Received January 25, 2020, accepted February 4, 2020, date of publication February 10, 2020, date of current version February 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2973043

FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution

ZHIJIE GUI, HUI SHU, FEI KANG, AND XIAOBING XIONG
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
Corresponding author: Zhijie Gui (zakiguixyz@gmail.com)

ABSTRACT The security situation of the Internet of Things (IoT) is particularly severe, and a large number of IoT devices are prone to vulnerabilities. In this study, we present FIRMCORN, the first vulnerability-oriented fuzzer for IoT firmware. Based on the novel technology of optimized virtual execution, FIRMCORN focuses on three typical problems of IoT firmware fuzzing: (1) high throughput required by fuzzing, (2) inaccuracy of emulation compared with real devices, and (3) instability of emulation due to lack of hardware. Here, we optimize the initial environment and the execution process of virtual execution to achieve faster, more accurate, and more stable fuzz testing. To improve the efficiency of vulnerability mining with FIRMCORN, a vulnerable-code search algorithm is designed to obtain the entry points of fuzzing according to the characteristics of IoT firmware; further, this vulnerability-oriented fuzzing is applied to IoT device firmware. Our evaluation results show that optimized virtual execution used by FIRMCORN can significantly improve the throughput, accuracy, and stability compared with conventional virtual execution. FIRMCORN runs for only 2 hours to mine two 0-day vulnerabilities on a machine. Thus, compared with the current state-of-the-art IoT firmware fuzzing framework, FIRMCORN can more effectively mine vulnerabilities in real-world devices.

INDEX TERMS IoT, firmware, fuzzing, vulnerability, CPU emulator.

I. INTRODUCTION

In recent years, various Internet of Things (IoT) devices have begun to access the Internet on a large scale, profoundly changing people's lifestyles; the number of IoT devices is expected to exceed several times the global population by 2022 [1]. Despite the explosive growth in the number of IoT devices, manufacturers are yet to prioritize security issues in such devices. Attackers can exploit vulnerabilities in IoT firmware and control the IoT devices and even the entire IoT system, because it often comprises a large number of identical devices. Common IoT devices include routers, IP cameras, and network-attached storage, and these devices function more powerful compared with traditional embedded devices in a network. Due to the limited resources of IoT devices, the protection mechanisms commonly used in desktop devices, such as Address Space Layout Randomization (ASLR) [2] and Stack Canary [3], are not widely used for IoT devices; therefore, exploitation of IoT devices becomes even easier.

IoT firmware architecture is diverse and device dependent, so firmware analysis and vulnerability mining are difficult to research. For example, FIE [4] can only support automated analysis of firmware using MSP430 microcontrollers. FIRMADYNE [5] can analyze firmware based on full-system emulation, but NVRAM emulation failures often cause mining process crashes. IoTFuzzer [6] can only test security for App-based IoT devices.

Fuzz testing, an effective test method for discovering security vulnerabilities, has been widely used in industry and academia and has discovered many real-world vulnerabilities. The current research trend is to apply fuzz technology to quickly and effectively discover vulnerabilities in IoT device firmware. For example, FIRM-AFL [7] is based on FIRMADYNE, which applies AFL [8] to IoT firmware vulnerability mining through greybox fuzzing. However, this tool can only test firmware that FIRMADYNE can emulate.

Many problems exist in applying fuzz technology to IoT firmware. First, because accurate testing of IoT firmware needs to be based on real devices, large-scale parallel testing requires considerable real hardware. The scheme based on emulation is computing-resource intensive and uses an emulation environment different from the actual device operating environment. Moreover, due to the hardware dependence of device firmware, emulation sometimes crashes when encountered with lack of hardware.

The associate editor coordinating the review of this manuscript and approving it for publication was Cong Pu.

We also note that greybox fuzzing that leverages code coverage is not suitable for IoT firmware for the following reasons: First, only a small part of the firmware constitutes vulnerable codes; therefore, in spite of the higher code coverage of the fuzzer, most of the mined codes are not vulnerable. Second, the firmware initialization process involves a large amount of hardware interaction logic, but this part of the code cannot be exploited remotely even if there is vulnerability. Therefore, a more efficient fuzz testing technique is expected.

Our Solution: Vulnerability-oriented fuzzing on IoT firmware via optimized virtual execution. In this study, we aim to address the following general problems in firmware fuzzing: (1) high-throughput requirement for fuzzing (2) inaccuracy of emulation compared with real devices (3) instability due to emulation crashes caused by lack of hardware.

We propose optimized virtual execution, with the main idea of optimizing the virtual execution initial environment and the execution process. More specifically, we improve emulation accuracy by incorporating the real context of the actual device, system throughput by using heuristic algorithms to skip unnecessary functions of fuzzing, and fuzzing stability by hooking hardware dependency functions. We design a vulnerable-code search algorithm to determine the vulnerabilities of firmware for vulnerability-oriented fuzz testing.

In this article, we present FIRMCORN, a vulnerability-oriented IoT firmware fuzzing framework based on optimized virtual execution. From the user's viewpoint, FIRMCORN can be used to dump runtime context information on real devices, perform automatic analysis on firmware codes to obtain vulnerable parts, and efficiently and stably conduct fuzz testing on IoT firmware of various architectures in a CPU emulator.

We tested FIRMCORN with a benchmark program and 10 IoT devices, and the following results were obtained: (1) Optimized virtual execution technology used by FIRMCORN could improve the efficiency and stability of fuzzing; (2) The vulnerable-code search algorithm could effectively locate vulnerable parts of the firmware; and (3) FIRMCORN could effectively mine undisclosed vulnerabilities and detect two 0-day vulnerabilities in real IoT devices after running for 2 hours on a machine. We also provide a case study to demonstrate the application of FIRMCORN to perform fuzz testing on real devices.

Contributions: In summary, this study provides the following contributions:

- We summarize existing methods for analyzing IoT firmware. Most existing methods do not sufficiently solve the typical problems of firmware fuzzing; therefore, we propose a novel technology called optimized virtual execution, and use it as the basis for firmware fuzzing.
- We propose and implement a vulnerable-code search algorithm that performs static analysis on IoT firmware to obtain vulnerable parts.
- We design and implement FIRMCORN, an IoT firmware fuzzing framework based on optimized virtual execution, and apply vulnerability-oriented fuzzing to IoT firmware for the first time.
- The results of our extensive evaluation of the effectiveness of FIRMCORN indicate that FIRMCORN can significantly improve speed and stability compared with the conventional virtual execution technology, and FIRMCORN could discover two 0-day vulnerabilities in real IoT devices within 2 hours.
- For supporting open science, our framework will be open source to help other researchers in the in-depth study of IoT firmware fuzzing.(https://github.com/FIRMCORN-Fuzzing/FIRMCORN)

II. BACKGROUND

A. IoT FIRMWARE

Firmware refers to a binary program that exists in an EEPROM or a FLASH chip. There are two types of firmware: low-level firmware and high-level firmware. Low-level firmware mainly exists in an EEPROM, and it is difficult to modify or update; high-level firmware usually resides in Flash. Firmware works between the underlying hardware and the upper layer software, and provides a simple call interface for the software by effectively managing the hardware devices.

Firmware mainly includes the firmware header, bootloader, system kernel, and file system, and due to the limited computing resources and storage space of IoT devices, firmware is often burned in the device in a compressed form. For performing firmware program analysis to understand the device behavior and mine its potential vulnerabilities, a debugging environment needs to be set up using static analysis and dynamic debugging.

B. INTRODUCTION TO FUZZING

Fuzzing is an automated software vulnerability mining technology whose basic idea is to input a large amount of malformed data to the target software to be tested and repeatedly drive the target program to run while monitoring the running status of the program. If an abnormal situation such as a program crash occurs, testers analyze crash samples and error locations to detect exploits or improve software quality. Fuzz testing technology can be classified into whitebox, blackbox, and greybox fuzzing according to the mastery of target program behavior and information.

Whitebox fuzzing [9] analyzes the program before testing and obtains certain information to guide the input generation. With the understanding of the program structure and input conditions, whitebox fuzzing is a targeted process and avoids generation of many useless test samples. However, due to the complexity of the target program execution process, the preliminary analysis requires considerable time and resources. Blackbox fuzzing [10] is a simple and effective test solution. Unlike whitebox fuzzing, blackbox fuzzing does not consider the internal state and execution flow of the program but directly analyzes a large number of random inputs to test the target program. Blackbox fuzzing is simple and effective but can generate a lot of useless test cases. Following the introduction of whitebox and blackbox fuzzing, some researchers proposed greybox fuzzing [11], which uses a lightweight analysis method to guide the fuzz test and provides results more efficiently than whitebox fuzzing and with greater accuracy than blackbox fuzzing.

Coverage-based fuzzing has mined many real-world vulnerabilities, it is considered effective in practice. However, in-depth research of fuzzing has suggested that a fuzzing strategy that simply improves code coverage is inefficient because vulnerable codes account for only a small proportion of the entire codes [12]. More efficient strategies should determine the vulnerable part of the target program and then execute targeted testing.

III. OVERVIEW

A. MOTIVATION

Debugging and analysis of IoT firmware is the basis for fuzz testing. In this section, we discuss the advantages and problems of the existing approaches and illustrate our solution.

