimental evaluation has been performed to demonstrate the effectiveness and efficiency of FirmHybirdFuzzer.

The remainder of this paper is organized as follows. Section 2 is the background. Section 3 gives the overview and detailed description of the approach. Section 4 introduces the details of implementation. Section 5 illustrates the processes and results of the experimental evaluation. Section 6 presents and discusses the related works. Section 7 is the conclusion.

II. Background

A microcontroller(MCU)-based IoT device is a special kind of embedded device with network connectivity. The sensors are peripherals that collect environmental information, while the actuators are peripherals that exert influence on the physical-world objects. The microcontroller is a single integrated circuit for executing the firmware. We focus on the ARM Cortex-M based Microcontroller in this paper, Fig. 1 illustrates the architecture diagram of MCU devices. The firmware is a program consisting of the bootstrap code, tasks, libraries, the operating system, and interrupts service routines. It is running in the processor core and interacts with peripherals via memory-mapped registers, interrupts, and DMA. The firmware is responsible for collecting data from sensors and network interfaces and generating commands to the actuators and network interfaces.

To better understand the semantics of coverage feedback collected in our approach in the following, we define the concepts of firmware inter-procedure control flow graph (FICFG) and firmware peripheral access dependence graph (FPADG) as follows.

Definition 1: A FICFG is a directed graph $G = (V, E)$, where:

- A vertex $v_i \in V$ represents a basic block $b_i \in B_I \cup B_T \cup B_L \cup B_R$ of the firmware image. Note that the

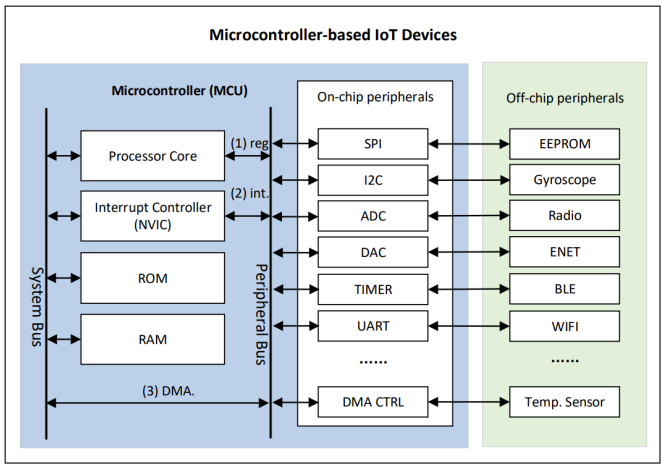

Fig. 1. Architecture diagram of MCU devices

B_I, B_T, B_L, B_R represent the sets of basic blocks of bootstrap code, tasks, underlying operating system, and interrupt service routines respectively.
- An edge $(v_i, v_j) \in E$ represents the dynamic flow of control $c_i \in C_{branch} \cup C_{funCall} \cup C_{Interrupt}$ from b_i to b_j of the firmware image. The C_{branch}, $C_{funCall}$, $C_{Interrupt}$ represent controls from program branches, function call and interrupt responses respectively.

Definition 2: An FPADG is a directed graph $G = (P, E, \rightarrow)$ that specifies the dependence of different peripheral access points at run-time, where:

- A vertex $p_i \in P$ represents a program point of accessing the peripheral to read or write data.
- An event $e_i \in E$ represents the data-read at a peripheral access point p_i. Note that data-write is not included because data-write will not affect the control flow of the firmware execution and the transition of peripheral accesses.
- An edge $(p_i, e_i, p_{i+1}) \in \rightarrow \subseteq P \times E \times P$ is the transition from p_i to p_{i+1}. Note that we write $p_i \xrightarrow{e_i} p_{i+1}$ for a short notation for (p_i, e_i, p_{i+1}).

The execution of a task is event-driven. Hence, events generated by non-network peripherals can be viewed as part of the execution context of the task, while the events generated by network peripherals can be viewed as network input. Formally, we denote the events from network peripherals as $E_{network}$, and events from other peripherals as $E_{context}$. Furthermore, we define the testcase for a firmware execution as follows.

Definition 3: A testcase $t \in T$ for a firmware execution is a sequence of events $< e_0, e_1, ..., e_i, ..., e_n >$, where $e_i \in E_{network} \cup E_{context}$, $0 < i < n$.

Example. An abstract of the firmware code fragment is illustrated in Fig. 2 for a better understanding of the concepts defined above. The instrument information in each basic block is abstracted to retain the read and write events at peripheral access points or function calls in Figure 2. The solid line represents the C_{branch}, and the dotted line represents the $C_{funCall}$. Then, the FICFG and FPADG could be extracted as shown in Figure 3 and Figure 4. Furthermore, one possible

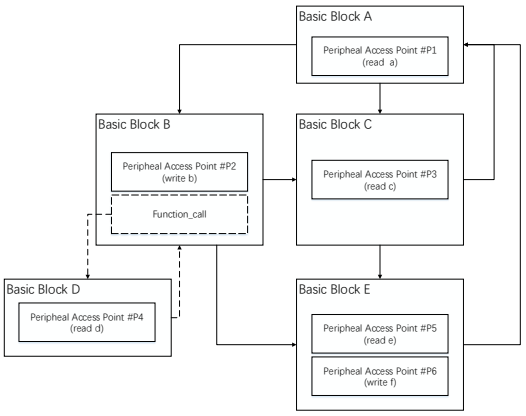

Fig. 2. Abstract of Firmware Code Fragment

execution path of the firmware is $< A, B, D, C, E, A, ... >$, and the testcase for the execution path is $< a, b, d, c, e, ... >$.

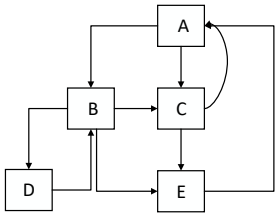

Fig. 3. FICFG

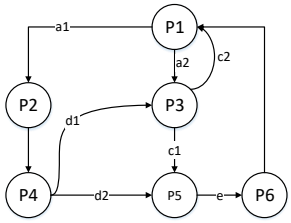

Fig. 4. FPADG

Problem Formalization. Hybrid fuzzing of microcontroller-based firmware without peripheral hardware dependence refers to executing the microcontroller-based firmware in a software-based environment and generating effective testcase $t \in T$ by hybrid method (i.e., combining symbolic execution with fuzzing), in order to explore more execution paths of firmware, try to maximize the coverage of FICFG and FPADG, and to detect vulnerabilities at the same time. Formally, the problem could be formalized as follows.

$$\begin{aligned} HybridGeneration(T) \rightarrow & Maximize(FICFG) \wedge \\ & Maximize(FPADG) \wedge \\ & \exists t \in T, TriggerBug(t). \end{aligned}$$

Why does microcontroller-based firmware matter? We target microcontroller-based firmware for two reasons. (1) Microcontrollers have a huge user base. ARM Cortex-M family is the dominating product in the microcontroller market. It consists of a lot of cores including Cortex-M0Cortex-M3, Cortex-M4, and so on. For each ARM core, the ARM company defines the basic functionality and the memory map for its core peripherals such as the interrupt controller (Nested Vector Interrupt Controller, NVIC), system timer (SysTick), and so on. Then, ARM sells the licenses of its cores as intellectual property (IP). The participating manufacturers who bought the IP will be free to customize their implementation as long as it conforms to the standard of the ARM core. Different manufacturers customize their products in different ways, leading to a vast diversity of Cortex-M processors with various other custom-made peripherals functions. ARM Cortex-M processors map everything into a single address space, including the ROM, RAM, and different peripherals (i.e., memory-mapped IO). All these peripheral functions are invoked by accessing the corresponding registers or memory (i.e, Direct Memory Access, DMA).

The most remarkable difference between PC/mobile processors and Cortex-M processors is that Cortex-M processors do not support *Memory Management Unit (MMU)*. This means 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. Thus, many other ecosystems have been developed around it including Amazon FreeRTOS [45], Arm MbedOS [46], and so on. Bare-metal and RTOS-based firmware is the majority in the market of IoT devices. In 2017, the proportion of microcontroller devices reached 66% among all IoT devices [2]. (2) Nowadays, there is no physical device-agnostic greybox fuzzing tool that can effectively execute and fuzz microcontroller-based firmware.

Why is hybrid fuzzing needed? Hybrid fuzzing [47] [20] [18] is an advanced technique that combines the advantages of fuzzing and symbolic execution. Fuzzing [48] [49] is one of the most effective approaches to exploring vulnerabilities in software systems. Especially, the coverage-based greybox fuzzing [7] [8] have been widely adopted by industries such as Google [9], Microsoft [11] and so on. It usually applies lightweight instrumentation to collect coverage feedback to optimize the input generation using a genetic algorithm [50] [51] [16] [13], which has proven extremely successful in detecting vulnerabilities of the traditional desktop software. Fuzzing is good at generating massive inputs to explore the code space efficiently. But it is hard to bypass some constraints such as magic bytes comparison and so on. Besides, existing greybox fuzzing tools can not generate massive effective inputs for diverse peripherals interfaces without corresponding valid seeds, which could be generated by symbolic execution without manual specification. Symbolic execution [52] is a program analysis technique based on simulation execution. Its core idea is to use symbolic values, instead of concrete values, as the input of the program. Together with constraint solvers [53] [54], symbolic execution engine (e.g., KLEE [55], Angr [56] and Sage [19]) is able to automatically generate concrete inputs for a feasible execution path by solving the corresponding path constraints. But the overhead of using a solver is large and symbolic execution will be stuck into path explosion problems when handling complex structures such as loops. Thus, it will be a large cost to generate massive input to explore code space by using symbolic execution in limited testing resources, while fuzzing's advantage is able to generate a lot of inputs efficiently for exploration. By combining the fuzzing and symbolic execution, hybrid fuzzing could generate a lot of inputs to explore the code space of firmware in limited testing time, generate valid input that leads the firmware to

