Detecting Vulnerability on IoT Device Firmware: A Survey

Xiaotao Feng, Xiaogang Zhu, *Member, IEEE*, Qing-Long Han, *Fellow, IEEE*, Wei Zhou, Sheng Wen, *Senior Member, IEEE*, and Yang Xiang, *Fellow, IEEE*

***Abstract*—Internet of things (IoT) devices make up 30% of all network-connected endpoints, introducing vulnerabilities and novel attacks that make many companies as primary targets for cybercriminals. To address this increasing threat surface, every organization deploying IoT devices needs to consider security risks to ensure those devices are secure and trusted. Among all the solutions for security risks, firmware security analysis is essential to fix software bugs, patch vulnerabilities, or add new security features to protect users of those vulnerable devices. However, firmware security analysis has never been an easy job due to the diversity of the execution environment and the close source of firmware. These two distinct features complicate the operations to unpack firmware samples for detailed analysis. They also make it difficult to create visual environments to emulate the running of device firmware. Although researchers have developed many novel methods to overcome various challenges in the past decade, critical barriers impede firmware security analysis in practice. Therefore, this survey is motivated to systematically review and analyze the research challenges and their solutions, considering both breadth and depth. Specifically, based on the analysis perspectives, various methods that perform security analysis on IoT devices are introduced and classified into four categories. The challenges in each category are discussed in detail, and potential solutions are proposed subsequently. We then discuss the flaws of these solutions and provide future directions for this research field. This survey can be utilized by a broad range of readers, including software developers, cyber security researchers, and software security engineers, to better understand firmware security analysis.**

***Index Terms*—Firmware emulation, internet of things (IoT) firmware, network fuzzing, security, static analysis.**

I. Introduction

VARIOUS internet of things (IoT) devices have come into our sight, and their presence has brought us new life patterns and convenience [1]–[4]. Recent statistics suggested that the number of IoT devices will reach 75.44 billions by 2025 [5]. However, IoT devices have long been reported to be vulnerable to various attacks [6]–[11]. For example, Mirai [12], which was first exposed in 2016, used a huge number of compromised IoT devices to launch distributed denial of service (DDoS) attacks on valuable/sensitive targets including the website of US White House. According to Paloalto's report released in 2020 [13], over 50% of the IoT devices in the world are vulnerable to attacks whose severity were ranked medium to high.

Security analysis is important for both software and hardware [14]–[18]. To ensure that the devices are secure and trusted, people usually investigate the firmware of IoT devices and detect potential vulnerabilities in the firmware. Fig. 1 presents the components of IoT devices and the current ways to analyze firmware security. An IoT device is a small embedded system that has firmware running on the system. It has various peripheral devices, applications running in the firmware, and a network communication processor that can communicate with the Internet. Complex IoT devices are equipped with the underlying operating system kernels and drivers that help interact with peripheral devices. To detect security flaws, most security analyses of IoT devices require the access to firmware. With the firmware, analyzers can perform manual analysis, binary lifting or emulation to detect flaws in firmware. Since IoT devices can communicate with the outside world through network, another way of vulnerability detection is to test IoT devices based on network.

Although many solutions have been proposed to detect vulnerabilities in IoT firmware (e.g., code analysis and fuzzing with emulation [19]–[23]), there are still several challenges that require further research. Compared with software security analysis, inconsistent development standards and a closed market environment make it more difficult to develop IoT firmware vulnerability detection. Specifically, the two major challenges for security analysis of firmware are the complex execution environment and the close source of firmware. First, IoT firmware runs independently in specially designed embedded systems. Many security researchers try to use different methods (i.e., side channel or network-level monitoring) to obtain more information for firmware analysis when the IoT device is running. However, these methods are often restrictive and can only obtain little information due to the complex execution environment. Second, the acquisition of firmware is often impractical due to the security concerns from device manufacturers. Many manufacturers will not open-source their firmware and will disable the Debug mode or joint test action group (JTAG). Therefore, many methods based on firmware analysis become impractical in the wild. In order to overcome these challenges, researchers have designed

Manuscript received May 28, 2022; revised July 1, 2022; accepted July 14, 2022. Recommended by Associate Editor Tao Yang. *(Corresponding author: Qing-Long Han.)*

Citation: X. T. Feng, X. G. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on IoT device firmware: A survey," *IEEE/CAA J. Autom. Sinica*, vol. 10, no. 1, pp. 25–41, Jan. 2023.

The authors are with the School of Science, Computing and Engineering Technologies, Swinburne University of Technology, Hawthorn VIC 3122, Australia (e-mail: xfeng@swin.edu.au; xiaogangzhu@swin.edu.au; qhan@swin.edu.au; weizhou@swin.edu.au; swen@swin.edu.au; yxiang@swin.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JAS.2022.105860

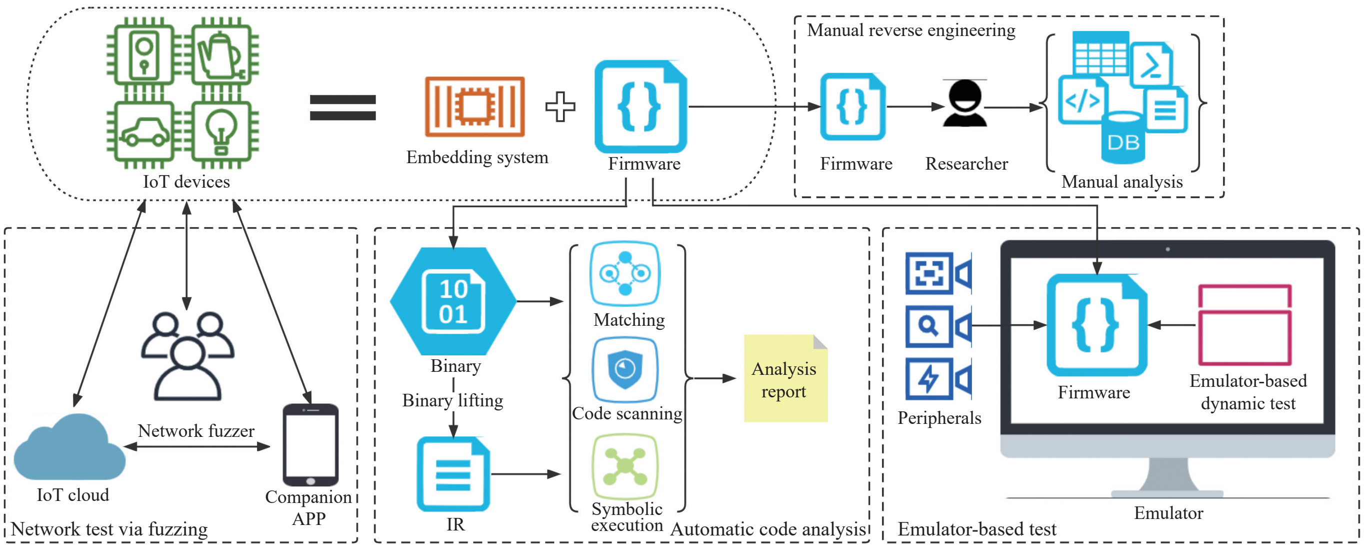


Fig. 1. Firmware security and its solutions. The firmware security can be analyzed based on binary code, firmware image, IoT network and manual analysis.

various practical solutions to test firmware from different analysis aspects. For example, emulator-based testing is a type of method specifically designed to analyze IoT firmware. In order to run firmware, a virtual environment is provided by the emulator. The execution information then can be obtained during runtime, and the virtual environment enables dynamic analysis on firmware. In addition, some automatic code analysis methods are designed for analyzing firmware without running the firmware. Moreover, there are also many studies that perform manual reverse engineering analysis based on specific devices and scenarios. However, these analysis methods require the acquisition of firmware. Considering the interactive ability of IoT devices with the network, some other researchers have proposed methods for testing IoT devices through the network without firmware.

As the security of IoT device has been gradually taken seriously in recent years, there are already some research surveys published on it. For example, some studies have concluded that vulnerabilities in IoT device firmware are more complicated [24]–[27] and more difficult to detect [28], [29] compared with detecting vulnerabilities in traditional software. Another work focuses on the categorization of bugs in IoT devices [25]. To present the challenges of emulation and rehosting in firmware, the problem encounted in dynamic analysis is analyzed [30]. In addition, the taxonomy of approaches in binary analysis is analyzed, including symbolic execution, dynamic analysis, and static analysis [31].

To help researchers and developers better understand firmware security, we conduct a more comprehensive study on vulnerability detection techniques of IoT devices. In this survey, we present the analysis solutions in three levels. At the top level, we classify detection solutions based on the analysis perspectives, and obtain four classes that are emulator-based test, automatic code analysis, network test via fuzzing, and manual reverse engineering. At the second level, detection solutions are categorized based on different challenges. The third level includes detailed solutions for their corresponding challenges. This survey also discusses the pros and cons of the solutions as well as future directions in this area. Different from existing surveys that are concerned about the technical classification [31], [32], our survey focuses on the challenges in vulnerability detection of IoT firmware. Since we systematically analyze the solutions to those challenges, our survey offers deep insights of vulnerability detection in IoT devices.

The main contributions of this survey are summarized as follows:

1) We describe three types of embedded systems that help the analysis of emulator-based solutions. Moreover, we collect four techniques, including fuzzing, symbolic execution, fault injection, and binary lifting, which are commonly used in existing vulnerability detection of IoT firmware.

2) We propose a taxonomy, which classifies vulnerability detection of IoT firmware in three levels, including test perspectives, challenges, and solutions. Additionally, we systematically analyze the challenges and solutions, considering both breadth and depth.

3) We further discuss the limitations for existing solutions. Meanwhile, we provide future directions for readers to follow.

The rest of this survey is organized as follows. In Section II, we first discuss the classification methods of IoT devices, and introduce the taxonomy of various IoT firmware security analysis methods and some preliminary techniques. We then respectively discuss the emulator-based test, automatic code analysis, network test and manual reverse engineering methods in Sections III–VI. After that, we analyze the challenges faced by these method categories and possible solutions in Section VII and discuss the possible solutions in future direction in Section VIII, followed by Section IX for the conclusion of this survey.