Existing Approaches. We summarize below the IoT firmware analysis approaches that are currently available:

1) **Hardware Interface Debugging.** The debugging method directly debugs an IoT device through its hardware interface, such as the `UART` or `JTAG` debug interface [13]. This method is accurate and reliable. However, it relies on physical equipment and can only debug the equipment that exists and is retained in the debugging interface. In addition, the analysis process greatly depends on manual execution, which is not conducive to large-scale automated analysis.
2) **Full Static Analysis.** [14] The full static analysis method is the most direct and simple way, requiring complete and full decompression of the firmware, and it is unrelated to a single program or multiple programs. This method is suitable for large-scale and parallel analysis and can collect a large number of firmware for automated testing. Static analysis does not actually run on firmware, so there is no external environment interaction or missing NVRAM as in dynamic debugging. However, this method has an obvious limitation that the actual behavior in the firmware operation cannot be accurately analyzed, resulting in a large number of false positives.
3) **User-Mode Emulation.** User-mode emulation technology is a simple and fast dynamic debugging method. By decompressing firmware, a root file system is obtained, and a single program in the firmware can be run using the QEMU [15] user mode. It can also cooperate with `chroot` command to change the location of the root directory referenced during program execution in order to solve the problem of missing dynamic link library files during runtime under the complete root file system. Further, the QEMU supports remote debugging, wherein a remote target can be set by gdb and its firmware can be effectively debugged with QEMU. However, in this method, the firmware program may perform many hardware device-based initialization operations at startup, and because emulation of the hardware devices is not possible, the process often crashes at startup.
4) **Full-system Emulation.** FIRMADYNE [5] is based on full-system emulation and provides the user with a web interface. This method is based on QEMU's system mode; it collects system architecture information according to busybox using a firmware extraction file system, matches the customized system kernel, and then runs the firmware using the QEMU system mode. The configuration file of the device is generally stored in NVRAM, and the NVRAM values can be read in the firmware through functions similar to `nvram_get`. If NVRAM is missing, these functions will crash during the emulation. To solve the problem of missing NVRAM, FIRMADYNE customizes the libnvram.so file, load it through LD_PRELOAD, and hijack the call of an NVRAM-related function. However, in this method, the whole system emulation is extremely difficult, with large time overhead, thereby limiting the adaptation of the method to large-scale test analysis. Moreover, this method has a low success rate and is unstable during fuzzing, because if the program calls the libnvram.so file without an emulated function, the emulation immediately crashes.
5) **Augmented Process Emulation.** This technology augments user-mode emulation with full-system emulation and attempts to solve the firmware operation compatibility problem by using system-mode emulation and the low throughput problem by user-mode emulation. FIRM-AFL is based on this method. However, the full-system emulation component of this method itself cannot completely solve the hardware dependency function problem existing in firmware. If the method encounters an external function without emulation, the process will crash. Moreover, the switching between the user mode and system mode will cause performance overhead.
6) **Multi-target Orchestration Analysis.** Multi-target orchestration analysis is based on both real devices and emulation technology and strives to solve the accuracy problem that cannot be solved by emulation and the automatic analysis problems that cannot be solved by hardware debugging. Avatar [16] and Avatar^2 [17] are concrete examples of this concept, trying to forward I/O operations to real devices to solve external environment interaction problems. However, the Avatar and Avatar^2 schemes have the problem that switching between emulation environment and real devices affects the running speed in large-scale fuzz testing.

Although existing approaches play an important role in firmware debugging, the existing approaches still have problems in terms of speed, accuracy, and stability; therefore, they cannot be used as the basis of IoT firmware fuzzing. In order to solve the three typical problems of IoT firmware fuzzing, we propose novel techniques in the design process, namely optimized virtual execution and a vulnerable-code search algorithm, and implement the FIRMCORN framework to achieve faster, more accurate, and more stable IoT firmware fuzzing.

Before the fuzz testing process starts, FIRMCORN first analyzes the firmware using the vulnerable-code search algorithm and determines the entry point of fuzzing, runs to the entry point in the actual device, and dumps context information of the location as the initial state of fuzzing. Then, FIRMCORN sets up the registers and memory layout at the entry point in the CPU emulator. It then uses heuristic algorithms to collect functions that cannot be emulated or do not need to be emulated before fuzzing. These include hardware-dependent functions, functions that read and write to dynamically allocated memory space, and functions that are not necessary for fuzz testing; these are hereafter called hardware-specific, unresolved, and unnecessary functions, respectively. Finally, starting from the entry point, hooks are added to the above functions or filters to start fuzz testing for vulnerable codes.

Optimized Virtual Execution: Virtual execution technology does not actually execute firmware, but uses a technique to read and execute some firmware instructions through a CPU emulator. This lightweight emulation solution avoids large time overhead caused by full-system emulation. Conventional virtual execution based on a CPU emulator is lightweight but not accurate and stable, and some output functions such as `puts` are not necessary for fuzz testing. Optimized virtual execution optimizes the initial environment of virtual execution by using the real IoT device dump context. Heuristic algorithms are used to search the three types of functions to optimize the virtual execution process, thereby achieving faster, more accurate, and more stable virtual execution. In the implementation of FIRMCORN, we adopt optimized virtual execution as the basis for fuzz testing.

B. CHALLENGES IN FIRMCORN DESIGN

In this section, we summarize the challenges encountered in the design of FIRMCORN and provide their solutions.

1) CHALLENGES

In the implementation of FIRMCORN, the following challenges need to be resolved.

2) ENTRY POINT GENERATION

IoT device firmware contains a number of hardware-dependent functions that cannot be remotely interacted with and effectively exploited even if they contain vulnerabilities; therefore, this area of code needs to be omitted from fuzzing. At the same time, virtual execution itself requires computing resources, and the coverage-based fuzzer constantly generates new inputs to determine new paths, making it an extremely inefficient test strategy. The entry point of fuzzing is key information that must be obtained in the framework design and it affects fuzzing efficiency and the ability of vulnerability mining.

3) LARGE-SCALE TESTING

In the process of fuzzing, large-scale repeated testing is often performed to improve the accuracy of test results. However, firmware operation is closely dependent on the device; because large-scale testing requires a large number of devices, fuzz testing may frequently lead to program execution crash and frequent restart, resulting in considerable runtime. A framework must be designed to effectively test IoT device firmware within a limited time frame with acceptable hardware and software resources.

4) AUTOMATED FUNCTION PROCESSING

During the virtual execution of firmware, functions that need to be processed are encountered. For example, in actual firmware testing, a large number of hardware-specific functions exist, which may lead to the crashing of virtual execution. Manual annotation of the addresses of such functions is time consuming. Hence, the framework design must consider automation of these functions to ensure stable and fast fuzz testing.

5) SOLUTIONS

To resolve the aforementioned problems, we provide the corresponding solutions:

6) VULNERABLE-CODE SEARCH ALGORITHM

We use a static analysis method to design and implement a vulnerable-code search algorithm. Based on IDAPython's advanced APIs, we calculate the cyclomatic complexity of functions and the number of references as the complexity index to obtain complexity groups. In each group, we calculate the number of memory operations and the number of sensitive function calls as the vulnerability feature index to obtain the order of vulnerability functions; then, we use the initial address of the function as the entry point of fuzzing.

7) CONTEXT DUMP

In the design process of FIRMCORN, we adopt the method of dumping context information to realize a collection and multiple tests. To achieve virtual execution accuracy, we dump the context information in real IoT devices, recover the complete context information, and set registers and memory layout before virtual execution.

8) HEURISTIC HOOK

We implement an automated processing mechanism for different types of functions based on heuristic algorithms. We coordinate the use of two hook techniques: Global Offset Table Hook (GOT-based) and Exception Hook (exception-based). Using these hooks, we can automatically replace and emulate unresolved and hardware-specific functions and skip unnecessary functions.

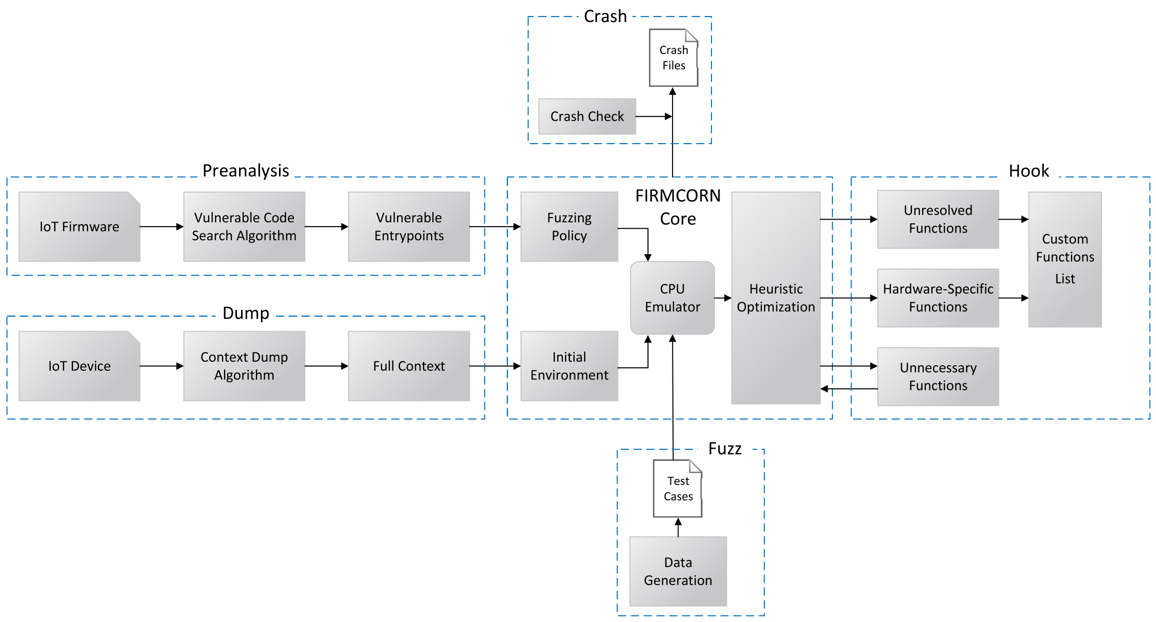


FIGURE 1. Overview of FIRMCORN.