[2]https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

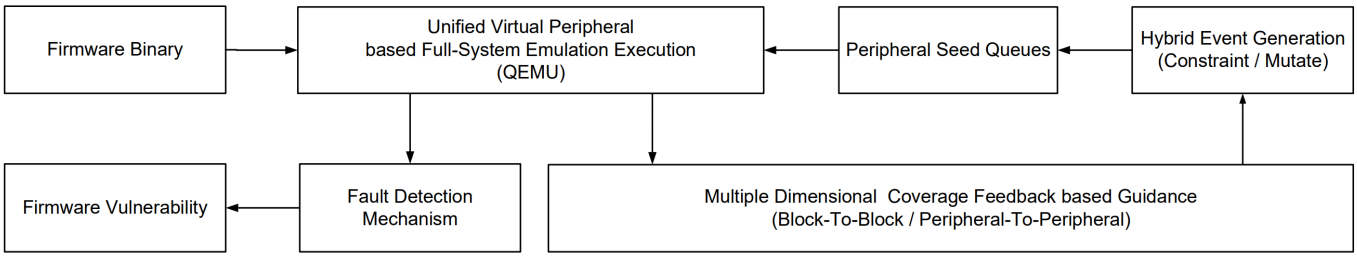

Fig. 5. Overview of Physical Devices-Agnostic Hybrid Fuzzing for IoT Firmware

execute along one specific path, and bypass some complex constraints which are hard to fuzzing. Thus, hybrid fuzzing is the best choice for achieving our goals.

III. Approach

The framework for the physical devices-agnostic hybrid fuzzing of IoT firmware is depicted in Figure 5. The inputs to the framework consist of the IoT firmware image and the corresponding fuzzing configuration. The fuzzing configuration includes details such as the core profile (e.g., ARM Cortex-M3/4), ROM and RAM locations, and fuzzing settings (e.g., fuzzing boundary, single/total execution time). The outputs of the framework are the identified firmware vulnerabilities.Unlike traditional fuzzing tools that focus on fuzzing either user-space programs [7] [16] or kernel-space drivers [57] [58] separately, our tool operates on the firmware as a whole, encompassing both the application tasks and the kernel. This approach allows for a more comprehensive testing of the firmware. The key techniques of the approach include symbolic peripheral-based execution, a symbolic peripheral-based execution method was employed to enable firmware execution in the absence of physical peripherals. hybrid event generation, a constraint-based and mutation-based test case generation method was used to generate inputs for different peripheral accesses. It enables the exploration of various input spaces associated with multiple peripherals. Multi-Feedback Guidance, a two levels feedback guidance mechanism was utilized to optimize the generation of test cases. Fault Detection Mechanism, a fault detection mechanism is implemented to identify common firmware vulnerabilities, such as memory corruption issues.

Intuitively, we take the abstract firmware code segment in Fig.2 for an example to illustrate the procedure of hybrid fuzzing of IoT firmware.

(1) The target firmware binary is executed in the mainstream emulators such as QEMU [28] with full-system emulation mode. We load ARM core peripheral into the system map, and all the unimplemented memory regions are marked as *special* memory. All the peripheral accesses to the *special* memory by assembly instructions such as *ldr* and *str* are intercepted at runtime and forwarded to the unified virtual peripheral. The virtual peripheral models the behaviors of various unknown peripherals of QEMU by feeding effective value for each peripheral access. By this means, we enable the firmware execution without any real peripheral hardware. In Fig. 2, when the firmware execution in the emulator reaches the peripheral access point $\#p1$ in basic block A, the firmware will read data from $p1$'s seed queue and continue execution, rather than paralyzed. The events in the seed queue of peripherals are generated by the hybrid events generation approach.

(2) The hybrid event generation leverages constraint-based and mutation-based generation techniques to generate two kinds of events. One is valid events that satisfy the syntax of peripheral access, the other one is semi-valid events that may trigger abnormal behaviors. The constraint-based generation (i.e., symbolic execution) generates events by collecting executed path constraints and solving them using SMT solvers. Mutation-based generation (i.e., fuzzing) generates events by mutating existing valid seeds with a sequence of mutation operators. In Fig. 2, when the execution reaches a peripheral access point such as $\#p1$, the execution in QEMU will suspend. An event from the corresponding seed queue of the peripheral access point should be fed to the emulator to continue its execution. If the length of the seed queue is less than a limit, which is specified by configuration. The execution state at the current moment in QEMU including the status of registers and memory will be transferred to the symbolic execution engine (i.e., Angr). By marking the input a that peripheral read as symbolic, symbolic execution from the current state is performed for a few steps to explore some possible future paths. The steps of forward exploration is configured by *Forward_Depth*. Assume the symbolic execution is performed for two steps and collect some execution paths such as $p^1 =< A, B, D >$, $p^2 =< A, B, C >$ and $p^3 =< A, C, E >$. We will choose one path by three rules listed in section 3.2.1, and invoke the solver to generate a concrete value for a. The value will be put into the seed queue and fed to the QEMU and continue the firmware execution from the suspend state. Similarly, if the peripheral access point is $\#p3$, which is a function call (i.e., *bl* instruction) to receive a packet from the network peripheral. The receive function will be hooked and replaced by a callback function, which reads a valid packet from the local file as seed for c. The seed will be written into the buffer address of the receive function at run-time. When the length of the seed queue for the peripheral access point is larger than the limit, mutation-based generation will be involved in the generation of the peripheral input. It uses a sequence of mutation operators listed in section 3.2.2 to modify existing seeds to generate more events. The scheduling strategy (i.e., section 3.2.3) of constraint-based generation and mutation-based generation will be adopted to determine which event generation technique to use at the moment.

(3) When an event is executed, its performance will be

evaluated. Two kinds of coverage feedback are collected to evaluate the performance of the executed event during fuzzing (i.e., section 4.3). They are the transition of the block to block (i.e., BB2BB) and peripheral access point to peripheral access point (i.e., PP2PP). This feedback is used to optimize the event generation using a genetic algorithm [59]. If the executed seed triggers a new BB2BB or PP2PP, we will mutate the seed by applying a sequence of predefined mutation operators to generate more seeds and store them in the seed queue of the corresponding peripheral access point. Note that the reason why we use PP2PP as a new coverage feedback to guide fuzzing is that maximizing the peripheral access dependence graph is good at exploring all the peripheral input spaces. Besides, due to the interrupt-driven mechanism, the PP2PP-based guidance may traverse all the interrupt scheduling situations, which is good at exploring the behavior of firmware execution.

(4) During the firmware execution in the emulator, a plugin-based fault detection mechanism is implemented and deployed to improve the sensitivity of security violations. It could identify common C/C++ implementation defeats by monitoring the execution status including registers, memory read and write, return address, and so on. A detailed vulnerability report will be exported when a vulnerability is identified. More specifically, three tracking plugins including stack tracking, heap tracking, and instruction tracking (i.e., section 4) are implemented to identify stack and heap corruption, integer-overflow, and division-by-zero vulnerabilities.

A. S1: Virtual Peripheral-based Execution

The inability of firmware execution without real devices is the biggest challenge to achieving physical devices-agnostic firmware fuzzing. Mainstream emulators such QEMU are limited by their inadequate support for previously unknown peripherals. When the firmware accesses an unknown peripheral, the emulator will be paralyzed, because the emulator does not know how to respond to it.

We propose a unified symbolic peripheral to model the behaviors of various unknown peripherals. In terms of behavior emulation for unknown peripherals, we extend previous work Laelaps [34] by supporting network peripherals and adding mutation-based event generation that model the unconstrained behavior. We use the unified virtual peripheral to interact with the firmware executed in the emulator, and all the I/O accesses to unknown peripherals are intercepted and forwarded to the unified virtual peripheral. The unified virtual peripheral will respond based on its emulation ability of peripheral behaviors. By this means, we achieve firmware execution without hardware dependence, which provides the basis for physical-agnostic firmware hybrid fuzzing. More specially, the unified virtual peripheral models three key behaviors of various unknown peripherals.

Peripheral discovery. The virtual peripheral will respond to the emulator when the processor accesses an unknown memory region corresponding to the peripheral by the way of direct memory access (DMA). This allows for peripheral discovery if the process accesses the memory corresponding to a peripheral.

I/O interaction. By interacting with peripheral read operations and providing effective or ineffective responses, the firmware execution can follow feasible execution paths including the error handling part. Compared with Laelaps [34] that could only handle context peripherals, We provide the support of network peripherals such as ENET, WIFI, Smac, Bluetooth, and so on. Note that data-write operations are ignored because the data-write will not affect the control flow of the firmware and the transition of peripheral accesses.

Interrupt injection. The unified symbolic peripheral supports injecting and delivering active interrupts to the emulator. Based on the QEMU machine protocol, we added three new QMP commands (i.e., active irqs, inject irq, and inject all irqs) and implemented a python interface, which could be used to inject randomly activated interrupts to QEMU. We use the python interface to randomly deliver active interrupts automatically. The simple design works fine for two reasons. First, in a real execution, firmware only activates a limited number of interrupts. Therefore, randomly delivering unexpected interrupts will not introduce to much performance penalty. Second, interrupt handlers can often gracefully deal with unexpected events. Although additional code is executed, it will not cause a great impact on firmware execution. Note that we do not regard the interrupt as the fuzzing entry when performing firmware fuzzing.

Different from previous work [26] that implements a precise software-based simulator for the corresponding peripheral device, our unified virtual peripheral is a general model for various peripherals. Its key functionality is to feed values for peripheral accesses and to drive firmware to execute along specific paths. Note that the symbolic peripheral may generate values beyond the valid range of real peripheral devices. But we argue that this unconstrained device behavior is acceptable for exploring the path of firmware and detecting vulnerabilities, especially in error handling code. Because sometimes the firmware would be crashed when the peripheral device provides an errant value in physical attacks [60].