II. Preliminary

In this section, we first introduce the classification criteria for IoT devices. When discussing device emulation (Section III-B), different classes of devices differ in their emulation solutions. We then introduce the techniques that are commonly used in various vulnerability detection methods. After that, we present the taxonomy of research perspectives and the challenges they faced, as well as the collection of cutting-edge IoT device firmware vulnerability detection methods.

A. Classification of IoT and Embedded Devices

Based on functionality, embedded devices can be divided into high-level categories, such as printers, smart meters, and IP cameras. Additionally, the embedded systems of these devices could be categorized by different criteria, such as the field of usage, the computing power, their unit cost and so on [28]. Moreover, the firmware of different kinds of embedded systems also falls in a large range, from a few lines of code program to customized versions of desktop OSes (e.g., Linux). We follow the classification of embedded systems from two works, which classify embedded systems based on the type of firmware [28], [30]:

1) General purpose embedded system (GPES) is Type 1 embedded system, which uses general purpose operating systems (e.g., real-time Linux and embedded Windows).

2) Special purpose embedded system (SPES) is Type 2 embedded system. The operating systems (e.g., ZephyrOS and VxWorks) IoT devices use are specifically developed for the embedded system.

3) Bare-metal embedded system (BMES) is Type 3 embedded system and they usually work without a true OS abstraction. Some of them have a light-weight OS-Library.

The categorization shown above is not accurate. For example, the line between GPES and SPES is blurred [30]. However, this classification based on the type of firmware is helpful to determine and understand what emulation method should be taken to emulate a target firmware (Section III). Although the operating systems of GPES are often cut to fit the embedded system, emulating these types of systems are still well supported by desktop software [22] (i.e., QEMU and Panda). Emulating the operating systems in SPES is more challenging and often requires emulating both the kernel and user space [30], because they are not derived from a desktop operating system. BMES's application can directly access hardware. Recently, there are some works [33], [34] researching how to emulate on these systems.

B. Preliminary Techniques

Before discussing how to analyze the firmware of IoT devices, we first introduce some standard technologies in security analysis. These technologies are more like basic ideas or tools than complete solutions to help researchers solve problems in specific situations. In other words, these methods provide a framework for how to detect vulnerabilities, but how to implement it on the firmware requires careful adjustment by researchers.

1) Fuzzing: Fuzzing is one of the most successful software testing techniques, and it has been used primarily for finding security-related bugs [35]–[39]. The core idea of fuzzing is to automatically or semi-automatically generate random data, feed it into a program, and monitor program exceptions, such as crashes or assertion failures, to find possible vulnerabilities, i.e., memory leaks. Usually, the user must provide one or more initial inputs as the original seed(s) before fuzzing campaigns. A fuzzer (an implementation of a fuzzing algorithm) will use these seeds as materials, generate new test cases through mutation strategies, and then feed them to the program. By monitoring the program, fuzzing can detect whether the program is abnormal (e.g., crash) after using test cases as inputs. The main steps of fuzzing are continuously generating test cases, feeding the program with test cases, and detecting whether the program is abnormal. These three steps will circulate continuously until the user stops them.

The composition of fuzzing also changes when faced with different environments and test objectives. When testing firmware, unlike the direct running and testing of binary executable programs, the fuzzing algorithm requires a virtual environment in which the program can run. In the test of communication over the network, the fuzzing algorithm cannot feed input to the target program through the standard input (stdin). It takes the form of network communication messages to send test cases to the target.

In recent years, coverage-guided fuzzing has been proven to be successful and applied widely in finding vulnerabilities in various applications [40]–[42]. The typical example of coverage-guided fuzz testing, American fuzzy lop (AFL) [43], first inserts instrumentation into the program to get the code coverage of each test. These coverage-guided fuzzers will reward inputs (e.g., retaining them as seeds) if they discover a new coverage. Such a mechanism will make fuzzing continuously explore new and unknown program states. Experimental results show that more program coverage discovered by fuzzing can help fuzzer find more vulnerabilities [44].

2) Symbolic Execution: Symbolic execution is a program analysis technique, which can obtain the input that allows specific code areas to be executed by analyzing the program. By symbolizing the variables, the symbolic execution maintains a set of constraints for each execution path. When the target code is reached, the analyzer can obtain the corresponding path constraint and then use the constraint solver to obtain the specific value that can trigger the target code. This technique does not use fully specified input values but abstractly represents variables as symbols and uses a constraint solver to construct actual instances that may cause attribute conflicts.

Many security practitioners have brought the concept of symbolic execution to vulnerability detection. KLEE [45], a method for automatically generating symbolic execution tests, was proposed in 2008. S2E [46] is another popular open-source symbolic execution platform proposed in 2011. Since it is based on QEMU [47], it enables symbolic execution on full system. It can also support testing on both user-space applications and drivers. In their practical tests, the test coverage of BUSYBOX and other embedded system management suite software exceeded 90% on average. Moreover, there is an active community that constantly writes and maintains many useful S2E plugins for the improvement of performance (e.g., better state pruning algorithms) or new program analysis tool development [48]. Many existing security analysis methods that use symbolic execution are based on KLEE and S2E.

3) Software Fault Injection: In software testing, the technique of fault injection is to inject errors to cover those situations that are not likely to occur under normal circumstances. As a consequence, it will increase the coverage of the test

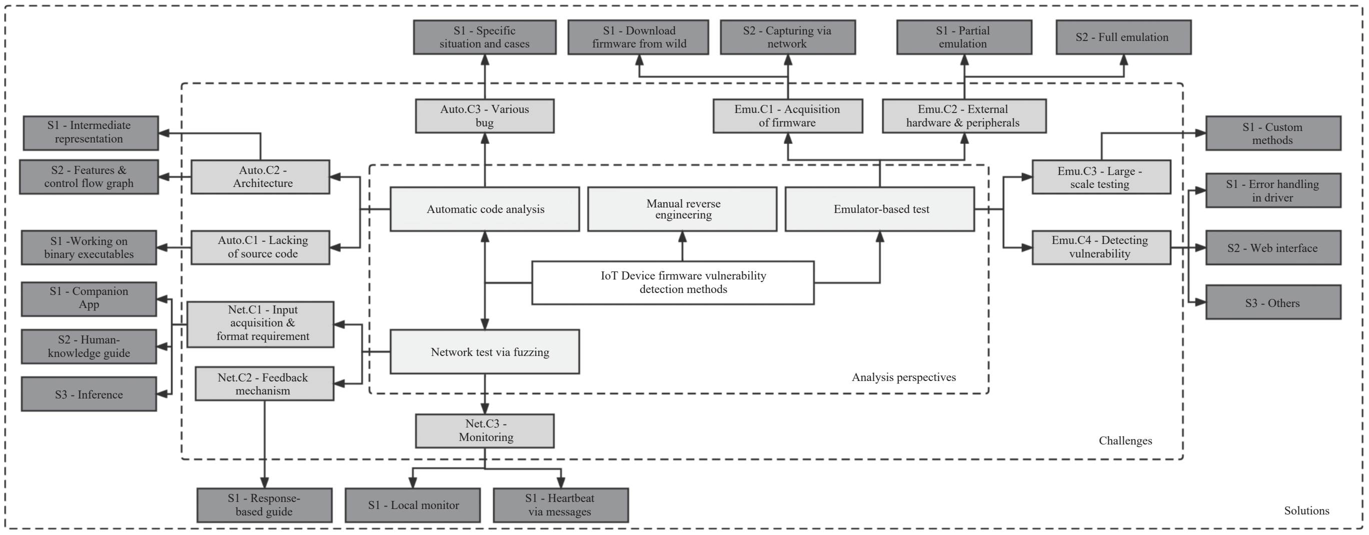

S1 - Specific situation and cases
S1 - Download firmware from wild
S2 - Capturing via network
S1 - Partial emulation
S2 - Full emulation
Auto.C3 - Various bug
Emu.C1 - Acquisition of firmware
Emu.C2 - External hardware & peripherals
S1 - Intermediate representation
S2 - Features & control flow graph
Auto.C2 - Architecture
S1 -Working on binary executables
Auto.C1 - Lacking of source code
Automatic code analysis
Manual reverse engineering
Emulator-based test
Emu.C3 - Large - scale testing
Emu.C4 - Detecting vulnerability
S1 - Custom methods
S1 - Error handling in driver
S2 - Web interface
S3 - Others
IoT Device firmware vulnerability detection methods
S1 - Companion App
S2 - Human-knowledge guide
S3 - Inference
Net.C1 - Input acquisition & format requirement
Net.C2 - Feedback mechanism
Network test via fuzzing
Analysis perspectives
Net.C3 - Monitoring
Challenges
S1 - Response-based guide
S1 - Local monitor
S1 - Heartbeat via messages
Solutions

Fig. 2. Firmware security and its solutions. The firmware security can be analyzed based on code, firmware image, IoT network and manual work.

[49], [50]. When testing a program and verifying the integrity of its functions, the program's robustness under certain extreme conditions must also be considered in some scenarios. The errors can be injected into related software environments such as environmental variables, registers, memory, file systems, registries, and system calls when the software is running. One of the most commonly used implementation methods is hooking a system call. After the hook is reached, errors are injected by modifying the call parameters and return values.

4) Disassembly & Binary Lifting: Disassembly converts target code or machine language into assembly language code. Software security analysis methods often require the target assembly code or source code. However, in the wild, it is often challenging to get information from the program under test (PUT). Although the disassembled program will be slightly different from the original one, it is important to have the capability of getting the readable assembly code [51]. In the existing market, tools such as IDA Pro [52] and OllyDBG [53] can complete the disassembly work well.

In addition, many static analysis methods need to convert the binary machine code into another unified general higher-level language. Intermediate language, as an equivalent internal representation code of a source program, featured by easy translation into the target program that has nothing to do with specific machine characteristics, is suitable to be the object of various code analysis programs [54]. Therefore, in addition to decompilation, researchers have begun studying binary lifting, which converts machine code into various forms of intermediate representations. Regarding binary lifting, there have been many mature studies and methods [51], [55], [56].