IV. DETAILED DESIGN

In this section, we introduce the design and implementation details of FIRMCORN. One of FIRMCORN's design principles is modularity programming. According to different functions, we can divide the framework into five submodules: preanalysis, dump, hook, fuzz, and crash. These five submodules are called by the FIRMCORN Core. We illustrate the detailed design of FIRMCORN in Figure 1.

A. PREANALYSIS

The algorithm can be divided into two stages: complexity grouping and vulnerability-feature ranking stages.

1) FIRST STAGE: COMPLEXITY GROUPING STAGE

In the complexity grouping stage, the algorithm classifies all the functions in the firmware on the basis of the complexity index to obtain complexity grouping lists. For a function, the higher its complexity, the more complex the function logic is and the higher the probability of vulnerabilities in the function is. At the same time, in order not to miss the vulnerabilities in a low-complexity function, we group all functions and judge the vulnerability characteristics in each group. The complexity of a function is reflected in two aspects: the complexity of the logic of the function itself and the complexity of the reference relationship, which are measured, respectively, by cyclomatic complexity [18] and the number of times the function is called.

a: CYCLOMATIC COMPLEXITY

We calculate the cyclomatic complexity of a function according to the number of points and edges of the function control flow graph (CFG), according to the following formula:

$$M_F = E - N + 2 \times P \tag{1}$$

where M_F represents the cyclomatic complexity of function F; E represents the number of edges in the CFG; N represents the number of nodes in the graph; and P represents the number of connected components.

b: NUMBER OF TIMES A FUNCTION IS CALLED

The number of times a function is called can intuitively reflect the complexity of the function call relationship. If a function is vulnerable and is called multiple times, it implies that there are multiple ways to trigger its vulnerability. The number of times a function is called, X_F, is calculated as follows.

$$X_F = \sum_{i=1}^{n} XrefsTo(here)_i$$

where *XrefsTo*, which takes the function start address *here*, returns a list of callers. In the calculation of this value, we ignore the function that has been called 0 times in the target program. In the implementation of the vulnerable-code search algorithm, we define the functional complexity index, $complex_F$, as follows.

$$\begin{aligned} complex_F &= \lfloor M_F \ln M_F \rfloor \\ &= \lfloor (E - N + 2 \times P) \ln(E - N + 2 \times P) \rfloor \\ &\quad + \sum_{i=1}^{n} XrefsTo(here)_i \end{aligned}$$

This index can ignore the effect of a small circle complexity on the index when the cyclomatic complexity of a function is less than 3; it can enhance the effect of the cyclomatic

TABLE 1. Sensitive functions.

Function Type	Function list	Description of function
String operations	strcpy(char*dest, const char *src)	String copy
	strcat(char*dest, const char *src)	String concatenate
Data input	read(int fd, void *buf, size_t count)	Read string
	scanf(const char * restrict format,...)	Read content from stdin
	gets(char * str)	Read string from stdin
	read(int fd, void *buf, size_t count)	Read string
	gets(char * str)	Read string from stdin
Format string	sprintf(char *string, char *format [,argument,...])	Writes formatted data to buffer
Command execution	system(char *command)	Execute a command
	execve(char *path,char *argv[],char * envp[])	Start the program in a child process
Environment variables	getenv(char *envvar)	Get environment variables

complexity on the index when the cyclomatic complexity has reference significance and ensure the result to be an integer by rounding down. Next, in this stage, we group the functions of the firmware program with the same complexity into a group and thus obtain the function complexity groups.

Algorithm 1 Vulnerable-Code Search Algorithm

```
Input: Firmware functions $funcs$
Output: Vulnerability feature rank $vulnerable\_rank$
1:  Initialize complex group $Group1$, vulnerable rank $Group2$
2:  // stage1: complexity grouping stage
3:  for $i = 0$ to $len(funcs)$ do
4:      $complex_i = GetFuncComplex(i)$
5:      if $complex_i == 0$ then
6:          continue
7:      end if
8:      $Group1[complex_i] = Group1[complex_i] \cup i$
9:  end for
10: // stage2: vulnerability feature ranking stage
11: Initialize ordering rules $cmp$
12: for $i = 0$ to $len(Group1)$ do
13:     for $j = 0$ to $len(Group1[i])$ do
14:         $vulnerability_j = GetFuncVuln(j)$
15:         if $vulnerability_j == 0$ then
16:             continue
17:         end if
18:         $cmp[i] = cmp[i] \cup vulnerability_j$
19:     end for
20:     $Group2[i] = sort(Group1[i], cmp[i])$
21: end for
22: $vulnerable\_rank = Group2$
23: return $vulnerable\_rank$
```

2) SECOND STAGE: VULNERABILITY-FEATURE RANKING STAGE

In the vulnerability-feature ranking stage, the algorithm sorts the functions in each group based on the vulnerability feature index and determines the most vulnerable function in each group. The vulnerability feature of a function can be reflected in two aspects: the sensitivity function call index and the number of memory operations.

a: SENSITIVITY FUNCTION CALL INDEX

The call of sensitive functions is the most intuitive vulnerability feature. For example, the function `system`, which executes system commands, can directly execute high-risk vulnerabilities if its parameters are controllable. In the implementation process, we maintain a list of sensitive functions, as shown in Table 1.

In addition, FIRMCORN is extensible, enabling users to add sensitive functions according to specific conditions. For example, we add a sensitive function `CGI_Find_Parameter` to analyze QNAP NAS firmware; this function is defined in the file `libulinux_cgi.so.0.0` to obtain HTTP request data. FIRMCORN also supports users to assign different weights to different sensitive functions depending on the value of the functions. The sensitive function call index S_F is calculated as follows:

$$S_F = \sum_{i=1}^{n} w_i \times SenFunc(F)_i$$

where w_i represents the weights of different sensitive functions; $SenFunc(F)$ represents a function of the list of sensitive function calls of F.

b: NUMBER OF MEMORY OPERATIONS

Incorrect memory operations can often lead to Out-of-Bounds Read and Write. If used properly, it can cause serious consequences such as information leakage or arbitrary write. We obtain the disassembly result of a function, then traverse all the instructions and obtain the *OpList* set of each function as follows:

$$OPList = \{GetOPType(ea) \mid ea \in (func_{start}, func_{end})\}$$

Then, we judge whether the instruction has memory operation and constitutes a memory operation instruction set *MemOPList*. In the algorithm, we mark the number of memory operations of a function F as P_F and calculate it as follows:

$$\begin{cases} MemOPList = \{op \mid op \in OPList \ and \ op == op_{mem}\} \\ P_F = \dfrac{\sum_{i=1}^{n} MemOPList_i}{len(OPList)} \end{cases}$$

The number of memory operations is a more significant index of vulnerability features than the sensitive function call index. The algorithm defines the vulnerability feature index $vulnerability_F$ of a function F as the weighted sum of two values, and finally ignores those functions whose vulnerability features are 0. We describe the vulnerable-code search algorithm in detail in Algorithm 1.

B. DUMP CONTEXT

Optimized virtual execution ensures the accuracy of the emulation environment by dumping the context of a firmware program running in an actual device.

In the analysis of IoT devices, accessing the inside of the device for more detailed security analysis is often not possible. For example, the D-Link DIR series of routers provide users with only one web interface, facilitating some simple configuration work. In this case, we cannot gain access to the current router process and port information; therefore, we obtain the shell of the device through one of the following three ways and execute the system commands.

1) BASED ON TELNET/SSH SERVICE

If the device itself provides the `Telnet` or `SSH` service at boot time, we can easily get the shell of the device and execute system commands.

2) BASED ON DEVICE DEBUG INTERFACE

After the hardware analysis of the device, the `Universal Asynchronous Receiver/Transmitter(UART)` interface is found and the `RXD`, `TXD`, and `GND` pins are pulled out to set the appropriate baud rate. We can generally obtain the shell of the device through the `UART` port.

3) BASED ON FIRMWARE UPDATE MECHANISM

IoT devices are generally facilitated with a means to update the firmware. If the firmware is not validated during the update process, the firmware to be upgraded can be extracted, the startup script `rcS` is modified, and the `Telnet` or `SSH` service is provided when the device is turned on. The firmware is repackaged and the modified firmware is finally updated to the device.

After obtaining the device shell, we upload a statically linked `gdbserver` to the device over the network, then specify the debug port on the device side through `gdbserver`, connect the remote debug port on the host side through `gdb`, and then run to the entry point to begin preparing the dump context.