B. S2: Hybrid Event Generation

The nature of multi-peripherals based interaction in IoT system design invalidates the existing greybox fuzzing tools. Existing tools do not support the exploration of multi-peripheral input spaces because they do not support exploring multiple interactive interfaces. They are unable to generate effective inputs for diverse peripherals because the inputs to different peripherals could differ significantly in terms of type, syntax, range, etc. To support multi-peripheral input spaces exploration, we propose a hybrid input generation mechanism for peripheral-based firmware fuzzing. The mechanism consists of constraint-based and mutation-based generation techniques.

1) Constraint-based Generation.: The constraint-based generation utilizes a symbolic execution engine (e.g., Angr) to generate external input from peripherals. When the firmware execution in QEMU accesses a memory region corresponding to unknown peripherals, the QEMU will suspend. The current

```
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 1. Code snippet using timer peripheral

state of QEMU including the memory and register information will be transferred to the symbolic execution engine. Then, symbolic execution is performed starting from the current state to explore future paths. We choose one path according to the heuristic-based path selection strategy and invoke an SMT solver to generate the corresponding input by solving the path constraints. The input will be stored in a seed queue dedicated to each peripheral access point and the queue is fed to the suspended QEMU and continue the firmware execution. We summarize the key techniques of constraint-based generation as follows.

Access-based symbolization. The core idea of symbolic execution is to use symbolic values, instead of concrete values, as the inputs of the program. We utilize a peripheral access-based symbolization technique to insert symbol values, that is we assign symbols for every peripheral access, even if the peripheral access point has been visited before. This is because of the volatile nature of peripheral memory, their values will change non-deterministically. In this sense, we assign new symbols spatially (i.e., different access points get different symbols) and temporally (i.e., different times get different symbols). Consider an example shown in Listing 1. $base \rightarrow CNT$ is a peripheral register that keeps an increasing counter. Line 1 and line 2 read the current values. Although they are accessing the same memory, since they accessed at different times, we assign two different symbols. Otherwise, line 5 can never be reached because subtracting a variable from itself always gets zero.

Speculative symbolic execution. Speculative symbolic execution technique [61] is adopted to reduce the overhead of invoking a constraint solver. We do not invoke the solver once we encounter a new branch. Instead, we allow the analysts to configure $context\_depth$ to specify the number of branches the symbolic execution engine has to accumulate before invoking the solver. The downside is that a larger $context\_depth$ leads more paths to be explored in symbolic execution and thus consumes more time overhead. There are three rules consisting of the heuristic-based path selection algorithm.

- ***Rule 1: favoring deep path.*** We prefer to select the path with the highest address due to it has a higher chance to move forward quickly and deeply. This is based on two key observations. First, programs are designed to execute sequentially. Second, the booting code of firmware typically initializes each peripheral one by one. We design the rule because we tend to move forward and deep quickly.
- ***Rule 2: prioritizing new path.*** We prioritize the untouched path in order to maximize coverage. We maintain a list of previously executed basic blocks. By calculating the similarity between the historical paths and each of the explored paths for the future, we choose the candidate path with the lowest similarity. The way to calculate the similarity is based on the ratio of intersection to a union of the future and historical paths.
- ***Rule 3: avoiding infinite loop.*** We choose the path based on infinite loop elimination. This is because symbolic execution will be stuck in path explosion due to an infinite loop [62]. Instead of applying fixed-point theorems [63] to identify an infinite loop, we compare the state (e.g., registers and program counter) of current paths with that explored before. If two states are the same, we regard the current path as one with an infinite loop.
  More specifically, we allow the analysis to specify a parameter *forward_depth*, which is the maximum number of basic blocks that the symbolic execution engine can advance from a branch along one path. 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 one single path from the branch is selected, we say it chooses the path based on the infinite loop elimination.

2) Mutation-based Generation.: The mutation-based generation utilizes a sequence of mutation operators to modify existing seeds and generate new events. These mutation operators are usually used by greybox fuzzing tools such as AFL and LibFuzzer. When the firmware execution in QEMU reaches a peripheral access point to read value, the existing valid seeds will be mutated by applying mutation operators such as bit-flips to generate new events. These new events are stored in the seed queue of a corresponding peripheral access point and fed to QEMU to continue its concrete execution. More specifically, we summarize the techniques of mutation-based generation as follows.

Peripheral-aware valid seed generation. The initial seed that is used for subsequent mutation is usually valid with correct syntax and semantics. It will provide a better basis for mutation-based fuzzing. More specifically, for context peripherals, we provide valid seeds by using a symbolic execution technique. For network peripherals, it is hard for symbolic execution to generate an input that satisfies the packet format. Thus, We use the hook function to replace the network function and read valid packets from the local files.

Peripheral-specific mutation capability. A set of mutation operators are designed by considering the characteristics of peripheral-based interaction. The key observation is that (1) the peripheral access is achieved by reading from or writing to memory regions corresponding to peripherals, (2) the value types from most context peripherals are integers, and (3) the value types from network peripherals are packets. Therefore, we designed peripheral-specific mutation operators as follows.

- ***Bit Flip.*** It flips 1, 4, 8, 16, 32, 64, 128 bits of existing seeds;
- ***Arithmetic.*** It adds or subtracts random integer values from existing seeds.

- ***Interesting.*** It inserts some interesting values into seeds such as the fixed value used in condition expressions.
- ***Overwriting.*** It overwrites some blocks of existing seeds by fixed content (e.g, header) at a specific field of protocols.

3) Probability-based Scheduling: Generally, constraint-based generation is good at generating events to drive firmware execution along a specific path. A mutation-based generation has advantages in generating diverse events that explore more paths, including error handling code.

We prefer to use constraint-based generation first because the firmware execution will paralyze at an early stage if we use mutation-based generation with random initial seeds first. The detailed scheduling process of hybrid event generation is as follows: (1) If the length of the seed queue for the corresponding peripheral is smaller than the manually specific limit, then we use constraint-based generation (i.e. symbolic execution) to generate a seed to execute. (2) If the length of the seed queue for the corresponding peripheral is larger than the manually specific limit, then the mutation-based generation is involved. We begin to use the probability-based scheduling strategy to adjust the usage of constraint-based generation and mutation-based generation as follows.

More specifically, the transfer between constraint-based generation and mutation-based generation is based on the manually specified probability. A probability is set for the selection of constraint-based generation. If we set the probability to be 30%, then we have a 30% chance to choose constraint-based generation when faced with the choice of the above two generation methods. The detailed algorithm of the hybrid event generation is illustrated in the following.

Algorithm 1: Hybrid Event Generation