C. Taxonomy of IoT Firmware Security Research

In Fig. 2, we present our taxonomy of IoT firmware security research perspectives. Based on different analysis perspectives, we classify IoT firmware security analysis methods into four categories: emulator-based test, automatic code analysis, network test via fuzzing and manual reverse engineering. Moreover, we analyze various challenges encountered in different research methods, which is shown in Table I.

The emulator-based test refers to performing security testing on the firmware after rehosting the device firmware with an emulator. Firmware is an integral part of the emulation process. However, obtaining firmware is not an easy task. In the process of emulation, due to the existence of external devices of each firmware, some firmware images depend on the interaction of external devices. How to learn or avoid the interaction of external devices is a challenge for emulating firmware. In addition, there is a lot of manual work in the process of simulating firmware, and how to use it for large-scale testing is also an issue. Emulating firmware is the first step in testing, and how to test firmware in a constrained emulation environment is also a challenge.

Automatic code analysis is a category of vulnerability detection methods based on program code. Similar to emulation, automatic code analysis methods rely on firmware code. Many well-established analysis solutions [89]–[91] work on source code, whereas in the context of IoT device firmware, there is often only binary code. Moreover, the architecture used by the device is also different. This means that automatic code analysis needs to have the ability to work across different platforms. There are also various vulnerabilities in IoT devices, which may only exist in some specific environments or pieces of code. How to use automatic code analysis methods to find such vulnerabilities is also a big challenge.

Communicating with the outside world through the network is the most significant difference between IoT devices and traditional embedded devices. Most of the existing network methods to test IoT devices focus on fuzzing. Therefore, we use the perspective of fuzzing theory to introduce and discuss various fuzzers for testing IoT devices. First, fuzzer algorithms mostly require users to provide initial input (communication messages). Moreover, IoT devices have strict format requirements for the input. Therefore, how to capture a set of messages that can communicate with the device and let the fuzzer generate messages that meet the requirements of the device is a challenge for network communication testing. In addition, due to the inability to perform monitoring methods

TABLE I
TABLE FOR IoT DEVICE FIRMWARE VULNERABILITY DETECTION METHODS

Perspective	Paper	Year	Challenge & solution	Architecture
Emulation-Based test (Section III)	SymDrive [57]	2013	Emu.C4S1	x86
	PROSPECT [58]	2014	Emu.C2S1	MIPS
	Avatar [59]	2014	Emu.C2S1	ARM
	SURROGATES [60]	2015	Emu.C2S1	ARM
	CostinFA [61]	2016	Emu.C2S2; Emu.C3S1; Emu.C4S2	Multiple
	Firmadyne [22]	2016	Emu.C1S1; Emu.C2S2; Emu.C4S2	ARM; MIPS
	Avatar^2 [62]	2018	Emu.C2S1	Multiple
	Pretender [63]	2019	Emu.C2S2	ARM mbed
	FirmFuzz [64]	2019	Emu.C4S2	ARM; MIPS
	FIRM-AFL [65]	2019	Emu.C4S3	ARM; MIPS
	PeriScope [66]	2019	Emu.C4S3	AArch64
	P^{2}IM [34]	2020	Emu.C3S1	ARM Cortex-M
	Laelaps [67]	2020	Emu.C3S1	ARM Cortex-M
	HALucinator [33]	2020	Emu.C3S1	ARM Cortex-M
	FirmAE [68]	2020	Emu.C3S1	ARM; MIPS
	FIFUZZ [23]	2020	Emu.C4S1	×
	Jetset [69]	2021	Emu.C2S2; Emu.C4S1	AMD 486; ARM
	ECMO [70]	2021	Emu.C2S1	ARM
	DICE [71]	2021	Emu.C2S1	ARM Cortex-M; MIPS M4K/M
	uEmu [72]	2021	Emu.C3S1	ARM
	IFIZZ [50]	2021	Emu.C4S1	ARM; MIPS
Automatic code analysis (Section IV)	FIE [19]	2013	Auto.C2S2	MSP430
	Firmalice [21]	2015	Auto.C3S1	ARM and PPC
	DiscovRE [73]	2016	Auto.C1S1; Auto.C2S1	x86, x64, ARM and MIPS
	Genius [74]	2016	Auto.C1S1; Auto.C2S1	x86, ARM and MIPS
	FirmUSB [54]	2017	Auto.C2S2	ARM and x86
	Gemini [75]	2017	Auto.C2S1	x86, ARM and MIPS
	VulSeeker [76]	2018	Auto.C1S1; Auto.C2S1	Multiple
	FirmUP [77]	2018	Auto.C1S1	MIPS32, ARM32, PPC32 and Inter-x86
	KARONTE [78]	2020	Auto.C3S1	ARM, AARCH64 and PPC
	CPScan [79]	2021	Auto.C3S1	MIPS and ARM
	SaTC [80]	2021	Auto.C3S1	Multiple
	PASAN [81]	2021	Auto.C3S1	ARM
Network test via fuzzing (Section V)	RPFuzzer [82]	2013	Net.C3S1	NA*
	BooFuzz [83]	2014	Net.C1S2; Net.C3S1	NA
	IoTFuzzer [84]	2018	Net.C1S1; Net.C3S2	NA
	SRFuzzer [85]	2019	Net.C1S2	NA
	Diane [86]	2021	Net.C1S2; Net.C3S1	NA
	Snipuzz [87]	2021	Net.C1S3; Net.C2S1; Net.C3S2	NA
	ESRFuzzer [88]	2021	Net.C1S2	NA

C*n*S*m* means that the method belongs to the*m*th solution under the *n*th challenge in the following discussion.

such as instrumentation to firmware, it is difficult for various fuzzer algorithms to enable guidance mechanisms such as coverage to optimize algorithms when testing devices. Moreover, since the operating environment of the device is similar to a black box, it is difficult to know the change of the device state during the testing process.

Unlike other research perspectives, manual reverse engineering is not a one-size-fits-all methodology. In general, manual reverse engineering requires researchers to obtain and analyze firmware code through other methods such as decompilation or disassembly, and reorganize its logic. After that, researchers need to perform detailed inspection and analysis where there may be flaws in the code to find out the vulnerabilities. Such methods require careful manual analysis of the case by the researcher. In this process, faced with different cases, the research methods and analysis methods adopted by researchers are very different. This means that the challenges encountered in different cases are also different.

Table I shows the collected state-of-the-art methods under different research perspectives. The column "Challenge & solution" in the table represents the challenges faced by the methods and the solutions proposed to solve those challenges.

III. Emulator-Based Test

The emulation of firmware has been an emerging technology in recent years. Emulators allow security researchers to run embedded software without hardware. Most dynamic testing is often accompanied by destructiveness. For example, fuzzing will cause the device to enter a crash state repeatedly. Therefore, for fragile and expensive IoT devices, emulation along with dynamic testing is an attractive solution. However, a successful emulation is not as simple as importing the firmware into the emulator. It will face the challenges posed by many real-world environmental constraints. For example, Non-uniform firmware development principles lead to the need for manual customization of the emulators required to emulation firmware. In addition, it is challenging to deploy various test tools on top of the emulator and test the running firmware.

In this section, we categorize the challenges in emulator-based test into four groups: acquisition of firmware; external hardware & peripheral; large-scale testing; and detecting vulnerability. In each subsection, we also introduce existing techniques and tools that can be different solutions to each challenge.

A. Acquisition of Firmware (Emu.C1)

Firmware is the key to the entire emulation process. Many existing methods discuss how to conduct emulations but ignore the firmware acquisition process. To protect the security of the product, most manufacturers do not release the relevant firmware. Moreover, various debug ports of IoT devices are blocked by manufacturers to prevent the leakage of firmware. Therefore, it is challenging to obtain the target firmware. To overcome this challenge, researcher have proposed different solutions as follows:

1) Download Firmware From Wild (Emu.C1S1): Before firmware security problems broke out on a large scale, many manufacturers published firmware images of some products on the Internet. For example, some researchers downloaded a large number of firmware images from different vendors [22], [73], [74]. However, the information about the downloaded firmware is often incomplete. Moreover, researchers often need to manually detect the format or architecture of the firmware. For example, in an IoT device of type BMES or SPES, the downloaded firmware might be a user-level application. Therefore, emulation of such firmware requires an additional operating system kernel.

2) Capturing via Network Traffic (Emu.C1S2): Intercepting the firmware of IoT devices through network traffic is also a possible approach. Many IoT devices have the ability to update themselves over the network. They download the latest firmware to the device itself by sending a request to the target server. By assuming a network sniffer, the address of the server and the requested message can be captured, and the firmware can be downloaded to the local by replay or other methods. However, this process may be encrypted or authentication (such as tokens and timestamps) may be required in the message.

In addition, firmware can be extracted from the network traffic of the device as it is being updated through a network traffic capture tool (such as DroidSniff [92], EtherApe [93], and NetworkMiner [94]) attached to the actual hardware. The problem is, the update process for devices is often not loading a fresh firmware, but applying a patch.

B. External Hardware & Peripherals (Emu.C2)

Emulating the firmware in a virtual environment is a challenge. Due to the lack of peripherals required for firmware, some firmware cannot be emulated via a virtualizer such as QEMU [47]. In addition, most devices customize their firmware, resulting in diverse standards of architecture and kernel. We divide existing solutions to this challenge into partial emulation and full emulation.

1) Partial Emulation (Emu.C2S1): Partial emulation is proposed to solve the problem that it is difficult to emulate the peripherals of embedded devices. The partial emulation executes the firmware in the emulator, and at the same time forwards the instructions sent by the firmware to the real peripherals. For example, PROSPECT [58] is a system that can overcome the problem of being unable to emulate peripheral hardware. It transparently forwards peripheral hardware access from the original host system to the virtual machine so that embedded software can be run without knowing how the peripheral hardware components are accessed. In addition, PROSPECT can dynamically analyze binary firmware codes in any analysis environment.