We define the complete context as a combination of the register status set *Regs*, the memory segment set *Segs*, and the architecture information *Arch*:

$$Context = Regs \cup Segs \cup Arch$$

In the FIRMCORN design process, we implement the complete context dump algorithm in the dump submodule, as presented in Algorithm 2:

In FIRMCORN, the dump submodule uses Algorithm 2 to obtain the context *Context* of the entry point location,

Algorithm 2 IoT Firmware Context Dump Algorithm

```
Input: IoT device firmware runtime state
Output: Context information file $dump.json$ and segments
    packed files $seg.bin$
 1: Initialize register status list $reg\_state$, memory segment
    list $seg\_state$
 2: Initialize architecture $arch$
 3: for $reg = 0$ to $len(Multi\_Arch\_Regs_{arch})$ do
 4:     $reg\_val = GetRegister(reg)$
 5:     $reg\_state[reg] = reg\_val$
 6: end for
 7: for $seg = 0$ to $len(Vmmp\_Segs)$ do
 8:     $seg\_content = \text{ReadMemory}(seg)$
 9:     $packed\_seg\_content = \text{Compress}(seg\_content)$
10:     Write($packed\_seg_content$) to $seg.bin$
11:     $seg\_state[seg] = packed\_seg\_content$
12: end for
13: Write($reg\_state$,$seg\_state$,$arch$) to $dump.json$
14: return $dump.json$, $seg.bin$
```

including the *dump.json* file and multiple memory segment compressed files. The *dump.json* file includes the architecture information and the register status set.

C. HOOK

We implement function hijacking and skipping in the dump submodule.

Before starting the emulation, the framework analyzes the GOT information of the firmware program. After setting the initial environment of the CPU, we traverse the GOT table and read the memory for the address of each entry of the GOT to obtain the actual address of the function. However, due to the lazy binding mechanism of the ELF [19], the binding of the address will not be complete for some functions in the GOT. Next, by parsing the symbol table information of the dynamic link library file in the firmware, the offset of the library function in the dynamic link library can be obtained. This allows calculation of the actual address of all functions in the memory, as described below.

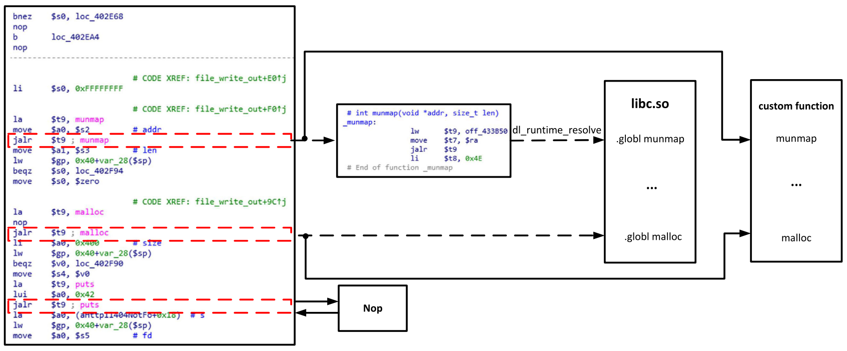
Select a function that is denoted as *funcX*; the function's address in memory is defined as mem_addr_{funcX}, and the offset address of *funcX* in the dynamic link library is $offset_{funcX}$. Hence, we obtain the actual load address of the dynamic link library file in memory, $libc_{addr}$, as follows:

$$libc_{addr} = mem\_addr_{funcX} - offset_{funcX}$$

For any function *func_i*, if the function address is not resolved, we calculate the actual address of the function in memory by the following formula:

$$mem\_addr_{func\_i} = libc_{addr} + offset_{func\_i}$$

With the above calculation, we obtain both the actual memory address list of functions GOT_{mem} and the unresolved



```
bnez    $s0, loc_402E68
nop
b       loc_402EA4
nop
-----------------------------------------------------------

                        # CODE XREF: file_write_out+E0↑j
li      $s0, 0xFFFFFFFF

                        # CODE XREF: file_write_out+F0↑j
la      $t9, munmap
move    $a0, $s2        # addr
jalr    $t9 ; munmap
move    $a1, $s3        # len
lw      $gp, 0x40+var_28($sp)
beqz    $s0, loc_402F94
move    $s0, $zero

                        # CODE XREF: file_write_out+9C↑j
la      $t9, malloc
nop
jalr    $t9 ; malloc
li      $a0, 0x400      # size
lw      $gp, 0x40+var_28($sp)
beqz    $v0, loc_402F90
move    $s4, $v0
la      $t9, puts
lui     $a0, 0x42
jalr    $t9 ; puts
la      $a0, (aHttp11404NotFo+0x18)  # s
lw      $gp, 0x40+var_28($sp)
move    $a0, $s5        # fd
```

```
# int munmap(void *addr, size_t len)
_munmap:
                lw      $t9, off_433B50
                move    $t7, $ra
                jalr    $t9
                li      $t8, 0x4E
# End of function _munmap
```

FIGURE 2. Function hook.

address list of functions GOT_{orig}. In the hook submodule of FIRMCORN, we use the hook_add function provided by Unicorn Engine [20] to add a callback function of type UC_HOOK_CODE to monitor whether the running address of the firmware is in GOT_{mem} or GOT_{orig}. Unicorn Engine is a CPU emulator based on QEMU. On the basis of QEMU, only the CPU emulator part is retained and unnecessary peripheral emulation is removed. Unicorn Engine can support the emulation of multiple architecture codes under one framework as well as support multiple levels of hooks. If the match is successful, FIRMCORN provides an interface for the user to replace the original function with a custom function or to skip the address resolution of __dl_runtime_resolve [21] and jump directly to the actual address of the function in memory. Similarly, FIRMCORN can automatically identify and skip unnecessary functions in the fuzz testing process based on GOT_{mem} and GOT_{orig}, thereby increasing the speed of virtual execution. The implementation principle of the above process is presented in Figure 2.

For statically linked binaries, the compiler compiles the required library files into the program during the compilation of the executable program. This approach can still automatically identify the function and add the hook by parsing the binary symbol table.

D. OPTIMIZED VIRTUAL EXECUTION

In the FIRMCORN Core, we implement the optimized virtual execution technology. In this section, we first introduce a CPU emulator and Unicorn Engine and then introduce the framework to support multiple architectures. Finally, we introduce heuristic algorithms to optimize the virtual execution process.

1) CPU EMULATOR

A CPU emulator is a tool that uses pure software to emulate a CPU, thereby implementing instructions that run different instruction sets on one architecture. The input to a CPU emulator is a binary code or a fragmented binary code. Before using a CPU emulator, the context state needs to be set and the architecture information must be specified. A CPU emulator creates and maintains a virtual stack and memory segment and then decodes binary code into multiple instructions based on the specified instruction set and endianness information. It then reads each instruction for interpretation execution after the formal execution and updates the context state after each instruction execution.

It is not an easy task to implement a full-featured and accurate CPU emulator. Due to the variety of architectures and instructions, the implementation of a CPU emulator is very complicated and cumbersome. Fortunately, QEMU has implemented a pure C CPU emulator, and Unicorn Engine [22] only retains the CPU emulator part of QEMU, removes emulation for other devices, and provides a python interface binding. We use the Unicorn Engine CPU emulator to avoid reinventing the wheel.

The FIRMCORN Core is based on Unicorn Engine APIs. For examples, reg_write is used to write register groups, mem_map is used to allocate stack space, and mem_write is used to write memory to implement context state import, and complete the migration of the entry point state from a physical device to the CPU emulator.

2) MULTI ARCHITECTURE

FIRMCORN is designed to support multiple architectures. The framework will extract program architecture information, including instruction set, word length, and endianness, and automatically configure the CPU emulator.

Function parameters, function calls, and endianness vary in different architectures. For example, in an x86 32bit environment, all parameters of function calls are passed through the stack, and the return value of the function is stored in the eax register. However, in an MIPS environment, the first four parameters are passed through the $a0~$a3 registers, the latter parameters are passed through the stack, and the return value of the function is stored in the $v0~$v1 register. Therefore, in the framework design, we abstract the registers that may be used and provide users with a consistent interface to allow them to write custom emulation functions. FIRMCORN currently supports testing four architectures, shown in Table 2. We implement register abstraction for these four architectures.

TABLE 2. Abstract registers.

Abstract Registers	x86 32	x86 64	ARM 32	MIPS 32
REG_PC	X86_REG_EIP	X86_REG_RIP	ARM_REG_PC	MIPS_REG_PC
REG_SP	X86_REG_ESP	X86_REG_RSP	ARM_REG_SP	MIPS_REG_SP
REG_RA	NULL	NULL	ARM_REG_LR	MIPS_REG_RA
REG_RES	X86_REG_EAX	X86_REG_RAX	ARM_REG_R0	MIPS_REG_V0 MIPS_REG_V1 MIPS_REG_V2
REG_ARGS	NULL	X86_REG_RDI X86_REG_RSI X86_REG_RDX X86_REG_RCX X86_REG_R8 X86_REG_R9	ARM_REG_R0 ARM_REG_R1 ARM_REG_R2 ARM_REG_R3	MIPS_REG_A0 MIPS_REG_A1 MIPS_REG_A2 MIPS_REG_A3

3) HEURISTIC OPTIMIZATION

First, we specifically describe the three types of functions, namely unresolved, unnecessary, and hardware-specific functions:

a: UNRESOLVED FUNCTION

Although the context information is extracted as much as possible, dynamically allocated memory, such as heap space, may not be initialized and thus not obtained at the entry point; therefore, import of this part of memory in advance is not possible. If the library function reads and writes this part of the memory, it will cause errors in the emulation process. We define these functions as unresolved functions in the framework.

b: UNNECESSARY FUNCTION

There are some functions that are not needed in the fuzzing process, such as the `puts` function and similar functions. We define these functions as unnecessary functions. To realize more efficient fuzz testing, our framework provides an interface for users to skip certain functions. When these functions are executed, the program counter (PC) register will be set to the address of the next instruction and the stack will be balanced.

c: HARDWARE-SPECIFIC FUNCTION

IoT device firmware has access to hardware functions, such as reading GPIO pins or NVRAM regions; however, the lack of these hardware pins during emulation will cause the program to halt and crash. These functions depend on the hardware of the device, and therefore defined as hardware-specific functions.