```
Data: firmware binary b
Result: seed queues of peripherals S
Initialization;
foreach peripheral access point pc that b reaches in QEMU do
    state s → QEMU(b, pc);
    suspend QEMU(s);
    event fed_e = null;
    if sizeof(S[pc]) ≤ limit then
        transfer(QEMU(s), Angr(s'));
        path p = heuristic-based-path-selection (s');
        event e = SMT_solver(p);
        e → S[pc];
        fed_e = e;
    else
        if constraint-based generation then
            transfer(QEMU(s), Angr(s'));
            path p = heuristic-based-path-selection (s');
            event e = SMT_solver(p);
            e → S[pc];
            fed_e = e;
        else
            fed_e = S[pc][0];
    restart QEMU(s);
    BB2BB, PP2BB = process(fed_e);
    if TriggerNew(BB2BB) or TriggerNew(PP2PP) then
        eSet = mutate(fed_e);
        eSet → S[pc];
    else
        delete fed_e from S[pc];
```

C. S3: Two Levels Feedback-based Guidance

Feedback guidance and genetic optimization are the underlying key innovation of the greybox fuzzing technology. The inability of instrumenting and collecting feedback brings a big obstacle to achieving greybox fuzzing for IoT firmware. That is why existing firmware fuzzing works such as RPFuzzer [43] and IoTFuzzer [44] are all black-box fuzzing techniques – they can not collect the runtime coverage information to guide fuzzing.

We overcome this issue by utilizing the indirection layer provided by QEMU. Since the firmware is translated by QEMU, we can easily obtain the runtime information during firmware execution. This forms the basis for our approach to dynamically collect feedback information, including executed basic blocks, visited peripheral access points, etc. Different from the AFL's QEMU mode [7] that uses the single path coverage as the feedback to guide fuzzing, we collect multi-dimensional coverage feedbacks, i.e., the transitions of block-to-block and peripheral-to-peripheral, to advance the hybrid firmware fuzzing. The reason why we additionally use the peripheral-to-peripheral coverage is that we hope to steer the fuzzing toward new peripheral access points. Intuitively, a new peripheral is only accessed after the previous peripheral has been successfully initialized. By guiding the execution to access new peripherals, we have a good chance to cover new paths.

BB2BB Coverage. The transition of the block-to-block (i.e., BB2BB) refers to the execution from one basic block of the firmware to another. We use the entry address of a basic block *A* as its unique identification, which is denoted by *ID(A)*. Furthermore, the transition from basic block *A* to basic block *B* is uniquely represented by $ID(A \rightarrow B) = (ID(A) \ll 1) \oplus ID(B))$. We track all the visited unique BB2BBs at run-time because the number of visited unique BB2BB is a direct coverage metric of FICFG. If a new BB2BB is visited, a pair including its ID and hit information $< ID(BB2BB), Hit >$ will be stored into a shared bitmap. Furthermore, if an event triggers a new BB2BB, we regard it as interesting and will mutate it to generate more seeds by applying mutation operators.

PP2PP Coverage The transition of peripheral to peripheral (i.e., PP2PP) represents the execution from one peripheral access point to another. We use the address of access instruction (e.g., *ldr* and *str*) at the peripheral access point *A* as its unique identification, which is denoted by *PID(A)*. Then, the transition from peripheral access point *A* to peripheral access point *B* is

uniquely represented by a tuple of $< PID(A), PID(B) >$. We track all the unique PP2PP at run-time because it is a direct metric to show the coverage of FPADG. All the visited unique PP2PP are stored in a list. The list acts as a coverage map of PP2PP. If an event triggers a new PP2PP, we will store the new PP2PP into a list, we regard it as interesting and mutate it to generate more events.

Coverage Feedback Guidance. The coverage feedback based guidance is aimed to maximize the coverage of firmware inter-procedural control flow graph and peripheral access dependence graph. We achieve the goal by guide the hybrid fuzzing to mutate on interesting seeds that trigger new BB2BB or PP2PP and generate more effective seeds based on the underlying genetic algorithm.

D. S4: Fault Detection Mechanism

Traditional fuzzing techniques rely on observable crashes as immediate consequences of run-time faults. The Linux- and Windows-based systems usually facilitate various crash detection mechanisms that a fuzzing tool can be leveraged. These mechanisms can be triggered through software instrumentation (e.g., stack canaries) or hardware protection (e.g., segment fault). Unfortunately, these mechanisms are rarely present in microcontroller-based firmware, resulting in lots of silent crashes [26]. Missing fault detection mechanism greatly limits a fuzzing tool's ability to detect vulnerabilities. To address this problem, we implemented a fault detection mechanism for microcontroller-based firmware inspired by the heuristics used to detect memory corruption in PANDA [26]. Because the latest QEMU has better emulation capability for ARM Cortex-M based devices, we ported the mechanism of PANDA into the latest QEMU and extended it to support detecting more vulnerabilities. More specifically, three tracking plugins are designed and implemented as QEMU TCG plugins [3] in the fault detection mechanism.

Stack Tracking. This tracking plugin is designed to identify stack overflow and out-of-bound read and write vulnerabilities. More specially, we monitor all the direct and indirect function calls as well as the return instructions. We check if their return addresses are overwritten. Furthermore, we track all the stack frames of function calls and check if the contiguous memory accesses cross corresponding stack frames.

Heap Tracking. This tracking plugin is designed to detect both temporal and spatial heap-related bugs such as heap overflow, use after free, double free, and so on. It achieves its goal by evaluating the arguments and return values of allocation and deallocation functions and bookkeeping the location and sizes of heap objects. This allows for easily detecting out-of-bounds memory accesses or access to a freed object.

Instruction Tracking. This tracking plugin is designed to detect integer overflow and division by zero vulnerabilities. We track the execution state of each instruction and check if the register value that acts as the operand of division instructions equals zero. Furthermore, We trace instructions for all arithmetic operations (e.g., *add*) to see if the result is outside the range of different integer types, thus detecting the integer overflow vulnerability.

[3]https://github.com/guillon/qemu-plugins

IV. IMPLEMENTATION

The hybrid fuzzing system for IoT firmware was implemented based on our previous work Laelaps [34] and AFL [7]. We integrated them into our system and add over 4.4K lines of C code and 5K lines of python code. The details of some key implementations are presented as follows.

I/O Interception. The I/O operations performed by firmware accessing unknown peripherals are intercepted and forwarded to a unified symbolic peripheral. We implemented it based on Avatar2, which implements a remote memory mechanism in which accesses to unmapped memory regions in QEMU are forwarded to a python script. We modified the python script as the symbolic peripheral with the ability of the hybrid event generation.

State Transfer. The avatar uses GDB interface to synchronize the state of the register and memory, but it must be issued when the target is stopped. In our scenario, we can not predicate the point of firmware execution that accesses unknown peripherals and set breakpoints beforehand. We overcame this issue by invoking the QEMU internal function to suspend the firmware execution when encountering unknown peripherals on the fly. We implemented the on-the-fly state transfer by exporting all RAM regions through shared memory-based inter-process communication. A POSIX shared memory object is created and bound to the RAM region using mmap when a RAM region is created in QEMU. As a result, the symbolic engine can directly address the firmware RAM by reading the exported shared memory.

Interrupt Injection. We implemented the interrupt injection based on the QEMU machine protocol (QMP), which is a JSON-based protocol. We added three new QMP commands, i.e., active-irqs, inject-irq, and inject-irq-all. They could be used to get the current activated interrupt numbers, inject an interrupt, and inject all the activated interrupt numbers. 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).

Feedback Maintenance. We implemented the feedback collection and maintenance mechanisms based on the shared memory and the pickle, which is a python library for object serialization. More specifically, a shared bitmap is created the first time the IoT firmware image is executed. The bitmap is used to store information about unique BB2BB transitions. We implemented it based on AFL's QEMU mode. In addition, a shared list in Python is established at the same time, which is used to store the unique peripheral access to peripheral access transitions. And a Python map between each peripheral access point and its seed queue information is created. This feedback information is shared between QEMU and Python in the shared memory.

Open Source. The core source code of the physical devices-agnostic hybrid fuzzing system for IoT firmware is online available as follows.

https://github.com/stuartly/FirmHybridFuzz

V. Evaluation

The section presents the evaluation of the proposed hybrid fuzzing system for IoT firmware. Since there is no ready-to-use benchmark to evaluate a fuzzing tool designed for microcontroller-based firmware, we designed and built the first set of firmware images suitable for assessing the capability of a fuzzing tool. Then we design the evaluation questions and discuss the detailed evaluation processes and results.

Benchmark. Inspired by LAVA [64], we established a benchmark to evaluate microcontroller-based firmware fuzzing tools. First, we selected firmware samples from the SDK provided by the chip manufacturers. The samples are diverse in terms of microcontrollers, underlying operating systems, and integrated peripherals.