Similar to PROSPECT, AVATAR [59], an emulator framework, arranges the execution of the emulator with the real peripheral hardware to realize dynamic analysis. AVATAR improves the system's performance by forwarding I/O accesses from the emulator to the embedded device, and dynamically optimizing the code and data distribution between the two environments. In the evaluation [59], AVATAR was used to perform analysis on three different devices, all with success.

Based on AVATAR, SURROGATES [60] improves the efficiency and stability of emulation by strengthening the connection between external devices and firmware. Specifically, SURROGATES uses a customized low-latency field-programmable gate arrays (FPGA) bridge between the host's peripheral component interconnect (PCI) express bus and the system under test, allowing the emulator to fully access the firmware's peripherals. In addition, it optimizes the entire emulation system to overcome the problems that exist in previous emulators, such as interrupts handling, DMA, and clock changes.

The emulation of ECMO [70] is based on a novel technology of peripheral transplantation. Its core idea is to port the drivers of the specified peripherals to the target kernel binary instead of manually adding the emulation of each external device in QEMU. Specifically, ECMO transplants two components during the emulation, namely, the emulator model of the peripheral to QEMU and the device driver of the peripheral to the firmware kernel.

2) Full Emulation (Emu.C2S2): When peripherals of a device can be successfully emulated, full emulation re-hosts the firmware outside of the device. For example, for many Linux-based embedded systems, full emulation of their firmware is possible when full hardware setup documentation is available.

To fully emulate the firmware, Firmadyne [22] extracts the file system from the device and puts it into a precompiled general-purpose Linux kernel that runs with QEMU. Due to the scalability of full emulation, Firmadyne can perform a large-scale test of target firmware. Firmadyne discovered 14 unknown vulnerabilities on 69 firmware images. ARM-X [95] adopts a similar idea to Firmadyne in emulation, but requires users to provide more information and configuration. Moreover, ARM-X can only be used for ARM architecture devices, and requires rootfs and NVRAM in the firmware as support. Pretender [63] is a framework that automatically re-hosts the firmware of various embedded systems in a virtual environment. It records the interaction between the physical hardware and the firmware and then these records are used to build models for describing each peripheral by using machine learning. Therefore, Pretender can completely place the firmware in a virtualized environment and does not need to maintain long-term access to hardware devices.

C. Large-Scale Testing (Emu.C3)

The challenges of emulating peripheral devices also include the difficulty of large-scale testing. Some IoT device systems have no peripheral devices, but some systems may be connected to programmable logic controllers (PLC), FPGA, sensors, databases, and many other peripheral devices. A lot of partial emulation approaches often require manual efforts, which include heavy engineering works to extend the method to large-scale testing. The full emulation method such as Firmadyne also encounters some challenges in large-scale testing. It does not successfully extract the desired filesystem from every firmware image. For example, out of 23 035 firmware images, Firmadyne can only successfully extract filesystem from 9486 images.

1) Custom Methods (Emu.C3S1): For better large-scale testing, the existed methods [33], [34], [67], [68], [72] use different ideas to solve this problem.

To find out the reasons for low emulation rates, the failing cases of FIRMADYNE emulation [22] in large-scale data sets were analyzed [68]. Although a failure behaves differently in different cases, it is found that most of these problems can be avoided by simple heuristics. Therefore, an automated prototype named FirmAE that can enhance the emulation effect was further developed [68]. FirmAE proposes an arbitration emulation technology and uses heuristics to help arbitration technology solve the problems in various failed cases. It was mentioned in the evaluation [68] that the number of firmware that FirmAE can emulate is about 4.8 times to that of Firmadyne.

Furthermore, in order to scale up firmware testing, researchers have tried various methods to automatically solve the problem of interaction between external devices and firmware. Laelaps [67] is an emulator for running software on various microcontroller devices. It uses symbolic execution to assist peripheral emulation to infer the expected behavior of the firmware and generate appropriate inputs to guide the operation. Based on the inference, Laelaps can run a variety of firmware without prior knowledge. However, Laelaps only stays in the mode of symbolic execution for a short period, which can mitigate the influence of the problem called path explosion. In other words, Laelaps can only be supported by symbolic execution in a short-term execution.

Firmware developers sometimes use abstraction to develop code, such as hardware abstraction layer (HAL), which simplifies the development. Based on this observation, HALucinator, a method that provides high-level emulation through the HAL functions, was proposed [33]. It uses heuristics to locate the code belonging to the hardware abstraction layer (i.e., a vendor-provided API for interacting with the hardware) in the firmware and replaces it with manually created handlers. HALucinator takes firmware as input and produces a fully-featured emulation environment.

P^{2}IM [34] is a software framework that can test firmware independently without hardware. It abstracts peripheral devices and dynamically processes firmware I/Os based on an automatically generated model. It takes the target firmware and its memory map as input and fuzzes the code by feeding the input from an off-the-shelf fuzzer (i.e., AFL) to the peripheral device. Then P^{2}IM analyzes the device access patterns exercised during this fuzzing phase to infer details about the MMIO (memory-mapped I/O) interactions between the firmware and peripheral devices, which can execute the firmware without crashing.

uEmu [72] is used to emulate firmware with unknown peripherals. It tries to learn how to emulate firmware execution at each peripheral intervention point correctly. uEmu accepts images as input, expresses unknown peripheral registers as representation objects for symbolic execution analysis, and infers access rules for unknown devices. These rules will help it perform dynamic firmware analysis.

D. Detecting Vulnerability (Emu.C4)

The purpose of using emulation is to help us better perform vulnerability analysis and testing. However, as mentioned before, emulation technology faces various challenges and there are many limitations in practical applications. Therefore, how to perform vulnerability detection on firmware in virtual environments with various limitations is also a big challenge.

1) Error Handling in Driver (Emu.C4S1): A driver may encounter errors when communicating between the operating system kernel and the hardware. However, because this kind of error does not happen frequently, the error handling code is not taken seriously in most of the existing tests [49]. The methods [96] of finding vulnerabilities in firmware often ignore the characteristics of device drivers, making it difficult for them to find vulnerabilities in error handling code.

SymDrive [57] is a framework of symbolic testing for Linux device drivers. There have been some studies [46], [97] trying to implement symbolic execution for driver testing, but these systems require developers to manually adjust when testing new drivers. SymDrive solves this problem by using static-analysis and source-to-source transformation to reduce the effort of testing a new driver significantly. It inputs the C code for the Linux drivers and attempts to find program paths that violate user-written assertions. Static analysis helps SymDrive analyze crucial features of the driver code, such as entry point functions and loops. SymDrive uses a driver detection program to test the driver on the firmware for error handling.

Fuzzing is a commonly used technique in dynamic testing, but it is challenging for existing fuzzers to test the error handling effectively. The reason lies in the fact that errors are often triggered in corner situations, where fuzzing is hard to reach. Furthermore, testing error handling code often leads to execution crashes, preventing the fuzzer from tracing the error path deeply. IFIZZ [50] is a new error detection system that can solve these problems. It uses fuzzing to test the error handling code in the Linux-based IoT firmware. IFIZZ first adopts a binary-based automation method to recognize errors and their conditions in actual operation by analyzing errors in the firmware. After that, it uses state awareness and bounded error generation to detect deep error paths effectively. In its evaluation [50], the depth of error paths covered by IFIZZ in IoT devices is on average 7.3 times that of traditional error injection methods.

FIFUZZ [23] is another fuzzing framework for error handling code. Unlike IFIZZ, FIFUZZ uses a context-sensitive software fault injection method to effectively detect error handling codes in different contexts. As a result, it finds profound errors hidden in complex trigger situations. Compared with some popular fuzzing tools [98], [99], FIFUZZ can find vulnerabilities missed by these tools.

2) Web Interface (Emu.C4S2): Many IoT devices have built-in web application services. Web applications provide an interface for the outside world to interact with the devices when they are working.

There are a large number of vulnerabilities hidden in web applications of IoT devices [61]. A fully automated framework, aiming to use dynamic firmware analysis to automate the discovery of vulnerabilities in embedded firmware in an extensible manner, was implemented [61]. This kind of framework only detects the vulnerabilities in the built-in web interface of the embedded device when the firmware is running in emulation. It was reported that 225 unknown serious vulnerabilities were found in 45 firmware images.

In addition to the emulator, FIRMADYNE [22] provides a set of methods for testing web services in the firmware. It implements three primary channels of automatic dynamic analysis in its system to facilitate analysis. FIRMADYNE uses 60 persistent vulnerabilities to check the firmware in the data set for similar vulnerabilities. Each vulnerability is executed in order, and the vulnerabilities are successfully verified by checking the corresponding logs. There are many exploited vulnerabilities, such as buffer overflow, command injection, information disclosure, and denial of service.

FirmFuzz [64] is a framework for independent emulation and automatic dynamic analysis of Linux-based firmware. It uses a grey-box-based generational fuzzer, combined with static analysis and system introspection to detect vulnerabilities in the firmware. In order to effectively detect more profound vulnerabilities in web applications, FirmFuzz uses the program interfaces of these web applications as entry points to generate grammatically valid input. Meanwhile, it injects monitors into the firmware running environment to monitor the context.

3) Others (Emu.C4S3): In addition to error handling codes and web applications, the dynamic detection methods are used to analyze vulnerabilities in many other scenarios and specific code segments. For example, the complete system emulation is about ten times slower than the user-mode emulation (i.e., AFL) [28]. Part of the reason that the emulation throughput is not ideal is the software implementation of the memory management unit [65]. FIRM-AFL [65] solves the performance bottleneck caused by system emulation through a new technology called enhanced process emulation. Furthermore, it solves compatibility issues by enabling fuzzing POSIX (Portable Operating System Interface) compatible firmware, which can be emulated in the system emulator. With such enhancement, the throughput of FIRM-AFL is on average 8.2 times higher than that of a fuzzer under full system emulation.

Generally, most attacks against the kernel are mainly located on the boundary of system calls [66]. However, as shown in some exploiting cases, there are kernel compromise paths that do not involve system calls [100]. Attackers can gain control of the kernel by destroying peripheral devices. In order to detect and fix such vulnerabilities that occur on the hardware-operating system boundary, PeriScope [66] was proposed. It can perform a fine-grained analysis of the interaction between the device and the driver. PeriScope hooks the page fault handling mechanism of the firmware's kernel to detect and record the traffic between device drivers and related hardware. In addition, PeriScope provides a fuzzing framework called PeriFuzz, which can emulate attacks on peripheral devices.