In order to handle the aforementioned functions, we coordinate GOT-based and exception-based hook techniques to carry out heuristic optimization. FIRMCORN adds two types of exception hook (`UC_HOOK_MEM_READ_UNMAPPED` and `UC_HOOK_MEM_WRITE_UNMAPPED`) functions; each time it reaches the memory space function that encounters dynamic allocation of read and write, it enters the hook function for exception handling. The list of unresolved functions is recorded in the hook function by recording the library function that throws an exception at that time. In the formal fuzzing process, FIRMCORN uses a custom library function to replace the function in the unresolved function list based on the hook submodule. In order to add support for dynamic memory allocation-related functions, we implemented a simple heap emulation with reference to uClibc in FIRMCORN, designed chunks as the basic unit of allocation, and implemented support for malloc, free, realloc, and other functions on this basis. To handle unnecessary functions, FIRMCORN uses the skip function of the hook submodule, which can automatically skip common unnecessary functions in the virtual execution process by adopting the GOT-based hook. As for hardware-specific function, FIRMCORN has written common NVRAM read/write functions that automatically replaces hardware-specific functions that cannot be emulated.

E. FUZZ TESTING

In this section, we will introduce the fuzz submodule and crash submodule of FIRMCORN.

1) START FUZZING

In FIRMCORN, the fuzz submodule is implemented, which is imported and called by the FIRMCORN Core to address two key issues: generation and execution of test cases.

In fuzz testing, test cases are generally generated by either a mutation-based or a generation-based approach [23].The mutation-based approach creates more number of test cases by mutating test samples, whereas the generation-based approach generates test cases on the basis of modeling the protocol or file format used by the system under test.

Due to the limited computing resources, IoT devices seldom use complex protocols or perform complex data processing. Our fuzz submodule allows users to simply model protocols or inputs and create test cases based on the generation method. At the same time, by default, the fuzz submodule supports users to create a large number of test cases using the mutation-based method for fuzz testing the target program. Through the generated test cases, we hijack an input function based on the hook module, shield the differences among various data input methods, and provide a unified interface to facilitate the target program to read into the test cases. For example, in the testing of the DLink router, we hijack the `getenv` function. The Common Gateway Interface (CGI) program of the router can get the CGI environment variables controlled by the attacker through `getenv`. To adapt to various firmware programs, FIRMCORN's design is extensible, thus supporting user-defined hijack functions. If at the entry point of the fuzzing, the tested field data have been imported into the memory, we monitor the parameters of the function. If the field or the address of the field is found, then FIRMCORN replaces this parameter with the generated test cases to implement fuzz testing in this case.

2) CRASH CHECK AND LOG

Another key issue in fuzz testing is monitoring and recording the crash of the sequence and providing the user with a test case that can crash the program. To achieve this, we implement the crash submodule in FIRMCORN.

In the process of optimized virtual execution, we consider two types of crash monitoring methods: memory corruption

TABLE 3. Summary of IoT devices used for experiments.

Vendor	Model	Version	Device Type	Program	Program Description
DLink	DIR-816	A2-1.10	Router	goahead	Embedded HTTP Server
	DIR-629	A1-1.03	Router	cgibin	CGI Binary Program
	DIR-629	B1-2.01	Router	cgibin	CGI Binary Program
	DIR-859	A3-1.06	Router	cgibin	CGI Binary Program
	DIR-823G	A1-1.00	Router	goahead	Embedded HTTP Server
TPLink	WR940N	V4	Router	httpd	Embedded HTTP Server
	WR941N	V4	Router	httpd	Embedded HTTP Server
Ezviz	C6C	C6C-3B1WFR	Camera	ezapp	Web Interface
Dahua	HFW5238M	L1	Camera	sonia	Web Interface
	HFW3236M	L1	Camera	sonia	Web Interface

check and exception detection. Because the MMU of some IoT devices does not have a memory destruction check module, there are cases where although overflow has occurred, the program has not crashed; this is referred to as silent memory corruption in the literature [24]. To counter this, after a sensitive function call, we check the stack data to see if an overflow exists. We also monitor abnormal situations in the execution process and identify exceptions for determining whether the executed program has crashed.

After monitoring the crash, FIRMCORN records the test case of the crash and provides it to the user to determine the specific location of the vulnerability and generate `.crash` files.

V. EVALUATION

In this section, we will evaluate the prototype implementation of FIRMCORN. Herein, we verify whether the proposed method solves the difficulties of IoT firmware fuzzing and test the effectiveness of FIRMCORN vulnerability discovery. In short, we aim to answer the following questions:

- *Accuracy.* Is FIRMCORN's optimized virtual execution more accurate than conventional virtual execution?
- *Efficiency.* By what extent is FIRMCORN's optimized virtual execution efficiency higher than conventional virtual execution efficiency?
- *Stability.* Is FIRMCORN's optimized virtual execution stable?
- *Effectiveness.* How effective is FIRMCORN in identifying vulnerabilities in real IoT devices.

Finally, we present an example to demonstrate how to perform a fuzz testing on an IoT device using FIRMCORN.

Testing Environment: Our experiments were conducted on an Intel(R) Core(TM) i7-7700 CPU 3.60 GHz CPU machine with 8G RAM and 512 G hard disk. The operating system was Ubuntu 16.04.6 LTS.

IoT Device Selection: For our experiments, we chose 10 IoT devices that are mainstream products, and the firmware of these devices can be downloaded from their official websites. Table 3 shows a list of the selected devices and their firmware versions as well as the programs that were tested. The photos of the devices are shown in Figure 3.

FIGURE 3. IoT devices used for experiments.

A. ACCURACY

For the 10 devices listed in Table 3, FIRMCORN used `gdbserver` to attach the firmware program and specify the debugging port. Using the `gdb` remote connection debugging port, the firmware of the devices was remotely debugged. Then, FIRMCORN was run at the entry point to start dumping the context information of the location as the initial condition for this part of the experiment.

In the first experiment, we set the context in FIRMCORN's CPU emulator and prepared to perform optimized virtual execution. The complete functions were executed while tracing the assembly-level jump of the functions in their execution sequence. Then, the result obtained was compared with the optimized virtual execution result; it indicates that the execution sequences of optimized virtual execution and the actual device were exactly the same.

In the second experiment, we set the context for traditional virtual execution in the traditional CPU emulator. However, the entire firmware crashes or exits during the subsequent operation. This is because of the existence of hardware interaction in the firmware or due to the NVRAM read/write functions.

In summary, optimized virtual execution used by FIRMCORN can be based on the actual firmware dump context. While it skips considerable hardware interaction occurring in the firmware initialization phase, it can guarantee the accuracy of virtual execution.

TABLE 4. Results of nbench. Unit: iterations/second.

Benchmark	Optimized	Normal	Overhead
NUMERIC SORT	623.11	616.69	1.04%
STRING SORT	86.03	83.285	3.30%
BITFIELD	3.0844e+08	3.0812e+08	0.10%
FP EMULATION	276.3	270.47	2.16%
FOURIER	2189.1	2107.9	3.85%
ASSIGNMENT	22.662	21.711	4.38%
IDEA	4329.7	4241.9	2.07%
HUFFMAN	1911.7	1833.1	4.29%
NEURAL NET	2.91	2.85	2.11%
LU DECOMPOSITION	85.30	84.13	1.39%
I/O OPERATION	3938.90	2449.04	60.83%
GPIO	230.94	Crash	Null

B. EFFICIENCY

To test the efficiency of virtual execution, we used a benchmark program of `nbench` [25] and added custom I/O operation scenarios and GPIO pin reading scenarios. Note that we modified the `nbench` compilation method as a dynamic link in order to make it adaptable to various firmware programs. FIRMCORN was used to obtain the context state of the starting position of the main function of the `nbench` program. Then, optimized virtual execution and conventional virtual execution were performed to run the program in the CPU emulator, and their performance was recorded in terms of execution time. Furthermore, low-level optimization was performed on unresolved functions in conventional virtual execution to ensure the stable execution of the program. The test results are presented in Table 4.

The results indicate that in the benchmark test program of `nbench`, the efficiency of the optimized virtual execution technology is improved compared with that of the conventional virtual execution technology. This is because optimized virtual execution simplifies the calling process of the library function, and can automatically jump to the actual address in memory, saving time in address resolution. In the I/O operation scenario, optimized virtual execution is significantly more efficient than conventional virtual execution, because FIRMCORN's heuristic optimization skips unnecessary functions, thereby improving the running speed. In the GPIO scenario, FIRMCORN writes custom functions to replace hardware-specific functions, thus ensuring continual running of virtual execution. However, this method does not accurately emulate the hardware, but only aims to be as accurate as possible in the virtual execution process.