- ***Microcontroller.*** Four ARM Cortex-M based microcontrollers (i.e., NXP FRDM-K66F, NXP FRDM-KW41Z, STM32 L475VG and STM32 Nucleo-L152RE) are selected. The reason why we choose them is that we have real development boards, which is helpful for us to debug the firmware execution and validate the fuzzing results.
- ***Operating System.*** Three popular real-time operating systems (i.e., FreeRTOS [45], MbedOS [46], and ChibiOS [65]) as well as Bare-metal are included. FreeRTOS is a market leader in the market of IoT devices. MbedOS is the official embedded OS for ARM Cortex-M based IoT devices. ChibiOS [65] is another compact and efficient RTOS supporting multiple architectures, especially for STM32 devices.
- ***Peripheral.*** More than 40 different peripherals are considered. They are ranging from basic sensors (e.g., ADC, LED, UART, etc) to complex network interfaces such as WIFI, BLE, and so on.

Then, we injected typical bugs into the firmware including stack overflow, heap overflow, out-of-bounds r/w, null pointer deference, use-after-free, double free, integer overflow, and division by zero. More specially, the process of bug injection is as follows. (1) Construct the inter-procedural call graph of the whole firmware using IDA pro; (2) Extract the sub-call graph of the main task;(3) Select some functions in the sub-call graph; (4) Insert code fragments [4] containing typical vulnerabilities at the entry block and other basic blocks of the selected function's source code. The script (i.e, BugInjecter.py) used for bug injection are also online available. Finally, the benchmark forms a ground truth, which could be used for the evaluation of security analysis techniques of IoT firmware.

A. Evaluation Questions

Our goal is to establish a physical device-agnostic hybrid fuzzing system for IoT firmware. We selected representative firmware as the benchmark to perform our evaluation. The evaluation was performed to demonstrate our approach's effectiveness and efficiency by answering the following questions.

- ***Q1:*** How is the effectiveness of the unified symbolic peripheral in enabling hardware-independent firmware execution?
- ***Q2:*** How is the effectiveness and efficiency of hybrid Fuzzing in exploring firmware code space?
- ***Q3:*** How is the effectiveness and performance of fault detection plugins in identifying vulnerabilities?
- ***Q4*** : How is the comparison between *FirmHybirdFuzzer* and related firmware fuzzing tools?

B. Evaluation Results

For large-scale deployment and evaluation, we performed the evaluation on a virtual machine with 8 Intel(R) Xeon(R) CPU E5-1650 v3 cores and 8GB memory, running a 64-bit Ubuntu 16.04 LTS system.

1) Q1: Effectiveness of unified virtual peripheral in enabling hardware-independent firmware execution: To evaluate the influence of the unified virtual peripheral on firmware execution, we chose the firmware images with different microcontrollers, OSs, and peripherals as listed in Table I. For each firmware image, we set the starting point of fuzzing at the beginning of firmware execution, and we set the ending point of fuzzing after device initialization and right before the main task. We used the boolean value to evaluate the influence of the unified virtual peripheral (VP) on firmware execution. That is whether the unified virtual peripheral could successfully enable the firmware execution to finish the device initialization and perform the main task, rather than being paralyzed when accessing unknown peripherals. We tested 62 firmware and collected the results as illustrated in Table I.

The results indicate that diverse firmware images with different OSs (i.e., Bare-metal, FreeRTOS, ChibiOS and MbedOS) and microcontroller (i.e., NXP FRDM-K66F, NXP FRDM-KW41Z, STM32-L457VG and STM32-Nucleo-L152RE) could be executed in our system with the help from the unified virtual peripherals. Assisted by the constraint-based event generation, the virtual peripheral could emulate the I/O interaction of over 40 different peripherals ranging from simple ADC to complex communication protocols such as NFC, WIFI, and BLE. Furthermore, with the help of the unified virtual peripheral, our system is able to drive over 88% of the tested firmware images to finish device initialization and to start performing the main tasks. All the firmware can not be emulated and executed without our symbolic peripherals. Note that our system provides interfaces for controlling the firmware execution, these interfaces could be used to (1) bypass code, (2) fix the known peripheral value, (3) overwrite function, and (4) inject interrupt. Some manual analysis for the firmware is needed before using these interfaces. We need to figure out which function should be bypassed, which interrupt signal should be specified in the config file, and so on. For example, some firmware images (e.g., Lwip_Httpsrv_Rtos) depend on some custom-made peripherals (e.g., CRC) to implement

[4]https://samate.nist.gov/SRD/around.php#juliet_documents

TABLE I
INFLUENCE OF UNIFIED VIRTUAL PERIPHERAL (VP) ON FIRMWARE EXECUTION.

MCU	OS	Firmware	Peripheral	Size (KB)	Init without VP	Init with VP	Mannual Assist
NXP-K66F	Bare-Metal	Drive_Adc16_Polling	ADC	999	N	√	×
		Drive_Cmp_Polling	CMP	995	×	√	×
		Drive_Cmt	CMT	999	×	√	×
		Drive_Crc	CRC	995	×	×	×
		Drive_Dac_basic	DAC	995	×	√	×
		Drive_Dspi_interrupt	DSPI	1018	×	×	×
		Drive_Edma_Sactter_Gather	EDMA	1086	×	×	×
		Drive_Enet_Txrx_Transfer	ENET	1029	×	√	×
		Drive_Ewm	EWM	1000	×	√	×
		Drive_Flexcan_Loopback	FLEXCAN	1015	×	√	×
		Drive_Ftm_Timer	FTM	1005	×	√	×
		Drive_Gpio_Input_Interrupt	GPIO	996	×	√	×
		Drive_I2c_Interrupt	I2C	1012	×	×	×
		Drive_Lptmr	LPTMR	998	×	√	×
		Drive_Lpuart_polling	LPURT	985	×	√	×
		Drive_Mcg_Pee_Blpi	MCG	979	×	√	×
		Drive_Pflash	PFLASH	1035	×	√	×
		Drive_Pit	PIT	997	×	√	×
		Drive_Rnga_Random	RNGA	993	×	×	×
		Drive_Rtc	RTC	999	×	√	×
		Drive_Sai_Interrupt	SAI	1156	×	√	×
		Drive_Sdcard_Polling	SDCARD	1084	×	×	×
		Drive_Sysmpu	SYSMPU	1001	×	×	×
		Drive_Wdog	WDOG	996	×	√	×
	FreeRTOS	Rtos_Sem_Static	SEM	357	×	√	×
		Rtos_Swtimer	SWTIMER	354	×	√	×
		Rtos_Uart	UART	344	×	√	×
NXP-K66F	Bare-Metal	Lwip_Dhcp_Bm	DHCP	445	×	√	√
		Lwip_Httpsrv_Bm	HTTPS	526	×	√	√
		Lwip_Iperf_Bm	IPERL	519	×	√	√
		Lwip_Ping_Bm	PING	455	×	√	√
		Lwip_Tcpecho_Bm	TCP	450	×	√	√
		Lwip_Udpecho_Bm	UDP	446	×	√	√
		Lwip_Httpsrv_Mbedtls_Bm	TLS	1151	×	√	√
	FreeRTOS	Lwip_Dhcp_Rtos	DHCP	589	×	√	√
		Lwip_Httpsrv_Rtos	HTTPS	816	×	√	√
		Lwip_Httpssrv_Wolfssl_Rtos	SSL	1275	×	√	√
		Lwip_Ping_Rtos	PING	652	×	√	√
		Lwip_Tcpecho_Rtos	TCP	619	×	√	√
		Lwip_Udpecho_Rtos	UDP	618	×	√	√
NXP-KW41Z	Bare-Metal	Ble_Blood_Pressure_Bm	BLOOD_PRESSURE	1302	×	√	√
		Ble_Health_Thermometer_Bm	THERMOMETER	1303	×	√	√
		Ble_Wireless_Power_Ptu_bm	BLE	1366	×	√	√
		Ble_Glucose_Sensor_Bm	BLE	1310	×	√	√
		Ble_Proximity_Reporter_Bm	PROXIMITY, BLE	1304	×	√	√
		Smac_Connectivity_Test_Bm	SMAC	729	×	√	√
		Smac_Wireless_Messenger_Bm	SMAC	619	×	√	√
	FreeRTOS	Ble_Cycling_Power_Rtos	BLE	1360	×	√	√
		Ble_Pluse_Oximeter_Rtos	OXIMETER	1367	×	√	√
		Ble_Heart_Rate_Rtos	HEART RATE	1368	×	√	√
		Ble_Temperature_Rtos	TEMPERATURE	1457	×	√	√
		Ble_Wireless_Uart_Rtos	UART	1441	×	√	√
		Smac_Low_Power_Rtos	SMAC	756	×	√	√
		Smac_Wireless_Uart_Rtos	SMAC	654	×	√	√
STM32-L457VG	Bare-Metal	NFC_WriteTag	NFC	306	×	√	√
		Nfc_WriteToBleApp	NFC	242	×	√	√
		Wifi_Client_Server	WIFI	273	×	√	√
		Wifi_Http_Server	WIFI	352	×	√	√
		Ble_HeartRate	BLE	18	×	√	√
		Ble_P2P_LEDButton	BLE	15	×	√	√
STM32-L152RE	ChibiOS	Qemu_Uboot	USART	550	×	√	×
	Mbed OS	Hal_Flash	UART	920	×	√	×

complex computations such as check-sum or cryptography, the symbolic peripheral is fundamentally ineffective in handling such sophisticated computation operations limited by the ability of solver. We could migrate it by bypassing the code of these complex operations by pre-provided configuration. For some firmware (e.g., Ble_Wireless_Power_Ptu_Bm) that receives network packets from outside, the data generated using symbolic execution is hard to satisfy the grammar. We address this problem by hooking to replace the network function and read valid packets from the local file. In addition, for some firmware (e.g., Smac_Wireless_Uart_Rtos) that are blocked in the idle task and waiting for a specific interrupt to trigger the main task, we could inject specific interrupt events through system interfaces.

Based on the above observations, we could positively answer **Q1** that the unified virtual peripheral could effectively enable physical device-agnostic firmware execution. Moreover, it could feed effective values for corresponding peripheral accesses, and drive 55/62 firmware execution to finish device initialization.

2) Q2: Effectiveness and efficiency of the hybrid event generation in exploring firmware code space: To evaluate the influence of hybrid event generation, we selected firmware targets that interact with the network input and other context peripherals in their main tasks. The influence is evaluated from two aspects as follows.

Influence of mutating different peripherals. We evaluated the influence of mutating different peripherals by three settings, i.e., (1) mutating context and fixing network, (2) mutating network and fixing context, and (3) mutating both the context and network. For each setting, we fuzzed the main task of each firmware for the same iterations, and collected the coverage information and time cost as illustrated in Table III.

In terms of the coverage, our results indicate that higher coverage were obtained by mutating both the context and network. This proves that exploration of multiple input spaces benefit the coverage improvement. In addition, we found that the behavior of some firmware images (e.g., NXP-FRDM-KW41Z Bluetooth) has more dependence on the network, while some others are greatly affected by the context. The key insight is that context peripherals have important (even bigger) influence than that of networks on the behaviors of IoT firmware. In terms of the time overhead, the time cost of only mutating input is four times as much as that of only mutating the context. This is because mutating network input relies on the hooking plugin that reads local files, which is time-consuming. More importantly, fixing context based on symbolic execution technique is expensive.