IV. AUTOMATIC CODE ANALYSIS

Automatic code analysis techniques have been proved to be effective in software engineering security. For example, a tool was proposed [101] to use the code attribute graph in the source code to match code segments with the same pattern in other programs. Similar to this approach, CCFinder [89], CP-Miner [90] and DECKARD [91] also adopt pattern matching techniques to detect vulnerabilities. As a vulnerability mining technique that uses code features for analysis, code analysis methods can work without device entities. In addition, the code-matching solution is suitable for large-scale testing.

However, the false positive and the false negative rate are the evaluation criteria that code analysis cannot avoid. How to balance analysis efficiency and accuracy is a major challenge in code analysis. In addition, it is difficult to directly apply automatic code analysis methods to analyze firmware of IoT devices because real-world environments are different.

A. Lacking of Source Code (Auto.C1)

In Section III-A, we have introduced the difficulty of obtaining device firmware. Some automatic code analysis tools commonly used in software engineering are based on firmware source code. In a real-world environment where it is difficult to obtain device firmware, it is almost impossible to find the source code of the firmware.

1) Working on Binary Executable (Auto.C1S1): After recognizing the difficulty of obtaining source code, researchers focus on automatic vulnerability analysis of binary code [20], [73], [74], [76], [77], [102], [103]. Moreover, detecting vulnerabilities from a binary perspective may yield more accurate results than that from source code [73]. For example, when compilers get code from source to binary, the optimizer may introduce new vulnerabilities that are difficult to be identified by existing methods based on source code [73], [76].

When analyzing vulnerabilities on firmware binary, BinArm [102] uses a novel fine-grained multi-stage function matching method to find the candidate functions. It first collects a large database of vulnerable firmware programs as the search basis. Then, BinArm filters out some functions according to heterogeneous features and execution paths, and finds functions that may have vulnerabilities based on fuzzy graph matching. Experiments show that BinArm is three orders of magnitude faster than existing fuzzy matching methods, and it successfully finds 93 CVE vulnerabilities.

B. Architectures (Auto.C2)

Functions in binary executable may have similarities at the source code level, but they are quite different in the assembly version. For example, we might witness big differences in the structure of the control flow graph and the offsets of local variables on the stack. Different registers are also used for the same operation.

1) Features & Control Flow Graph (Auto.C2S1): To address cross-platform vulnerability search in general, many recent works [73]–[76], [103] proposed using different kinds of features and control flow graphs in the binary.

DiscovRE [73] mainly calculates the similarity between functions through the structure of the control flow graph, and uses this method to find potential vulnerabilities through functions with known vulnerabilities. Using a control flow graph structure allows DiscovRE's approach to avoid the impact of code differences caused by compilers, optimization levels, operating systems, and CPU architectures. In the prototype, DiscovRE supports four different instruction sets (x86, x64, ARM, and MIPS).

Genius [74] borrows ideas from the computer vision community to deal with similar problems. It directly uses CFG for matching, but chooses to extract higher-level numerical features from CFG and search for vulnerabilities based on numerical features. Such an approach makes Genius more immune to architectural changes. Moreover, the overhead of graph matching is usually expensive, and converting the control flow graph into higher-level features can significantly improve the efficiency of the searching process.

In addition, there are some works [75], [76] that optimize control flow graphs and code feature search algorithms through deep learning. To optimize the search algorithm, VulSeeker [76] is constructed by labeled semantic flow graph (LSFG) and the semantic-aware deep neural network (DNN) based function semantic generation. This approach captures more semantic information than using the control flow graph alone. Gemini [75] adopts deep neural network to generate embeddings of binary functions for similarity detection. It uses the graph embedding network to convert control flow graphs into embedding (i.e., numeric vector). By combining the graph embedded network with the Siamese network [104], the network can naturally capture two similar functions and make them close to each other.

2) Intermediate Representation (Auto.C2S2): There are also some methods [19], [54] using intermediate representation code to solve the problem of architecture. They first translate the code under different frameworks to the intermediate representation code in the same language, and then conduct vulnerability analysis on this basis. Many existing vulnerability analysis techniques already support a variety of different intermediate representation codes. For example, KLEE [45] supports LLVM intermediate representation (IR), but KLEE cannot run on binary executables.

A symbolic execution framework called FIE [19] was designed to detect vulnerabilities in firmware of the popular MSP430 microcontroller. FIE transfers source code into LLVM bytecode by using the Clang compiler. It then analyzes the bytecode by using the modified KLEE, which is a symbolic execution engine. It takes as input a piece of firmware, a memory map (e.g., regions such as RAM, ROM, and MMIO), and an interrupt specification that describes all locations where interrupts could be triggered. FIE will stop analyzing if the analysis exceeds pre-determined time or all possible states have been analyzed.

Similar to FIE, FirmUSB [54] uses a binary booster to convert binary files to bitcode and then deploys symbolic execution (based on KLEE) on Intel 8051 MCU. It analyzes USB firmware images by using domain-specific analyses to identify malicious behavior that targets USB (Universal Serial Bus) devices.

C. Various Bug (Auto.C3)

The automatic code analysis tools we introduced earlier are mostly based on a huge library of known vulnerabilities, using matching ideas (functions, control flow graphs or feature vectors) to find similar vulnerabilities in the library. However, such methods are difficult to work in some special cases, such as detecting vulnerabilities in the interaction between binary files, or in network application interfaces.

1) Specific Situation and Cases (Auto.C3S1): In addition to systematic methods such as matching analysis, many methods are designed for detecting specific types of vulnerabilities in IoT devices, such as integer overflow [105], [106], use-after-free [48], and buffer overflow [107]. Some other methods focus on detecting vulnerabilities in specific situations or code segments.

Firmalice [21] provides a framework for detecting authentication bypass vulnerabilities in binary firmware based on symbolic execution and program slicing. Firmalice observes that if an attacker can obtain the input of a privileged operation performed by the driver firmware, the authentication mechanism is either vulnerable or can be bypassed. Because there are many different manifestations of privileged operations in a device, Firmalice can customize the strategy of security analysis for each firmware.

Many existing devices have functions implemented through multiple binary file interactions, but static or dynamic security analysis has little effect on such multiple binary interaction services. Therefore, KARONTE [78] leverages static analysis techniques to perform multi-binary taint analysis and accurately finds the vulnerability by tracking the data flow in the firmware. It uses the commonality of the inter-process communication (IPC) paradigm to detect where an user input is introduced into the firmware and it also identifies various components. Then, KARONTE performs binary taint analysis by tracking the data flow between components.

Code pruning is common in customizing the Linux kernels of devices by IoT vendors. However, due to the inherent complexity of the Linux kernel and the lack of long-term maintenance, in the process of code tailoring, a manufacturer may mistakenly delete some necessary security operations [79], resulting in various types of vulnerabilities. CPScan [79] is a system that can automatically detect vulnerabilities caused by code pruning in an IoT kernel. A graph-based method can also effectively identify the deleted security operation (DSO) in the kernel.

There are many vulnerabilities in the web services of embedded systems. SaTC [80], proposed in 2021, is a security solution based on taint checking for Web services of embedded systems. The string text of a web interface is usually shared between the front-end and back-end binary files. Based on this observation, SaTC extracts these commonly used keywords from the front-end files and uses them as a positioning reference in the back-end files. Finally, SaTC analyzes whether there are dangerous operations in the user input stream based on data flow analysis.

Concurrent error is one of the most challenging software vulnerabilities to be detected and debugged [81]. Due to the non-deterministic conditions of triggering such errors, it is challenging to design a method for concurrent errors. A code analysis tool, PASAN [81], was designed to detect concurrency of peripheral access in embedded firmware. PASAN uses a parser-ready memory layout document to find the MMIO address range of each peripheral device automatically. Specifically, it uses the corresponding device driver to extract the internal state machine of the external device and combines the MMIO address range to automatically detect concurrent vulnerabilities in the access of the peripheral device.

V. Network Test via Fuzzing

Benefiting from the ability of IoT devices to communicate with the network, security researchers have found the method to test IoT devices without requiring the device firmware or source code. The new method runs testing through network communication. Most of the emulator-based test methods and static code analyses need a large amount of knowledge about the device, such as manufacture information, firmware architecture, and even the firmware's source code. Manufacturers often do not release firmware publicly due to device security issues. They also develop many new techniques to prevent their firmware from various reverse engineering attempts, such as the one to block device's debug mode. Compared with other test methods, fuzzing does not require much device information. This advantage makes fuzzing a mainstream choice for testing IoT devices over the network. However, the types of vulnerabilities that can be found by such methods are also limited due to the nature of fuzzing.

A. Input Acquisition & Format Requirement (Net.C1)

Fuzzing techniques usually require users to provide inputs as the original seeds to participate in the mutation phase. The generation-based fuzzer may not require a specific input, but it also requires users to customize the method of generating inputs before testing. The input refers to a complete communication message that includes the protocol. For network fuzzers that use communication packets as test cases, the original input(s) can be one message, multiple different messages, or a sequence of messages. For example, many devices need to verify customer information (e.g., login account, password, etc.) before performing various functional operations. At this time, the user needs to send more than two messages (one is for login verification) to complete an operational command. Therefore, using one message as input will make it difficult to bypass the verification mechanism, resulting in low test coverage. For some bridge IoT devices, they connect many different IoT devices. When the users need to use a device under the bridge, they need to select the device first and then send commands to it. Such an operation requires the cooperation of multiple messages, and a relative order between the messages is also required. It is difficult for the messages to be executed correctly without sequence matching for multiple different messages.

In addition, IoT devices have strict requirements for input (communication messages). The communication messages usually comply with the protocol used by the device (i.e., HTTPS, MQTT, etc.). The content in the message also needs to comply with the corresponding format (JSON, XML, etc.). Messages that do not meet these requirements will be rejected by the IoT device firmware in the syntax detection part. Therefore, it is a significant challenge for the fuzzers to ensure the generated test cases meet the grammatical requirements.