C. STABILITY

To test the stability of optimized virtual execution, experiments were conducted in the selected device firmware. We imported the IoT firmware contexts into the FIRMCORN framework to perform conventional virtual execution. However, due to the lack of processing of the read and write functions of the dynamic memory area, all the tested firmware crashed. Next, we performed optimized virtual execution; we added automated processing of unresolved functions and hardware-specific functions, all of which could be executed normally in the CPU emulator. We recorded the three types of functions—unresolved, unnecessary, and hardware-specific functions—present in the firmware listed in Table 3, and plotted histograms shown in Figure 4.

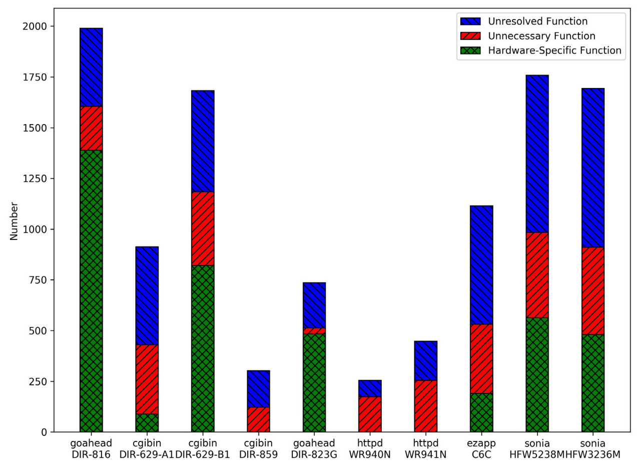


FIGURE 4. Function category.

In optimized virtual execution, the presence of unresolved functions (blue bar) and hardware-specific functions (green bar) majorly contributes to the restriction of the stability of the simulation process, but FIRMCORN can automatically recognize and replace these functions. FIRMCORN incorporates some library functions and common NVRAM operation functions as well as provide interfaces for users to write custom functions. However, a steady increase will result in an increase in virtual execution time; owing to the addition of hook functions in the execution process, optimized virtual execution results in a certain amount of time consumption compared to user-mode emulation.

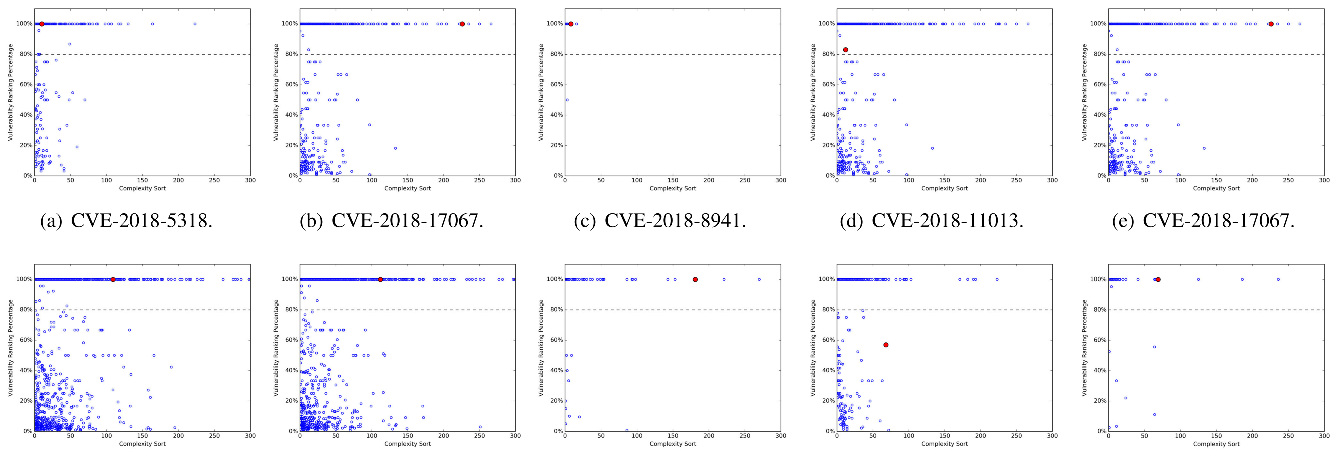
D. EFFECTIVENESS

We evaluated the effectiveness of FIRMCORN in determining real-world IoT device firmware vulnerabilities. This section is divided into three parts:

- Evaluation of the effectiveness of the vulnerable-code search algorithm
- Evaluation of the effectiveness of FIRMCORN in vulnerability discovery
- Evaluation of the accuracy of FIRMCORN's vulnerability reports.

1) EFFECTIVENESS OF THE VULNERABLE-CODE SEARCH ALGORITHM

We selected 10 firmware programs with known vulnerabilities from Firmadyne datasets [26] and CVE [27] list as the base dataset and used FIRMCORN's preanalysis submodule to analyze them. Then, we compared the entry points of the functions of 1-day vulnerabilities with the result. Figure 5 shows the relationship between complexity sorting and the percentage of the vulnerability features of these



(a) CVE-2018-5318. (b) CVE-2018-17067. (c) CVE-2018-8941. (d) CVE-2018-11013. (e) CVE-2018-17067.

(f) CVE-2017-13772. (g) CVE-2019-6989. (h) EDB-ID-44001. (i) EDB-ID-38720. (j) CVE-2017-17033.

FIGURE 5. FIRMCORN preanalysis results.

TABLE 5. Time to crash.

Vulnerability	Vendor	Model	Version	Device	Program	Time to crash
Buffer Overflow	DLink	DIR-629	B1-2.01	Router	cgibin	1h12min
Buffer Overflow	DLink	DIR-629	A1-1.03	Router	cgibin	3h29min
Buffer Overflow	DLink	DIR-859	A3-1.06	Router	cgibin	36min
Buffer Overflow	DLink	DIR-816	A2-1.10	Router	goahead	241s
Buffer Overflow	DLink	DIR-823G	A1-1.00	Router	goahead	1h2min
Buffer Overflow	TPLink	WR940N	V4	Router	httpd	42min
Buffer Overflow	TPLink	WR941N	V4	Router	httpd	54min
NULL	Ezviz	C6C	C6C-3B1WFR	Camera	ezapp	>24h
Unknown Crash	Dahua	HFW5238M	L1	Camera	sonia	8h51min
NULL	Dahua	HFW3236M	L1	Camera	sonia	>24h

firmware programs. In the figure, the red dots denote the entry points of the functions where the 1-day vulnerability was located.

Figure 5 indicates that all firmware has a similar function distribution: many functions with low-complexity and low-vulnerability features exist in the firmware, and irrespective of whether the complexity of 1-day vulnerability functions is high or low, they generally have more high-vulnerability features. Among the 10 test sets, seven 1-day vulnerability functions are ranked first in the complexity grouping, and nine 1-day vulnerability functions are ranked in the top 20% of the complexity grouping. In particular, in the analysis results of the DLink DIR-629 router, two 1-day vulnerabilities (CVE-2018-5318 and CVE-2018-10996) are ranked first in vulnerability ranking. However, note that the result of the preanalysis submodule shows only the sorting of potential vulnerability functions, but the functions with higher ranking are not necessarily vulnerable. For example, the vulnerability-feature percentage of EDB-ID-38720 is 57%, because other functions of the group have performed multiple memory operations and have been confirmed to have no exploitable vulnerabilities, thus resulting in a significant vulnerability feature for EDB-ID-38720.

In the experiment, FIRMCORN was tested according to the default configuration in order to ensure reasonable effectiveness evaluation. This resulted in the firmware effect for some sensitive functions with customization not being particularly good. For example, in the CVE-2017-17033 test sample, the vulnerability is a stack overflow of the QNAP NAS device. Although in the test results, the ranking of the feature of the 1-day vulnerability is still the best, the vulnerability feature of this function is not high. In practical use, the preanalysis module needs to be simply configured to fit different types of firmware. FIRMCORN supports experienced analysts to add the `CGI_Find_Parameter` function as a sensitive function, through which the `QTS` system could obtain user input. Therefore, there may be exploitable vulnerabilities in the process.

2) EFFECTIVENESS OF VULNERABILITY DISCOVERY

In this experiment, we focused on evaluating FIRMCORN's effectiveness of vulnerability discovery, especially in firmware vulnerability discovery of real-world IoT devices. We used FIRMCORN to initiate service requests to devices listed in Table 3. After obtaining the context of the entry point location, fuzz testing was performed in FIRMCORN. We recorded the time when the first valid crash occurred and plot the values in Table 5. To obtain statistically significant results, the above procedure was repeated 10 times and the average of 10 data was obtained.

As Table 5 shows, FIRMCORN completed execution within 16 h and 50 min and determined eight memory corruptions in 10 IoT devices. Upon checking, seven of these corruptions were due to buffer overflow (stack-based).

FIRMCORN has a significant efficiency and utility improvement over the current state-of-the-art IoT device fuzzing tools. Unlike IoTFuzzer, FIRMCORN does not need to restart the device repeatedly in the testing process, thus saving considerable time for device restart. Moreover, the IoTFuzzer framework can only test App-based devices, so it has major device limitation. By contrast, the DLink DIR-629 router tested by FIRMCORN, for example, is not controlled by an App. Compared with FIRM-AFL, FIRMCORN can find some special vulnerabilities; for example, after 42 minutes of testing of TPLink WR940N, FIRMCORN could locate a group with a complexity of 109. The first ranking of the vulnerability index in this group was for the `ipAddrDispose` function, and after testing this function to determine the stack overflow vulnerability caused by incorrect use of `strcpy` calls, the vulnerability of CVE-2017-13772 was confirmed. However, testing the firmware using FIRM-AFL did not report a valid crash for more than 24 h.