Influence of adopting different event generation scheduling strategies. We evaluated the influence of three different event-generation scheduling strategies as follows.

- ***Probability: 100% vs. 0%.*** The probability of using constraint-based and mutation-based generation is 100% vs. 0%.;
- ***Probability: 50% vs. 50%.*** The probability of using constraint-based and mutation-based generation is 50% vs. 50%.
- ***Probability: 10% vs. 90%.*** The probability of using constraint-based and mutation-based generation is 10% vs. 90%.

Note that the probability of using constraint-based and mutation-based generation is 0% vs. 100% is not allowed. Because the symbolic execution (i.e., constraint-based approach) is necessary, otherwise the firmware execution will be stuck in the early stage. For each setting, we fuzzed the main task of each firmware for the same iterations. The results including the coverage information and time cost are shown in Table III. The results indicate the time cost of only using constraint-based generation is four times as much as that importing 90% mutation-based generation. With the probability of using mutation-based generation increases from 50% to 90%, the resulting coverage grows in terms of both BB2BB and PP2PP.

Based on the above observations, we could positively answer **Q2** that hybrid event generation can generate effective values for exploration of multiple peripheral input spaces. The increment of the probability of using mutation-based generation contributes to coverage growth and performance improvement.

3) Q3: Effectiveness and Efficiency of BB2BB and PP2PP coverage feedback guidance on firmware coverage growth: The coverage growth over time was used as the metric to measure the impact of different coverage feedback guidance

TABLE II
RESULTS OF MUTATING DIFFERENT PERIPHERALS. #ITER IS THE NUMBER OF THE MAIN TASK'S EXECUTION, T(S) IS THE TOTAL FUZZING TIME, #BB IS THE NUMBER OF UNIQUE BB2BB, #PP IS THE NUMBER OF UNIQUE PP2PP.

Firmware	#Iter	Mutating Context			Mutating Input			Mutating Both		
		T(s)	#BB	#PP	T(s)	#BB	#PP	T(s)	#BB	#PP
Lwip_Dhcp_Bm	50	387.54	588	5	1215.9	672	5	425.88	685	5
Lwip_Httpsrv_Bm	50	446.83	477	5	1162.86	712	5	576.8	659	5
Lwip_Iperf_Bm	50	448.73	480	5	1252.85	619	5	463.41	632	5
Lwip_Httpssrv_Mbedtls_Bm	50	663.5	791	5	4263.44	941	4	662.8	945	5
Lwip_Httpsrv_Rtos	50	324.54	1823	27	1951.19	1835	16	590.87	1922	30
Lwip_Ping_Freertos	50	396.57	2268	18	851.9	2117	3	222.61	2272	12
Lwip_Httpssrv_Mbedtls_Rtos	50	1541.74	2148	38	7305.09	1993	17	1095.14	2307	38
Lwip_Httpssrv_Wolfssl_Rtos	50	609.53	3062	25	1732.07	3087	16	478.16	3092	25
Ble_Wireless_Power_Ptu_Bm	50	969.43	659	7	3388.18	638	5	1039.9	665	7
Ble_Proximity_Reporter_Bm	50	1132.07	659	7	3313.8	641	5	1322.31	667	7
Smac_Connectivity_Test_Bm	50	626.28	295	23	2742.06	272	19	721.23	293	23
Smac_Wireless_Messenger_Bm	50	658.18	322	26	2743.67	268	19	637.2	295	24
Ble_Pluse_Oximeter_Rtos	50	3545.47	4375	145	21815.08	3847	140	3717.89	4353	148
Ble_Temperature_Rtos	50	3686.65	4302	144	17252	3851	137	2950.04	4411	150
Smac_Low_Power_Rtos	50	250.65	584	8	506.47	564	7	260.6	607	8
Smac_Wireless_Uart_Rtos	50	309.64	1221	25	907.76	892	19	349	1224	27
Nfc_WriteTag	50	1916.23	1016	61	3207.55	821	48	1716.5	954	59
Nfc_WriteToBleApp	50	882.75	778	57	1636.53	656	44	902.11	786	52
Wifi_Client_Server	50	1222.62	886	34	2282.74	658	16	1303.94	877	35
Wifi_Http_Server	50	1114.36	458	1	1490.98	430	24	1530.53	720	57
SUM:		**21133.31**	**27192**	**666**	**81022.12**	**25514**	**554**	**20966.92**	**28366**	**722**

TABLE III
RESULTS OF DIFFERENT EVENT GENERATION SCHEDULING STRATEGIES. #ITER IS THE NUMBER OF THE MAIN TASK'S EXECUTION, T(S) IS THE TOTAL FUZZING TIME, #BB IS THE NUMBER OF UNIQUE BB2BB, #PP IS THE NUMBER OF UNIQUE PP2PP.

Firmware	#Iter	Probability:100% vs. 0%.			Probability:10% vs. 90%.			Probability:50% vs. 50%.		
		T(s)	#BB	#PP	T(s)	#BB	#PP	T(s)	#BB	#PP
Lwip_Dhcp_Bm	50	1208.83	588	5	425.88	685	5	944.42	672	5
Lwip_Httpsrv_Bm	50	1235.79	473	5	387.49	699	5	1051.70	684	5
Lwip_Iperf_Bm	50	1227.56	480	5	463.41	632	5	981.90	631	5
Lwip_Httpssrv_Mbedtls_Bm	50	4518.65	788	4	662.80	945	5	2365.57	972	5
Lwip_Httpsrv_Rtos	50	2085.33	1757	16	590.87	1922	30	1404.51	1933	25
Lwip_Ping_Freertos	50	894.13	2190	3	222.61	2272	12	553.06	2233	14
Lwip_Httpssrv_Mbedtls_Rtos	50	6667.07	2162	17	1095.14	2307	38	3771.88	2144	32
Lwip_Httpssrv_Wolfssl_Rtos	50	1853.08	2944	16	423.16	3134	24	1317.20	3070	24
Ble_Wireless_Power_Ptu_Bm	50	2992.04	638	5	1039.90	665	7	2612.05	658	7
Ble_Proximity_Reporter_Bm	50	2978.19	641	5	1322.31	667	7	2302.36	656	6
Smac_Connectivity_Test_Bm	50	3017.28	278	19	721.23	293	23	1717.62	308	22
Smac_Wireless_Messenger_Bm	50	2687.87	264	19	637.20	295	24	1678.37	300	22
Ble_Pluse_Oximeter_Rtos	50	18635.81	4209	140	3717.89	4353	148	10902.28	4334	142
Ble_Temperature_Rtos	50	18516.39	4225	137	2950.04	4411	150	10918.98	4329	139
Smac_Low_Power_Rtos	50	494.39	603	7	260.60	607	8	435.33	620	7
Smac_Wireless_Uart_Rtos	50	869.15	996	19	349.00	1224	27	769.42	1059	32
Nfc_WriteTag	50	10906.83	778	65	1716.50	954	59	2787.73	951	59
Nfc_WriteToBleApp	50	1626.03	662	42	902.11	786	52	1339.55	770	54
Wifi_Client_Server	50	2240.31	654	16	1303.94	877	35	2110.10	820	34
Wifi_Http_Server	50	476.00	376	1	1530.53	720	57	1961.65	711	56
SUM:		**85130.72**	**25706**	**546**	**20722.58**	**28448**	**721**	**51925.66**	**27855**	**695**

strategies. More specifically, we fuzzed the main tasks of each selected firmware for three hours under three settings, i.e., (1) fuzzing with BB2BB guidance, (2) fuzzing with PP2PP guidance, and (3) fuzzing with both BB2BB and PP2PP guidance. The coverage growth for each firmware under the three settings were collected. Due to the limited space, we show the results of six instances (i.e., Nfc-WriteBleApp, Nfc-WriteTag, Lwip-Httpsrv-Bm, Lwip-Httpssrv-Mbedtls-rtos, Ble-Temperature-Rtos, STM32-L475-NFC-WriteToBleApp, Httpssrv-mbedTLS-FreeRTOS, Lwip Temperature-Sensor-Rtos, Ble-Blood-Presure) in Fig. 6. Please refer to the website[5] for more information.

The results indicate that PP2PP coverage feedback guidance can facilitate the coverage growth effectively. The coverage results of combining PP2PP feedback performs better than using only BB2BB guidance for all the firmware images. Note that PP2PP guidance may perform better than the combined version in some cases as shown in Fig.6. On one hand, it indicates that the influence of PP2PP is larger than BB2BB on guiding exploring new code space for firmware. It is reasonable because firmware is interacting with the environment based on peripheral interfaces. During limited time, guiding firmware execution to new peripherals will contribute more code coverage. On the other hand, new test cases generated based on BB2BB coverage may not bring new code coverage for firmware, but costing extra overhead, thats the reason why PP2PP guidance may perform better than the combined version. In general, the results prove that feedback based on PP2PP is a good coverage feedback for fuzzing IoT firmware, and validates our insight that steering the fuzzing towards new

[5]https://sites.google.com/view/hybrid-fuzzing-firmware/home

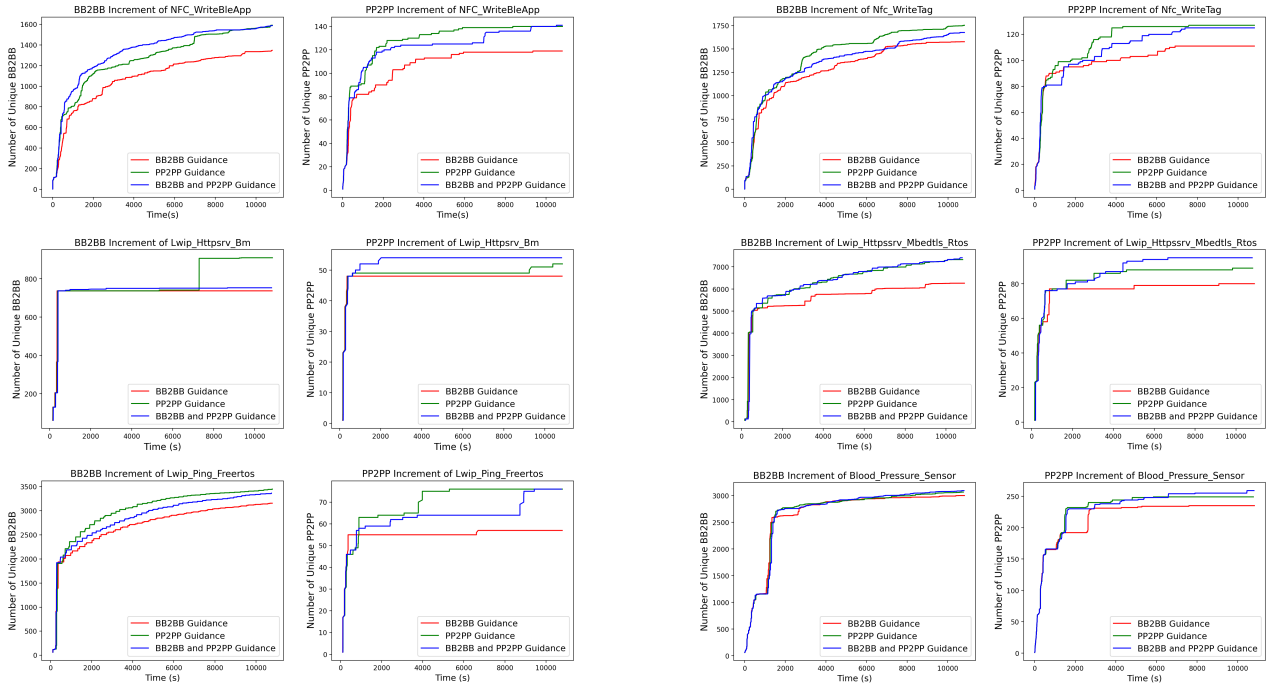

Fig. 6. Coverage growth of different feedback guidance

TABLE IV
INFLUENCE OF FAULT DETECTION PLUGINS ON VULNERABILITY DETECTION.

Firmware	Stack Tracking		Heap-Tracking				Instruction-Tracking	
	Stack Overflow	Out-of-Bound r/w	Heap Overflow	Null-Pointer Reference	Double Free	Use-After Free	Divison Zero	Integer Overflow
Lwip_Dhcp_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Httpsrv_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Iperf_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Httpssrv_Mbedtls_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Httpsrv_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Ping_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Httpssrv_MbedTLS_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Lwip_Httpssrv_Wolfssl_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Ble_Wireless_Power_Ptu_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Ble_Proximity_Reporter_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Smac_Connectivity_Test_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Smac_Wireless_Messenger_Bm	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Ble_Pluse_Oximeter_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Ble_Temperature_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Smac_Low_Power_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Smac_Wireless_Uart_Rtos	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
NFC_WriteTag	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
NFC_WriteToBleApp	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
WIFI_Client_Server	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
WIFI_Http_Server	8(1)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)	8(8)
Sum: 89.06%	**12.50%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

peripheral access points helps in expanding the exploration space.

Based on the above observations, we could positively answer **Q3** that multiple coverage feedback guidance is helpful to the growth of firmware coverage.

4) Q4: Effectiveness and performance of fault detection plugins in identifying vulnerabilities during hybrid fuzzing: The influence of fault detection mechanism is evaluated from two aspects: (1) the ability of different plugins to identify typical vulnerabilities; (2) the extra time overhead of using different plugins on firmware execution.