1) Companion App (Net.C1S1): IoTFuzzer [84] uses software such as MonkeyRunner to randomly click the UI in the app to send requests to the device automatically. IoTFuzzer first analyzes the UI elements of the app, and then uses data flow analysis to reversely identify the relevant program elements that send messages to the device from the control events. For example, the variable name in the function is identified through the switch button in UI. The name is highly relevant to the sending process of switch commands in the program. According to the fuzzing strategy specified by IoTFuzzer, these fields will be recorded once they are identified. If the request for changes involves the recorded protocol fields, IoTFuzzer uses function hooks to replace these fields so that it can achieve mutation in the fuzzing process.

Diane [86] is also a black-box fuzzing framework for testing IoT devices. It is observed that there are functions in the official companion app that can be used to generate optimal fuzzing input. Such a function can find it in the program after completing various verification codes in the execution position but before starting various character encryption or conversion [86]. Diane's core idea is to use a combination of static and dynamic analysis to capture such functions and then generate test cases to fuzz the target device.

Both IoTFuzzer and Diane find messages in the correct format from the official device app. Moreover, since their tests are all used to trigger the communication function of the app to send messages to the device, the commands issued in this way often consist of a series of messages with correct commands. Therefore, IoTFuzzer and Diane do not consider finding the correct format of the original message and the order among messages.

2) Human-Knowledge Guide & Learning Dataset (Net.C1S2): BooFuzz [83] uses human knowledge guidance to solve the problem of input problems. Before generating a network communication session, Boofuzz requires users to provide relevant network information of the target such as IP address and port number. It also requires a set of highly customized messages. In these messages, various attributes provided by Boofuzz can be used as features to present the messages. Users can use these attributes to further mark which characters in the message participate in the mutation phase and which characters will not change during the mutation process. Furthermore, users can specify which type of mutation strategy will be used. This method allows the user to specify the range of character segments for mutation operations in the message. In other words, if users are familiar with the message protocol and content format used by the device, they can instruct BooFuzz to generate various test cases without breaking the protocol and format.

Different from Boofuzz, SRFuzzer [85] captures a large number of web requests from the running devices, and then models the user-input semantics to generate test cases. SRFuzzer establishes a CONF-READ model to constrain the generated message sequence after observing the message sequence of the user's request to read the configure-related attributes initiated by the router device. Subsequently, to generate meaningful CONF requests and READ operations, SRFuzzer establishes a Key-Value model for each message content in the request. The Key-Value model can guarantee the generation of messages if the type of a variable is a domain name. The variable assigned to it will be a domain name. However, due to the lack of effective scheduling, the CONF-READ model makes SRFuzzer not be able to identify some critical vulnerabilities. To solve this problem, the team proposed ESRFuzzer [88] based on SRFuzzer in 2021. The improved ESRFuzzer can perform fuzzing in two modes. The first one is to use the Key-Value and CONF-READ models as the indicator to explore the vulnerabilities in these two types of situations. The second is to be used explicitly for testing when D-CONF Work aims at discovering a vulnerability that is ignored in CONF-READ mode. Both SRFuzzer and ESRFuzzer use models to standardize the format correctness of the generated input use cases. However, the standard model is based on collecting a large number of accurate communication records.

3) Inference (Net.C1S3): From the perspective of code analysis, the strict input syntax requirements are actually a large number of branch judgments of magic number and checksum. Among various types of fuzzing for binary, there are already fuzzers that can infer the role of each character in the input without the guidance of human knowledge. However, it remains a challenge to infer the input format required by IoT devices in a near-black-box test.

Snipuzz [87] takes the device's response message to infer the role of each character in the input. It first generates a sequence of probe messages based on modifying the original input. Then, the responses of the probe message sequence are used as a standard to segment the message snippets that play different roles in the message. When Snipuzz is generating test cases, each message snippet is used as a unit to mutate. Such method can ensure that while testing the syntax checking in part of the firmware, it can also generate some test cases that conform to the syntax.

B. Feedback Mechanism (Net.C2)

When testing executable binary programs, coverage-guided fuzzers have been widely proven to work well. They add instrumentation to the target program to obtain more internal execution coverage and then use it to guide the fuzzing process by exploring new execution paths. Since part of the information is collected from the inside of the program, this type of coverage-guided fuzzer uses the grey-box testing. However, it is too difficult to add instrumentation to IoT devices. Therefore, most of the existing network fuzzers [83], [84], [86] for IoT devices use black-box testing methods. In their fuzzing process, there is no feedback from the devices to guide the optimization of the mutation process.

1) Response-Based Guide (Net.C2S1): Snipuzz [87] first proposed using the device's response messages as feedback to guide the fuzzing strategy. Precisely, Snipuzz can determine whether different test cases have reached different execution paths in the device firmware through the content of the response messages. Based on this mechanism, Snipuzz uses a novel heuristic algorithm to detect the role of each byte in the message. Adjacent bytes have the same role in the initial message snippets, and can be packed together and linked to a basic unit of mutation. Snipuzz makes good use of the characteristics of IoT devices to build a set of fuzzing frameworks suitable for testing IoT devices.

C. Monitoring (Net.C3)

When we use vulnerability detection methods to test software programs, information such as program execution input and output and standard errors can be collected in time. Even while the program is running, various tools can be used to capture additional execution information such as program execution logs and CPU utilization. However, the kernel and binaries of IoT devices are closed to us. Even after each round of test, it is difficult to know whether a vulnerability in an IoT device has been triggered. Therefore, how to monitor the status of device is a challenge that various network testing methods need to face.

1) Local Monitor (Net.C3S1): Although we analog the firmware operation of IoT devices to a black box, some IoT devices themselves can provide some information as a reference for the state at runtime. For example, many routers provide users with a built-in web page to monitor and adjust the status of the device in real time while they are working. RPFuzzer [82] decides whether the router is abnormal by monitoring the CPU utilization, data regularity and the routers' system log. Such routers can provide rich information about traffic, and the local monitoring method is able to perform well in detecting the crash of routers and find abnormalities such as reboot and DoS attacks.

If the firmware can be run in a virtual machine, Boofuzz provides a proxy module that can monitor the running status of the virtual machine. It can restart the software in the virtual machine to continue the fuzzing test after the software crashes. However, network fuzzers such as BooFuzz are not designed for IoT devices. Their monitoring objects are more appropriate to software programs or servers. So the virtual machine monitoring service provided by BooFuzz is inefficient for IoT devices.

2) Heartbeat via Messages (Net.C3S2): When the device cannot provide additional internal execution information, the network testing methods often use the heartbeat detection to determine whether the device crashes. The heartbeat detection method sends a simple message to the target, and if the device does not reply to the message normally or does not reply to the message, it can be determined that the device is in the crash state.

Different network testing methods will customize the heartbeat detection method according to their own design. For instance, IoTFuzzer and Diane use the simple heartbeat message to detect whether the device has crashed every time after sending a message. Snipuzz sends a sequence of messages that are preset after each test to help the device return to the initial state of the test. It can find out whether a crash occurs through the network traffic. Moreover, since Snipuzz powers on the device through the smart plug, if the device crashes, Snipuzz can physically restart the device by temporarily powering off the smart plug remotely.

BooFuzz provides an uninterrupted monitoring proxy. After sending data to the target, Boofuzz will contact the proxy to determine whether the fault is triggered. If a fault occurs, the high-level information about the fault will be sent back to the session and stored.

VI. Manual Reverse Engineering

Unlike other research perspectives, the goal of manual reverse engineering analysis is to analyze and find vulnerabilities in a particular device or specific scenario. Researchers usually use manual solutions (e.g., reverse engineering) to launch various attacks on targets to detect whether the devices has exploitable vulnerabilities. Many interesting or high-threatening vulnerabilities in reality often require carefully constructed actions or inputs to trigger [108], [109]. Such vulnerabilities are often found only by delicate handwork like manual reverse engineering.

However, it is difficult to find the tools or solutions belonging to the same challenge with such a case-by-case research test. Therefore, in this section, we present some successful manual reverse engineering cases only.

In 2013, the remote firmware update (RFU) vulnerabilities on IoT devices and some real-world cases discovered in HP LaserJet printers were analyzed [110]. The printer vulnerabilities they discussed can be made into malware and embedded in harmless document formats such as PostScript. This type of malware can be delivered generally via PJL commands. It is worth noting that, this kind of defect does not only exist in printers. All kinds of devices that can perform remote firmware updates and use vulnerable third-party libraries may involve RFU.

An in-depth security analysis on a large number of medical devices that were deployed in various hospitals was conducted [108]. It is found that some of these devices rely on standard Wi-Fi security measures, which make them prone to man-in-the-middle attacks. In addition, such attack can steal medical records such as health insurance acquaintance bureau, medical history, and prescriptions, through medical devices.

The low-power Bluetooth devices have been widely adopted, which attracts the researchers' attention to the security of Bluetooth. By reverse engineering analysis, some vulnerabilities regarding remote code execution at the lowest level of the bluetooth low energy (BLE) protocol stack on firmware of multiple Bluetooth vendors were found [111]. By exploiting these vulnerabilities, an attacker can pair Bluetooth devices without authentication.

Problems on how to conduct safety research on connected cars were discussed [109]. A test platform that can test various related intelligent components in a car at a low cost was first proposed. Then, based on this platform, a set of attack chains that can penetrate from outside to the inside of a vehicle were designed. Using the vulnerabilities involved in the attack chain, the door could be remotely unlocked or the engine could be started. These vulnerabilities affected more than two million Mercedes-Benz smart-connected cars in China.

VII. Discussion

With the expansion of IoT market, manufacturers and security researchers are paying attention to IoT firmware security. For future research, researchers need to test IoT device firmware more systematically. Meanwhile, the security testing needs to cover all aspects of the firmware system and drivers. In this section, we discuss the directions for future development of security testing for IoT firmware.