We analyzed the above `.crash` files and confirmed that most of them were caused by the same known vulnerabilities; however, we also discovered two new vulnerabilities during our testing, which we will discuss below.

0-day Vulnerabilities: During the testing of the DLink DIR-859 router using FIRMCORN, we found two unknown vulnerabilities after 36 min and 1 h 18 min. We reported these to the equipment manufacturer and MITRE corporation. The details of these two vulnerabilities are as follows:

- CVE-2019-16341: Buffer overflow in D-link dir-859 (hardware version: A3 firmware version: 1.06). Attackers can use the `HTTP_SOAPACTION` field in the cgibin to cause device crash or remote command execution.
- CVE-2019-16342: Buffer overflow in D-link dir-859 (hardware version: A3 firmware version: 1.06). Attackers can use the `REMOTE_ADDR` field in the cgibin to cause device crash or remote command execution.

3) ACCURACY OF VULNERABILITY REPORTS

To test the accuracy of vulnerability report generated by FIRMCORN, we applied FIRMCORN to the devices listed in Table 3, ran it for 24 h, and analyzed `.crash` files. Figure 6 presents the plot to show the number of crashes reported by FIRMCORN and the number of crashes that is considered valid.

Figure 6 shows that a large number of configuration read functions are present in the IoT devices. For example, in the `goahead` program of DLink DIR-823G, `apmib_get` is called to read the username and password. These configuration files do not exist in the CPU emulator, causing the fuzz testing to crash; this is the main cause of false positives in the experiment. To eliminate these false positives, we added a simple emulation of the above functions so that these functions can return values normally. FIRMCORN automatically identifies these functions during fuzz testing and replaces them with custom emulation functions to ensure that fuzz testing will not produce false positives.

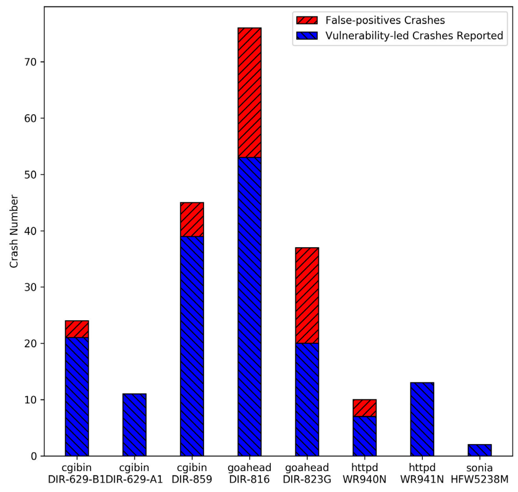


FIGURE 6. False positives.

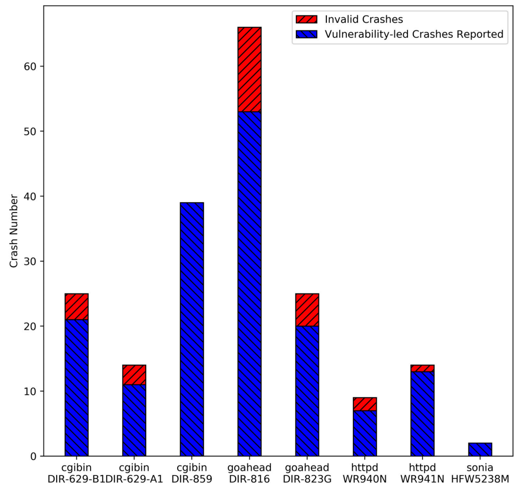


FIGURE 7. Fuzzing accuracy.

Further, we used vulnerability-led crash inputs as input for testing the actual device, and then monitored the operation of the device program to determine whether a crash occurred. Figure 7 shows the relationship between invalid crashes in actual equipment and the number of crashes reported by FIRMCORN.

Figure 7 indicates that the efficiencies reported by FIRMCORN cannot always be achieved on actual devices. In particular, vulnerability existed in the part of the branch that required authentication, which could not be bypassed in the actual device; therefore, vulnerability-led crashes reported by FIRMCORN could not cause the actual device crash.

E. CASE STUDIES

In this section, we present a test case showing the application of FIRMCORN to perform fuzz testing on real-world devices. The device is DLink DIR-859, with the hardware version A3 and firmware version 1.06.

We obtained the shell of the device through the UART ports by TTL serial to USB converter (FT232 chip), as shown in Figure 7.

We used FIRMCORN's preanalysis submodule to analyze the `cgibin` program in the firmware, which was a firmware CGI binary. We sorted the entry points of vulnerable codes and performed dumping of context information of these entry points in the device. After importing the context, we added a `Fuzzer` object using the `add_fuzz` function provided by FIRMCORN, and finally started the fuzz test using the `start_run` function. After FIRMCORN ran for 34 min, we tested the entry point address `0x40F7DC` with a complexity value equal to 8. The entire entry point information of the complexity group is as follows.

```
(8, [{31: [30, 1, 0x40b5c8]}, {23: [20, 3, 0x40f7dc]},
{12: [10, 2, 0x40a428]}, {4: [0, 4, 0x41f654]},
{4: [0, 4, 0x41153c]},{4: [0, 4, 0x408108]},
{4: [0, 4, 0x405428]}, {3: [0, 3, 0x41b314]},
{3: [0, 3, 0x4199e0]}])
```

Further, we continued to use the default seed for the fuzzing, and FIRMCORN prompted occurrence of crashes after approximately 2 min. We checked the `.crash` files under the outputs folder and checked the binary code to confirm that the crash was caused by stack overflow due to incorrect calls to `sprintf`. The binary code is represented by MIPS, as follows.

```
0x040F898:  lui     $s0, 0x43
0x040F89C:  lw      $a3, dword_435450
0x040F8A0:  la      $t9, xmldbc_set
0x040F8A4:  move    $a0, $zero
0x040F8A8:  move    $a1, $zero
0x040F8AC:  jalr    $t9 ; xmldbc_set
0x040F8B0:  move    $a2, $s3
0x040F8B4:  lw      $gp, 0x130+var_120($sp)
0x040F8B8:  lw      $a3, dword_435450
0x040F8BC:  lui     $a1, 0x42
0x040F8C0:  la      $t9, sprintf
0x040F8C4:  la      $a1, aSS          # "%s/%s"
0x040F8C8:  addiu   $a2, $s1,(unk_434010-0x430000)
0x040F8CC:  jalr    $t9 ; sprintf
0x040F8D0:  move    $a0, $s3          # s
0x040F8D4:  lw      $a1, 0x20($s2)
0x040F8D8:  lw      $gp, 0x130+var_120($sp)
0x040F8DC:  beqz    $a1, loc_40F8F4
```

In the above assembly code, we can see that the `sprintf` function is called at the `0x040F8CC` address. The destination address (register `$a0`) is the stack space, the format string (register `$a1`) is `%s/%s`, the first source address (register `$a2`) is the identified string, and the second source address (register `$a3`) is the content of the `HTTP_SOAPACTION` parameter obtained through `getenv`, which is controllable and causes the stack overflow.

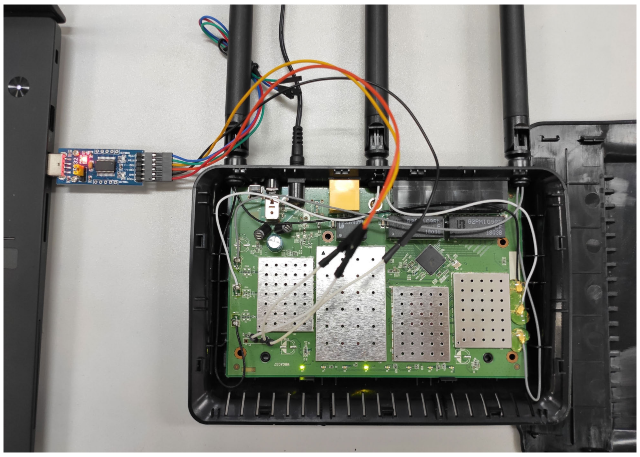


FIGURE 8. DLink DIR-859. Connected to a computer by a USB converter.

VI. DISCUSSION

Although our framework can effectively discover vulnerabilities in IoT devices, it has further scope for improvement. In this section, we discuss the limitations of the framework and provide some insights into future work.

A. LIMITATION ON CPU ARCHITECTURE

FIRMCORN currently supports the following CPU architectures: armel, armeb, mipsel, mipseb, and x86(LE). These architectures already account for 92.7% of the FIRMADYNE datasets and 81.5% of the reference [14] datasets. In the future work, it is relatively easy to provide more architecture support, because Unicorn Engine provides direct support for related architecture APIs. FIRMCORN also pays attention to the decoupling of different architectures during the implementation process, which is convenient for adding new CPU architectures.

B. LIMITATIONS ON OPERATING SYSTEM

Currently, FIRMCORN can only support the testing of firmware on Linux systems. This limitation comes from the way FIRMCORN dumps context information. In the future, we may explore additional dump methods for general context information to support more operating systems.

VII. RELATED WORK

In this section, we briefly describe the work of other researchers.