Ability to trigger vulnerabilities. We injected eight same vulnerabilities into one firmware image to form a new vulnerable firmware, eight types of vulnerabilities are used for injection. Thus, eight vulnerable firmware with eight different types of vulnerabilities are formed. By infecting 20 firmware images as shown in Table IV, 160 vulnerable firmware are generated. We fuzzed them for 24 hours to validate the abilities of different plugins to identify vulnerabilities. The results illustrated show that our fault detection mechanism can effectively identify common C/C++ vulnerabilities. More specially, the stack tracking is able to detect stack overflow and out-of-bound rw errors. The heap tracking can identify heap overflow, null pointer deference, double free, and use-after-free. The instruction tracking is able to detect division-by-zero and integer overflow vulnerabilities. Note that the three plugins could not be enabled at the same time for the current implementation. Among the 160 vulnerable firmware, over 89.06% of the injected vulnerabilities were identified within 24 hours. The stack overflow will stop the firmware execution in QEMU, leaving some vulnerabilities that may happen after the stack overflow untriggered.

Vulnerability discovery in real firmware. We tested two product-level firmware images (i.e., STM32-Nucleo-L152RE [26], AWS FreeRTOS with TCP [4]). The results indicate that our tool could identify the known vulnerabilities (i.e., CVE-2018-16601, CVE-2018-16603, CVE-2018-16523, CVE-2018-16524) in a few minutes.

Time overhead. We evaluated the extra time overhead of different fault detection plugins by fuzzing same firmware boundary under four settings as follows; (1) running without plugin; (2) running with stack tracking; (3) running with heap tracking; (4) running with instruction tracking. We collected the total time of finishing the same iterations for each setting, and illustrated the results in Figure V. The results indicate the extra time overhead caused by stack tracking, heap tracking and instruction tracking are 5.17%, 7.78% and 2.97% respectively.

Based on the above observations, we could positively answer **Q4** that our fault detection plugins are able to identify mainstream C/C++ vulnerabilities effectively at the expense of acceptable overhead.

TABLE V
THE INFLUENCE OF FAULT DETECTION PLUGINS ON FUZZING PERFORMANCE

Firmware	#Iter	Time Overhead(s)			
		No Plugin	Stack Tracking	Heap Tracking	Instruction Tracking
lwip_tcpecho_bm	10	2788.311	2829.197	2926.531	2808.222
bluetooth_blood_pressure_sensor_bm	10	1380.317	1384.937	1384.561	1384.228
bluetooth_wireless_power_ptu_bm	10	901.001	903.444	900.061	904.26
smac_wireless_messenger_bm	10	3261.476	3684.429	3837.802	3499.143
bluetooth_wireless_uart_freertos	10	896.229	902.913	896.785	905.469
Average:		**1845.466**	**1940.984 (+ 5.17%)**	**1989.148(+ 7.78%)**	**1900.264 (+2.97%)**

TABLE VI
RESULTS OF COMPARING FIRMHYBRIDFUZZER WITH RELATED FIRMWARE FUZZING TOOLS.

Firmware Fuzzing Tools	Hardware Independence	Source Independence	Supported Firmware Types	Supported Network Peripheral	Fuzzing Method
IoTFuzzer [44]	N	Y	Linux and MCU-based	Y	BlackBox Fuzzing
FirmAFL [30]	Y	Y	Linux-based	Y	GreyBox Fuzzing
P2IM [66]	Y	Y	MCU-based	N	GreyBox Fuzzing
HALucinator [36]	Y	N	MCU-based	Y	GreyBox Fuzzing
Laelaps [34]	Y	Y	MCU-based	N	Symbolic Execution
FirmHybirdFuzzer	Y	Y	MCU-based	Y	Hybrid Fuzzing

TABLE VII
RESULTS OF COMPARISON BETWEEN LAELAPS AND FIRMHYBRIDFUZZER. THE #ITER IS THE NUMBER OF THE MAIN TASK'S EXECUTION, T(S) IS THE TOTAL FUZZING TIME, #BB IS THE NUMBER OF UNIQUE BB2BB, #PP IS THE NUMBER OF UNIQUE PP2PP.

Firmware	#Iter	Laelaps			FirmHybridFuzzer		
		T(s)	#BB	#PP	T(s)	#BB	#PP
Lwip_Httpsrv_Rtos	50	2085.33	1757	16	590.87	1922	30
Lwip_Ping_Freertos	50	894.13	2190	3	222.61	2272	12
Lwip_Httpssrv_Mbedtls_Rtos	50	6667.07	2162	17	1095.14	2307	38
Lwip_Httpssrv_Wolfssl_Rtos	50	1853.08	2944	16	423.16	3134	24
Ble_Pluse_Oximeter_Rtos	50	18635.81	4209	140	3717.89	4353	148
Ble_Temperature_Rtos	50	18516.39	4225	137	2950.04	4411	150
Smac_Low_Power_Rtos	50	494.39	603	7	260.60	607	8
Smac_Wireless_Uart_Rtos	50	869.15	996	19	349.00	1224	27
Nfc_WriteToBleApp	50	1626.03	662	42	902.11	786	52
Wifi_Client_Server	50	2240.31	654	16	1303.94	877	35
SUM		**53881.69**	**20402**	**413**	**11815.36 (-78.07%)**	**21893 (+7.31%)**	**524 (+26.88%)**

5) Q5: Comparison between FirmHybridFuzzer with related firmware fuzzing tools.: The existing mainstream firmware fuzzing tools includes IoTFuzzer [44], Firm-AFL [30], P2IM [66], HALucinator [36] and Laelaps [34]. Due to the We compared FirmHybirdFuzzerwith these tools from five aspects: (1) Hardware-Independence, (2) Source-Independence, (3) Supported Firmware Types, (4) Supported Network Peripheral, and (5) Fuzzing Methods. The results are illustrated in Table VI. IoTFuzzer is a blackbox fuzzing tool, which relies on real devices. While the key contribution of FirmHybirdFuzzeris achieving physical-agnostic firmware fuzzing. FirmAFL focuses on fuzzing Linux-based firmware, while our FirmHybirdFuzzer's targets are MCU-based IoT firmware. P2IM could not handle firmware with network peripherals. HALucinator enables firmware execution without hardware dependence by providing generic implementations of located library functions in a full-system emulator. But, it needs the source code of the corresponding HAL SDKs to build the HAL function-matching database. Our approach achieves the same goal by providing a universal symbolic peripheral with the ability of hybrid event generation and does not assume such knowledge. Due to lots of troubles in deploying P2IM and HALucinator. We are not able to perform an experimental evaluation on same benchmarks to compare their performance with our FirmHybirdFuzzer.

We extended our previous work Laelaps [34] by: (1) formulating the problem of microcontroller-based IoT firmware fuzzing, and explaining why this problem is drastically different from Linux-based firmware fuzzing; (2) supporting network peripherals and providing more python interfaces (e.g., bypassing code, fixing peripheral value, overwriting function, etc.) to assist execution; (3) integrating mutation-based event generation and designed a hybrid approach to fuzzing IoT firmware. It improves performance by exploring multiple peripheral input spaces; (4) proposing a new coverage feedback mechanism (PP2PP) specifically for fuzzing firmware; (5) developing new heuristics as QEMU plugins for effective fault detection. We have selected a set of representative benchmarks to assess and compare the performance of our tool, referred to as FirmHybirdFuzzer, with Laelaps [34]. The comparative results are presented in Table VII, highlighting the following observations. Our tool, FirmHybirdFuzzer, demonstrates the ability to achieve higher code coverage, as evidenced by an increase of 7.31% in basic block (BB) coverage and 26.88% in peripheral access point (PP) coverage, when compared to Laelaps. Furthermore, these superior coverage metrics are obtained with a significantly reduced time overhead, with FirmHybirdFuzzerrequiring only one-fifth of the time consumed by Laelaps.

Based on the above observations, we could answer **Q5** that our hybrid fuzzing tool for IoT firmware (i.e., FirmHybirdFuzzer) has obvious differences and specific advantages compared to the related firmware fuzzing tools.

VI. Related Work and Discussion

This section reviews and discussed related work on IoT firmware analysis based on symbolic execution and fuzzing techniques, and the limitations of our approach and tool.

Symbolic execution. FIE [67] improves the symbolic execution engine KLEE to verify security properties of firmware based on source code, but it only supports MSP430 architecture. Firmalice [68] is a symbolic execution based binary analysis framework. It utilizes a novel model of authentication bypass vulnerability to detect backdoor in firmware. FirmUSB [69] uses domain knowledge of the USB protocol and symbolic execution to validate firmware against expected functionality, but it only supports the 8051 architecture. SymDisk [70] used a symbolic disk to test the file systems. SymDrive [33] is a system to test Linux and FreeBSD drivers without devices based on symbolic execution (i.e., S2E). However, our approach focus on the analysis of microcontroller-based firmware by using symbolic execution and fuzzing. Inception [71] is a KLEE based symbolic execution framework to test embedded firmware by merging LLVM bitcode, hand-written assembly, binary libraries and part of processor hardware behavior together. Laelaps [34] uses symbolic execution to model the effective behavior of unknown peripherals, and aims to speculatively selects the most "promising" path so as to initialize a context that is suitable for further analysis. We extended Laelaps [34] by: (1) formulating the problem of microcontroller-based IoT firmware fuzzing, and explaining why this problem is drastically different from Linux-based firmware fuzzing; (2) supporting network peripherals and providing more python interfaces (e.g., bypassing code, fixing peripheral value, overwriting function, etc.) to assist execution; (3) integrating mutation-based event generation (note the Laelaps only use symbolic execution) and designed a hybrid approach to fuzzing IoT firmware. It improves performance by exploring multiple peripheral input spaces; (4) proposing a new coverage feedback mechanism (PP2PP) specifically for fuzzing firmware; (5) developing new heuristics as QEMU plugins for effective fault detection. DICE [35] proposes a drop-in solution for firmware analyzers to emulate DMA input channels and generate or manipulate DMA inputs.