A. Peripherals in Emulation Process

The emulator-based test process can be divided into two steps: emulation of firmware and dynamic test based on emulation. For the firmware emulator, a significant challenge that cannot be bypassed is to emulate many hardware peripherals. Although we mention solutions such as partial emulation as well as full emulation, there are many limitations when emulating devices in reality. For instance, the use of dedicated bridge hardware usually increases the complexity of performing emulation and makes it difficult to extend to other devices. In addition, the communication efficiency between peripheral devices and analog firmware also has room for improvement [60].

The full emulation often has relatively high requirements for the device firmware. Typically, it requires that the device firmware is based on a specific architecture or the firmware conforms to use a specific kernel [22], [95]. Moreover, although many existing tools attempt to apply simulation methods to large-scale testing from different perspectives, there are still many limitations. For example, HALucinator [33] uses HAL to conduct a full-feature emulation on target firmware. However, the HAL library is provided by various microcontroller unit (MCU) vendors, and HALucinator cannot test the system on chips (SoCs) with proprietary SDKs [72] (e.g., Samsung SmartThings [112] and Philips [113]).

B. False Positive in Automatic Code Analysis

In theory, most code analysis methods rely on matching and code scanning, which means that false positives are likely to occur [73]–[75]. In their evaluation, these code analysis methods report the number and proportion of false positives. The traditional way to confirm false positives is manual detection. However, the cost of manual testing is unaffordable when we come to large-scale testing.

C. Black-Box Network Test

The challenges of analyzing device firmware security using network testing methods stem from a black-box-like testing environment. Since a similar environment is not available, the method used to test the firmware of IoT devices cannot follow the idea of testing binary programs. For example, some grammar-based fuzzing algorithms can produce and mutate input that meets the stringent requirements of binary programs. Also, many fuzzer algorithms [43], [98] use instrumentation to obtain program execution coverage and guide seed selection and mutation strategies. Because of the closed operation of IoT devices, such methods cannot be used for network test.

There are some aforementioned works that attempt to solve this problem. For example, Snipuzz uses response messages as a feedback mechanism and builds a fuzzing framework that can infer the role of input characters. When RPFuzzer tests the router, it uses the router's own system log and CPU monitoring capabilities to monitor the firmware status. However, neither of the two methods can solve this problem very well. It depends on the type of the device itself to determine the type of the response messages of the device. It also suggests whether the device provides self-checking functions.

D. Cost of Manual Reverse Engineering

Manual reverse engineering of firmware can find various types of vulnerabilities lurking deep within the logic of the code. However, this operation is too expensive to process. Typically, it requires researchers to have products' firmware so that they can decompile the firmware for reverse engineering. Through repeated reading and guessing the meaning of the code, researchers can organize the operation logic of the code and identify the vulnerabilities. Therefore, it is difficult to scale up such manual detection-based methods to large-scale testing.

VIII. Future Direction

1) Automatic Emulation (Section III): When emulating the firmware, existing methods face many challenges. The large number of differential structures and configurations make emulating different firmware need a manual-work support by researchers. However, works such as [33], [34], [71], [72] are already trying to automate the simulation process. They provide us methods such as using machine learning, fuzzing, and symbolic execution to try to solve the interaction problems of peripheral devices. Although, we have discussed in Section VII-A that these methods have various limitations, they are still a great contribution to the development of fully automated emulation.

2) Hybrid Approach (Section IV): Various hybrid approaches have been proven to be effective when testing binary executives. Fuzzing methods combined with symbolic execution [114] can use the advantages of both methods to complement each other and get better test results. There are also some directed fuzzing methods [115], [116] combined with static code analysis, which utilize the fuzzer to test suspected vulnerable code regions found in static analysis. However, using a similar hybrid approach on IoT devices may double the challenges. For example, to use code analysis and simulator-based combined testing methods, it is necessary to solve the challenges of both methods. Nevertheless, such methods can also solve some problems for each other. For example, code analysis can be used to model the interaction of peripheral devices to help the emulator better emulate the firmware.

3) Multiple Ways to Monitor (Section V): Monitoring devices is a significant challenge for network test methods. Usually the network test environment of the firmware is considered as a black box, and only the output of the device can be observed. The local monitoring of routers by RPFuzzer gave a good inspiration. Monitoring device should not be limited to methods such as instrumentation that can directly obtain the execution status. Some other statuses of the device or records (i.e., logs) describing the status of the device should be included in the scope of monitoring. Moreover, some energy states, such as voltage and heat, can be considered within the scope of reactions of devices.

IX. Conclusion

In this survey, we systematically review and analyze existing solutions for detecting vulnerabilities in IoT firmware. We collect various methods that perform security analysis on IoT devices, and classify them into four categories, including emulator-based test, automatic code analysis, network test and manual reverse engineering. Based on each category, we discuss the challenges they encounter and summarize how existing methods address them. In addition, we discuss the limitations of current development of these security analysis solutions and possible directions in future work.

References

[1] P. Gandhi, S. Khanna, and S. Ramaswamy, "Which industries are the most digital (and why)?" [Online]. Available: https://hbr.org/2016/04/a-chart-that-shows-which-industries-are-the-most-digital-and-why, Accessed on: Apr. 1, 2016.

[2] K. Wiles, "First all-digital nuclear reactor system in the U.S. installed at Purdue University," [Online]. Available: https://www.purdue.edu/newsroom/releases/2019/Q3/first-all-digital-nuclear-reactor-control-system-in-the-u.s.-installed-at-purdue-university.html, Accessed on: Jul. 8, 2019.

[3] O. Friha, M. A. Ferrag, L. Shu, L. Maglaras, and X. C. Wang, "Internet of things for the future of smart agriculture: A comprehensive survey of emerging technologies," *IEEE/CAA J. Autom. Sinica*, vol. 8, no. 4, pp. 718–752, Apr. 2021.

[4] M. A. Ferrag, L. Shu, and K. K. R. Choo, "Fighting COVID-19 and future pandemics with the internet of things: Security and privacy perspectives," *IEEE/CAA J. Autom. Sinica*, vol. 8, no. 9, pp. 1477–1499, Sept. 2021.

[5] StatInvestor, "Internet of things-number of connected devices worldwide 2015–2025," [Online]. Available: https://statinvestor.com/ata/33967/iot-number-of-connected-devices-worldwide/. Access on: Feb. 10, 2022.

[6] C. Brook, "Travel routers, NAS devices among easily hacked IoT devices," [Online]. Available: https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/, Accessed on: Apr. 10, 2017.

[7] R. Ackerman Jr, "Lack of IoT security could undermine growth," [Online]. Available: https://www.rsaconference.com/library/blog/lack-of-iot-security-could-undermine-growth, Accessed on: Jan. 20, 2021.

[8] Keen Security Lab of Tencent, "Car hacking research: Remote attack tesla motors," [Online]. Available: https://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/, Accessed on: Sept. 19, 2016.

[9] C. Valasek and C. Miller, "Remote exploitation of an unaltered passenger vehicle," [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf. Access on: Sept. 11, 2021.

[10] Johnk, "$20M in bounties paid and $100M in sight," [Online]. Available: https://www.hackerone.com/ethical-hacker/20m-bounties-paid-and-100m-sight, Accessed on: Aug. 30, 2017.

[11] CISO Mag, "Tesla offers US$1 Million and a car as bug bounty reward," [Online]. Available: https://www.trustedsec.com/news/ciso-mag-tesla-offers-us1-million-and-a-car-as-bug-bounty-reward/, Accessed on: Jan. 13, 2020.

[12] M. Fernando, R. I. Augusto, and M. Jemimah, "Mirai botnet exploit weaponized to attack IoT devices via CVE-2020-5902," Security Intelligence Blog, Tech. Rep., [Online]. Available: https://www.trend-micro.com/en_us/research/20/g/mirai-botnet-attack-iot-devices-via-cve-2020-5902.html, Accessed on: Jul. 28, 2020.

[13] Paloalto, "2020 unit 42 IoT threat report," [Online]. Available: https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf. Access on: Sept. 10, 2021.

[14] J. Zhang, L. Pan, Q.-L. Han, C. Chen, S. Wen, and Y. Xiang, "Deep learning based attack detection for cyber-physical system cybersecurity: A survey," *IEEE/CAA J. Autom. Sinica*, vol. 9, no. 3, pp. 377–391, Mar. 2022.

[15] Y. Miao, C. Chen, L. Pan, Q.-L. Han, J. Zhang, and Y. Xiang, "Machine learning-based cyber attacks targeting on controlled information: A survey," *ACM Comput. Surv.*, vol. 54, no. 7, p. 139, Sept. 2022.

[16] G. J. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020.

[17] J. Y. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Comput. Surv.*, vol. 53, no. 6, p. 126, Nov. 2020.

[18] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Commun. Surv. Tutorials*, vol. 21, no. 2, pp. 1744−1772, Dec. 2018.

[19] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proc. 22nd USENIX Conf. Security*, Washington, USA, 2013, pp. 463−478.

[20] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation*, New York, USA, 2014, pp. 349−360.

[21] Y. Shoshitaishvili, R. Y. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proc. 22nd Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2015.

[22] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. 23rd Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2016.

[23] Z. M. Jiang, J. J. Bai, K. J. Lu, and S. M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proc. 29th USENIX Conf. Security Symp.*, USENIX Association, 2020, pp. 146.

[24] F. Corno, L. De Russis, and J. Sáenz, "On the challenges novice programmers experience in developing IoT systems: A survey," *J. Syst. Softw.*, vol. 157, p. 110389, Nov. 2019.

[25] A. Makhshari and A. Mesbah, "IoT bugs and development challenges," in *Proc. IEEE/ACM 43rd Int. Conf. Software Engineering*, Madrid, Spain, 2021, pp. 460−472.

[26] B. L. R. Stojkoska and K. V. Trivodaliev, "A review of internet of things for smart home: Challenges and solutions," *J. Cleaner Prod.*, vol. 140, pp. 1454–1464, Jan. 2017.

[27] F. Corno, L. De Russis, and J. Sáenz, "How is open source software development different in popular IoT projects?" *IEEE Access*, vol. 8, pp. 28337–28348, Feb. 2020.