A. IoT FIRMWARE ANALYSIS

Costin *et al.* [14] performed the first public large-scale analysis of firmware images, unpacking 32000 firmware images and yielding 1.7 million individual files. The authors implemented an automated framework to collect and analyze large-scale firmware images, and detected 38 CVE vulnerabilities in more than 693 firmware images without performing complex static analysis. Shoshitaishvili *et al.* [28] proposed the Firmalice framework for embedded device firmware analysis based on symbolic execution engines, and proposed a new authentication bypass vulnerability model that can detect complex backdoor vulnerabilities without relying on the details of the firmware implementation itself.

Xu *et al.* [29] proposed a neural network-based approach to detect cross-platform binary code similarity, implemented a prototype system called Gemini, and evaluated it on OpenSSL. In addition, Wang *et al.* [30] proposed a novel staged firmware vulnerability detection method that provides a higher degree of similarity analysis accuracy in a two-stage combination. Feng *et al.* [31] proposed a bug search scheme for IoT systems, implemented the bug search engine called Genius, and conducted extensive evaluation on real devices.

Static analysis of IoT firmware is simple and efficient, but there are still problems such as inaccurate analysis results and high false positives. Zaddach *et al.* [16] proposed the Avatar framework to forward I/O accesses from the emulator to the embedded device. On the basis of this work, Muench *et al.* [17] implemented Avatar2 to enable interoperability between different dynamic binary analysis frameworks, debuggers, emulators, and real physical devices. However, the above scheme has considerable overhead in hardware and emulator switching, thereby limiting its application in fuzz testing.

Chen *et al.* [5] proposed a dynamic analysis framework FIRMADYNE based on full-system emulation, and based on this, the Metasploit framework was developed to discover known vulnerabilities. Costin *et al.* [32] conducted large-scale dynamic analysis of firmware and found vulnerabilities related to a Web interface; however, the scheme could not identify vulnerabilities in other modules of the firmware. Zandberg *et al.* [33] proposed the possibility of creating a secure, standards-compliant firmware update solution that uses security technology to secure IoT devices.

B. FUZZING

Li *et al.* [12] implemented V-Fuzz, analyzed the existing limitations of coverage-based fuzzing, proposed a vulnerability-oriented evolutionary fuzzing prototype, and used the probability of the existence of vulnerability of the deep learning model to guide fuzz testing. However, this solution requires dataset support and is currently only based on the NIST dataset training model, which is not applicable to IoT device firmware fuzzing. Du *et al.* [34] implemented the lightweight and extensible framework LEOPARD, which uses program metrics to identify potential vulnerability functions, but the framework is not suitable for firmware without source codes.

In addition, Maier *et al.* [22] explored the use of CPU emulation to fuzz arbitrary parsers in kernel space with coverage-based feedback and proposed a fuzzing framework based on Unicorn Engine, which can perform fuzzing tests on kernel modules and drivers. However, this approach has several problems in firmware fuzzing because it does not support the MIPS architecture and is not effective for hardware-dependent function processing. Yan *et al.* [35] proposed a fuzzing-based framework of quantifying software exploitability, called ExploitMeter. ExploitMeter integrates machine learning-based prediction and dynamic fuzzing tests in a Bayesian manner. The author evaluated the performance of ExploitMeter in a dynamic environment based on 100 Linux programs.

C. IoT FIRMWARE FUZZING

In 2018, Muench *et al.* [24] raised the point regarding the difficulty of fuzz testing on IoT firmware and proposed six heuristics for detecting faults due to memory corruption. However, this solution incurs immense overhead during the device restart and hardware switching phase, so it is not suitable for fuzz testing on real-world devices. Chen *et al.* [6] implemented IoTFuzzer, which can conduct fuzz testing on devices without firmware and automatically detect vulnerabilities in IoT devices. However, this scheme can only test App-based IoT devices, thus posing limitations of devices. In 2019, Zheng *et al.* [7] developed FIRM-AFL, which can combine the advantages of system-mode emulation and user-mode emulation via augmented process emulation, and tested IoT firmware based on greybox fuzzing. However, due to the hardware dependency of firmware and the low proportion of vulnerability codes, the scheme only relying on greybox fuzzing has the problem of low efficiency of vulnerability mining.

VIII. CONCLUSION

In this paper, we presented FIRMCORN, a firmware fuzz testing framework based on optimized virtual execution, and it was used to apply vulnerability-oriented fuzzing to IoT firmware for the first time. To achieve faster, more accurate, and more stable fuzz testing, we developed a series of novel technologies according to the characteristics of IoT device firmware, namely, the vulnerable-code search algorithm to determine the entry point for vulnerability-oriented fuzzing and optimized virtual execution technology to improve execution speed, accelerate execution accuracy, and ensure execution stability.

We extensively evaluated the efficiency and effectiveness of FIRMCORN and demonstrated the application of FIRMCORN to fuzz testing on real-world IoT devices. The results showed that FIRMCORN can significantly improve the accuracy, efficiency, and stability over conventional virtual execution and can detect 0-day vulnerabilities.

REFERENCES

[1] Gartner. *Leading the IoT—Gartner Insights on How to Lead in a Connected World*. Accessed: May 12, 2014. [Online]. Available: https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf

[2] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[3] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur. (ASIA CCS)*, 2015, pp. 555–566.

[4] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proc. 22th USENIX Secur. Symp. (USENIX Sec)*, Washington, DC, USA, Aug. 2013, 2013.

[5] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.

[6] 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," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[7] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proc. 8th USENIX Secur. Symp. (USENIX Sec)*, 2019, pp. 1099–1114.

[8] M. Zalewski. *American Fuzzy Lop*. Accessed: Jun. 1, 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[9] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, Jan. 2012.

[10] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, vol. 13, 2013, pp. 511–522.

[11] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IIEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[12] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing," 2019, *arXiv:1901.01142*. [Online]. Available: http://arxiv.org/abs/1901.01142

[13] M. Sharma, N. Agarwal, and S. R. N. Reddy, "Design and development of daughter board for USB-UART communication between Raspberry Pi and PC," in *Proc. Int. Conf. Comput., Commun. Autom.*, May 2015, pp. 944–948.

[14] A. J. A. Costin Zaddach Francillon and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. USENIX Secur. Symp.* Aug. 2014, pp. 95–110.

[15] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX ATC*, 2005, pp. 41–46.

[16] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.

[17] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.*, 2018, pp. 1–11.

[18] C. Ebert and J. Cain, "Cyclomatic complexity," *IEEE Softw.*, vol. 33, no. 6, pp. 27–29, Nov. 2016.

[19] C. Zhang, L. Duan, T. Wei, and W. Zou, "SecGOT: Secure global offset tables in ELF executables," in *Proc. 2nd Int. Conf. Comput. Sci. Electron. Eng. (ICCSEE)*, 2013, pp. 1–4.

[20] N. A. Quynh and H.-V. Dang, "Unicorn: Next generation cpu emulator framework," in *Proc. BlackHat*, 2015, pp. 1–39.

[21] A. D. Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "How the ELF ruined Christmas," in *Proc. 24th USENIX Secur. Sym.*, 2015, pp. 643–658.

[22] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *Proc. 13th USENIX Workshop Offensive Technol.*, 2019, pp. 1–11.

[23] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.

[24] 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 *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[25] *Nbench*. Accessed: Jan. 2, 2017. [Online]. Available: https://www.math.utah.edu/~mayer/linux/bmark.html

[26] *FIRMADYNE Dataset*. Accessed: Feb. 21, 2016. [Online]. Available: https://github.com/firmadyne/firmadyne/tree/master/database

[27] *CVE List Home*. Accessed: Feb. 5, 2019. [Online]. Available: https://cve.mitre.org/cve/

[28] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice–automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[29] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2017, pp. 363–376.

[30] Y. Wang, J. Shen, J. Lin, and R. Lou, "Staged method of code similarity analysis for firmware vulnerability detection," *IEEE Access*, vol. 7, pp. 14171–14185, 2019.

[31] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2016, pp. 480–491.

[32] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded Web interfaces," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS)*, vol. 16, 2016, pp. 437–448.

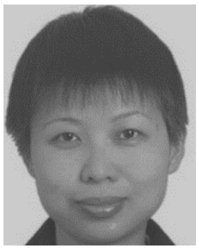
[33] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained IoT devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71907–71920, 2019.

[34] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 60–71.

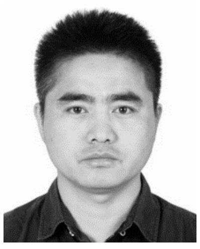
[35] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "ExploitMeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *Proc. IEEE Symp. Privacy-Aware Comput. (PAC)*, Aug. 2017, pp. 164–175.

ZHIJIE GUI was born in 1996. He received the B.S. degree in network engineering from Information Engineering University, Zhengzhou, in 2018. He is currently pursuing the M.S. degree in cyberspace security with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include the IoT security and vulnerability detection and exploitation.

HUI SHU was born in 1974. He received the Ph.D. degree in computer science and technology from Information Engineering University, in 2001. He is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include reverse analysis and the IoT security.



FEI KANG was born in 1972. She is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. Her research interests include network security mechanism analysis and malware analysis.



XIAOBING XIONG was born in 1985. He received the Ph.D. degree in computer science and technology from Information Engineering University, in 2013. He is currently an Assistant Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, China. His research interests include reverse analysis and malware detection.