Fuzzing. RPFuzzer [43] is a framework designed to discover router protocol vulnerabilities by sending packets to devices, keeping watch on CPU utilization and checking system logs. [26] demonstrates that memory corruption vulnerabilities often result in different behaviors on embedded devices, and illustrates the influence on security analysis of firmware. IoTFuzzer [44] is an App-based black-box fuzzer to detect memory corruption flaws of IoT firmware without access to the images. Avatar [31] and avatar2 [32] enable the dynamic analysis of embedded devices by orchestrating the execution of an emulator together with devices. They leverage the real peripherals to handle I/O operations, but emulate the firmware in an emulator and forwarded the I/O accesses from the emulator to the real devices. For hardware that is not available, they implement the corresponding software abstraction to act as the real peripherals. FIRMADYNE [29] utilizes a built-in modified Linux kernel to enable the QEMU's full system emulation of Linux-based firmware. Based on it, FIRM-AFL [30] proposed a high-throughput greybox fuzzer for POSIX-compatible firmware. It could only fuzz Linux-based firmware. But a large number of microcontroller-based firmware runs lightweight RTOS or bare-metal systems, which can be addressed by our hybrid fuzzing system. P2IM [66] use an abstract model for a target microcontroller architecture class, which defines the access patterns of different peripheral registers, to build an approximate emulator for firmware execution and testing. However, the construction of the abstract model relies on manual work and expert knowledge. Different from P2IM, our approach enables the automatic execution and fuzzing of firmware by using a unified symbolic peripheral with the ability of hybrid event generation. HALucinator [36] provides a high-level replacement for HAL functions to decouple the hardware from the firmware, and enables the re-hosting and analysis of firmware by providing generic implementations of identified library functions in a full-system emulator. Our approach achieves the same goal by providing a universal symbolic peripheral with the ability of hybrid event generation. But, HALucinator needs the source code of the corresponding HAL SDKs to build the HAL function matching database, while our approach does not assume such knowledge.

Limitaion. Our tool exclusively supports Cortex-M3/M4-based MCU firmware and is not compatible with Linux-based firmware. The inclusion of the data sheet for the MCU is essential in order to define the configuration files, and manual analysis is required for certain aspects. Moving forward, we endeavor to expand our research to encompass a wider range of MCU types and peripherals.

VII. Conclusion

We have devised and implemented a physical device-agnostic hybrid fuzzing system, specifically designed to perform fuzzing on microcontroller-based firmware without the need for actual devices. To achieve this, we have incorporated a unified symbolic peripheral into the emulator, allowing for the emulation of unknown peripherals' behaviors. Consequently, the firmware execution becomes independent of the specific physical devices used. Our approach employs a hybrid methodology that combines symbolic execution with greybox fuzzing to generate values for various peripheral accesses. To optimize the utilization of symbolic execution and fuzzing, we propose probability-based scheduling strategies. Additionally, we have developed multiple coverage feedback mechanisms, including unique transitions from one basic block to another and from one peripheral access point to another. These coverage feedbacks are collected and utilized to guide the fuzzing process, which is facilitated by a genetic algorithm. Furthermore, we have introduced a fault detection mechanism to identify common vulnerabilities in C/C++ code and successfully implemented it. To demonstrate the effectiveness and efficiency of our hybrid fuzzing system, we have conducted a comprehensive set of experimental evaluations using a benchmark.

ACKNOWLEDGMENTS

This work was (partially) supported by the Natural Science Foundation of China (No.62032010), NSF CNS-2019340, National Social Science Foundation of China (No.22CTQ029), Ministry of Education Philosophy and Social Science Research Foundation of China (No.21JHQ014), Jiangsu Social Science Foundation (No.21TQC004), Jiangsu Province Frontier Leading Technology and Basic research project (No.BK20202001), Doctoral of Entrepreneurship and Innovation Doctoral Program of Jiangsu Province (No.JSSCBS20210032), Science and Technology Innovation Program of Nanjing Scientists Studying Abroad (No.126), Fundamental Research Funds for the Central Universities (No.17).

REFERENCES

[1] R. S. Sinha, Y. Wei, and S.-H. Hwang, "A survey on lpwa technology: Lora and nb-iot," *Ict Express*, vol. 3, no. 1, pp. 14–21, 2017.
[2] S. Li, L. Da Xu, and S. Zhao, "5g internet of things: A survey," *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 2018.
[3] G. Beniamini, "Over the air: Exploiting broadcom's wi-fi stack (part 1).(2017)," 2017.
[4] O. Karliner, "Freertos tcp/ip stack vulnerabilities the details:https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/," 2018.
[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[6] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.
[7] http://lcamtuf.coredump.cx/afl/, "Afl."
[8] L. C. Infrastructure, "libfuzzer: a library for coverage-guided fuzz testing," 2017.
[9] R. Swiecki, "Honggfuzz," *Available online at: http://code. google. com/p/honggfuzz*, 2016.
[10] https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous fuzzing.html, "Google oss fuzzing," 2016.
[11] https://www.microsoft.com/en-us/security-risk detection/, "Microsoft security risk detection," 2016.
[12] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 26–47.
[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.
[14] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 679–696.
[15] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.
[16] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.
[17] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.
[18] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.
[19] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.
[20] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium*, 2018, pp. 745–761.
[21] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: complexity analysis with fuzzing and symbolic execution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 322–332.
[22] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.
[23] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.
[24] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 95–105.
[25] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: a gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 138–145.
[26] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, 2018.
[27] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 2015, p. 4.
[28] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.
[29] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016, pp. 1–16.
[30] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," 2019.
[31] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, 2014, pp. 1–16.
[32] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Workshop on Binary Analysis Research (colocated with NDSS Symposium)(February 2018), BAR*, vol. 18, 2018.
[33] M. J. Renzelmann, A. Kadav, and M. M. Swift, "Symdrive: testing drivers without devices," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation)*, 2012, pp. 279–292.
[34] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Annual Computer Security Applications Conference*, 2020, pp. 746–759.
[35] A. Mera, B. Feng, L. Lu, E. Kirda, and W. Robertson, "Dice: Automatic emulation of dma input channels for dynamic firmware analysis," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, May 2021.
[36] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium*, 2020, pp. 1–18.
[37] P. Amini and A. Portnoy, "Sulley-pure python fully automated and unattended fuzzing framework," *May*, 2013.
[38] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, p. 34, 2011.
[39] J. Pereyda, "boofuzz: Network protocol fuzzing for humans," *Accessed: Feb*, vol. 17, 2017.
[40] https://aflplus.plus/, "Afl++."
[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[42] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.
[43] Z. Wang, Y. Zhang, and Q. Liu, "Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing." *KSII Transactions on Internet & Information Systems*, vol. 7, no. 8, 2013.
[44] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," *Proc. 2018 NDSS, San Diego, CA*, 2018.
[45] FreeRTOS, "https://www.freertos.org/," 2019.
[46] MbedOS, "https://www.mbed.com/zh-cn/platform/mbed-os/," 2019.
[47] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering*. IEEE, 2007, pp. 416–426.
[48] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[49] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, 2018.

[50] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[51] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *arXiv preprint arXiv:1709.07101*, 2017.

[52] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[53] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[54] S. Jha, R. Limaye, and S. A. Seshia, "Beaver: Engineering an efficient smt solver for bit-vector arithmetic," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 668–674.

[55] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[56] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development*. IEEE, 2017, pp. 8–9.

[57] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for {OS} kernels," in *26th {USENIX} Security Symposium*, 2017, pp. 167–182.

[58] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings*, 2019, pp. 20–22.

[59] S. Mirjalili, "Genetic algorithm," in *Evolutionary Algorithms and Neural Networks*. Springer, 2019, pp. 43–55.

[60] T. Goodspeed and S. Bratus, "Facedancer usb: Exploiting the magic school bus," in *Proceedings of the REcon 2012 Conference*, 2012.

[61] Y. Zhang, Z. Chen, and J. Wang, "Speculative symbolic execution," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 101–110.

[62] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 246–256.

[63] C. C. Conley and E. Zehnder, "The birkhoff-lewis fixed point theorem and a conjecture of vi arnold," *Inventiones mathematicae*, vol. 73, no. 1, pp. 33–49, 1983.

[64] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 110–121.

[65] ChibiOS, "http://www.chibios.org/dokuwiki/doku.php," 2019.

[66] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[67] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "{FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Presented as part of the 22nd {USENIX} Security Symposium*, 2013, pp. 463–478.

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

[69] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2245–2262.

[70] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.

[71] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: system-wide security testing of real-world embedded systems software," in *27th {USENIX} Security Symposium*, 2018, pp. 309–326.