[28] 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. 25th Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2018.

[29] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based IoT deployments," in *Proc. IEEE Symp. Security and Privacy (SP)*, San Francisco, USA, 2019, pp. 1362−1380.

[30] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, pp. 5, Jan. 2022.

[31] A. Qasem, P. Shirani, M. Debbabi, L. Y. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Comput. Surv.*, vol. 54, no. 2, p. 25, Mar. 2022.

[32] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for IoT networks-a survey," *ACM Comput. Surv.*, vol. 54, no. 9, p. 178, Dec. 2022.

[33] 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 *Proc. 29th USENIX Conf. Security Symp.*, USENIX Association, 2020, pp. 68.

[34] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proc. 29th USENIX Conf. Security Symp.*, USENIX Association, 2020, pp. 70.

[35] X. G. Zhu, X. T. Feng, X. Z. Meng, S. Wen, S. Camtepe, Y. Xiang, and K. Ren, "CSI-Fuzz: Full-speed edge tracing using coverage sensitive instrumentation," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 2, pp. 912–923, Mar.–Apr. 2022.

[36] X. G. Zhu and M. Böhme, "Regression Greybox fuzzing," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, New York, USA, 2021, pp. 2169−2182.

[37] X. G. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, to be published. DOI: 10.1145/3512345.

[38] K. P. Zhang, X. Xiao, X. G. Zhu, R. X. Sun, M. H. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," in *Proc. 44th Int. Conf. Software Engineering*,

Pittsburgh, PA, USA: IEEE, May 2022, pp. 1658−1668.

[39] X. G. Zhu, X. T. Feng, T. Y. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. L. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proc. ACM Asia Conf. Computer and Communications Security*, Auckland, New Zealand, May 2019, pp. 658−663.

[40] V. J. M. Manès, H. S. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[41] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. 24th Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2017.

[42] G. Klees, A. Ruef, B. Cooper, S. Y. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Toronto, Canada, 2018, pp. 2123−2138.

[43] lcamtuf, "American fuzzy lop," [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[44] C. Miller, "Fuzz by number – more data about fuzzing than you ever wanted to know," [Online]. Available: https://www.ise.io/wp-content/uploads/2019/11/cmiller_cansecwest2008.pdf, Accessed on: Mar. 28, 2008.

[45] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, San Diego, California, USA, 2008, pp. 209−224.

[46] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Newport Beach, USA, 2011, pp. 265−278.

[47] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Technical Conf.*, Anaheim, USA, 2005, pp. 41.

[48] J. Feist, L. Mounier, and M. L. Potet, "Statically detecting use after free on binary code," *J. Comput. Virol. Hack. Techn.*, vol. 10, no. 3, pp. 211–217, 2014.

[49] J. J. Bai, Y. P. Wang, J. Yin, and S. M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *Proc. USENIX Annu. Technical Conf.*, Denver, USA, 2016, pp. 635−647.

[50] P. Y. Liu, S. L. Ji, X. H. Zhang, Q. M. Dai, K. J. Lu, L. R. Fu, W. Z. Chen, P. Cheng, W. H. Wang, and R. Beyah, "IFIZZ: Deep-state and efficient fault-scenario generation to test IoT firmware," in *Proc. 36th IEEE/ACM Int. Conf. Automated Software Engineering*, Melbourne, Australia, 2021, pp. 805−816.

[51] A. Ruef and A. Dinaburg, "Static translation of x86 instruction semantics to LLVM with McSema," [Online]. Available: https://infocondb.org/con/recon/recon-2014/static-translation-of-x86-instruction-semantics-to-llvm-with-mcsema. 2014.

[52] Hex-Ray, "A powerful disassembler and a versatile debugger," [Online]. Available: https://hex-rays.com/ida-pro/. Access on: Apr. 4, 2022.

[53] OllyDbg, "OllyDbg is a 32-bit assembler level analysing debugger for Microsoft® Windows®," [Online]. Available: https://www.ollydbg.de/. Access on: Jun. 20, 2022.

[54] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. B. Butler, "FirmUSB: Vetting USB device firmware using domain informed symbolic execution," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Dallas, USA, 2017, pp. 2245–2262.

[55] "lifting-bits/remill" [Online]. Available: https://github.com/lifting-bits/remill. 2018.

[56] Angr, "Angr," [Online]. Available: http://angr.io/. 2018.

[57] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *Proc. 10th USENIX Symp. Operating Systems Design and Implementation*, Hollywood, USA, 2012, pp. 279–292.

[58] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proc. 9th ACM Symp. Information, Computer and Communications Security*, Kyoto, Japan, 2014, pp. 329–340.

[59] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. 21st Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2014.

[60] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proc. 9th USENIX Conf. Offensive Technologies*, Washington, USA, 2015, pp. 7.

[61] 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. Computer and Communications Security*, Xi'an, China, 2016, pp. 437–448.

[62] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Proc. Workshop Binary Analysis Research*, San Diego, USA, 2018.

[63] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *Proc. 22nd Int. Symp. Research in Attacks, Intrusions and Defenses*, Beijing, China, 2019, pp. 135–150.

[64] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proc. 2nd Int. ACM Workshop Security and Privacy for the Internet-of-Things*, London, UK, 2019, pp. 15–21.

[65] Y. W. Zheng, A. Davanian, H. Yin, C. Y. Song, H. S. Zhu, and L. M. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. 28th USENIX Conf. Security Symp.*, Santa Clara, USA, 2019, pp. 1099–1114.

[66] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J. P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary," in *Proc. 26th Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2019.

[67] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Proc. Annu. Computer Security Applications Conf.*, Austin, USA, 2020, pp. 746–759.

[68] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis," in *Proc. Annu. Computer Security Applications Conf.*, Austin, USA, 2020, pp. 733–745.

[69] E. Johnson, M. Bland, Y. F. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *Proc. 30th USENIX Security Symp.*, USENIX Association, 2021, pp. 321–338.

[70] M. H. Jiang, L. Ma, Y. J. Zhou, Q. Liu, C. Zhang, Z. Wang, X. P. Luo, L. Wu, and K. Ren, "ECMO: Peripheral transplantation to Rehost embedded Linux kernels," in *Proc. New York, USA, ACM SIGSAC Conf. Computer and Communications Security*, New York, USA, 2021, pp. 734–748.

[71] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of DMA input channels for dynamic firmware analysis," in *Proc. New York, USA, IEEE Symp. Security and Privacy*, San Francisco, USA, 2021, pp. 1938–1954.

[72] W. Zhou, L. Guan, P. Liu, and Y. Q. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *Proc. 30th USENIX Security Symp.*, USENIX Association, 2021, pp. 2007–2024.

[73] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. 23rd Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2016.

[74] Q. Feng, R. D. Zhou, C. C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Vienna, Austria, 2016, pp. 480–491.

[75] X. J. 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. Computer and Communications Security*, Dallas, USA, 2017, pp. 363–376.

[76] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. G. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Software

Engineering, Montpellier, France, 2018, pp. 896–899.

[77] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, Feb. 2018.

[78] N. Redini, A. Machiry, R. Y. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *Proc. IEEE Symp. Security and Privacy*, San Francisco, USA, 2020, pp. 1544–1561.

[79] L. R. Fu, S. L. Ji, K. J. Lu, P. Y. Liu, X. H. Zhang, Y. X. Duan, Z. H. Zhang, W. Z. Chen, and Y. J. Wu, "CPscan: Detecting bugs caused by code pruning in IoT kernels," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, New York, USA, 2021, pp. 794–810.

[80] L. B. Chen, Y. B. Wang, Q. P. Cai, Y. F. Zhan, H. Hu, J. Q. Linghu, Q. S. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *Proc. 30th USENIX Security Symp.*, USENIX Association, 2021, pp. 303–319.

[81] T. Kim, V. Kumar, J. Rhee, J. Z. Chen, K. Kim, C. H. Kim, D. Y. Xu, and D. Tian, "PASAN: Detecting peripheral access concurrency bugs within bare-metal embedded applications," in *Proc. 30th USENIX Security Symp.*, USENIX Association, 2021, pp. 249–266.

[82] Z. Q. Wang, Y. Q. Zhang, and Q. X. Liu, "RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing," *KSII Trans. Internet Inf. Syst.*, vol. 7, no. 8, pp. 1989–2009, Aug. 2013.

[83] J. Pereyda, "Boofuzz: Network protocol fuzzing for humans," [Online]. Available: https://boofuzz.readthedocs.io/en/stable/. Access on: Oct. 17, 2019.

[84] J. Y. Chen, W. R. Diao, Q. C. Zhao, C. S. Zuo, Z. Q. Lin, X. F. Wang, W. C. Lau, M. H. Sun, R. H. Yang, and K. H. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. 25th Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2018.

[85] Y. Zhang, W. Huo, K. P. Jian, J. Shi, H. L. Lu, L. Q. Liu, C. Wang, D. D. Sun, C. Zhang, and B. X. Liu, "SRFuzzer: An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities," in *Proc. 35th Annu. Computer Security Applications Conf.*, San Juan, USA, 2019, pp. 544–556.

[86] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices," in *Proc. IEEE Symp. Security and Privacy*, San Francisco, USA, 2021, pp. 484–500.

[87] X. T. Feng, R. X. Sun, X. G. Zhu, M. H. Xue, S. Wen, D. X. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, New York, USA, 2021, pp. 337–350.

[88] Y. Zhang, W. Huo, K. Jian, J. Shi, L. Q. Liu, Y. Y. Zou, C. Zhang, and B. X. Liu, "ESRFuzzer: An enhanced fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities," *Cybersecurity*, vol. 4, no. 1, p. 24, 2021.

[89] T. Kamiya, S. Kusumoto, and K. Inoue, "CCfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[90] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.

[91] L. X. Jiang, G. Misherghi, Z. D. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Software Engineering*, Minneapolis, USA, 2007, pp. 96–105.

[92] Evozi Team, "Droidsniff," [Online]. Available: https://github.com/evozi/DroidSniff. Access on: Feb. 6, 2022.

[93] Riccardo Ghetta and Juan Toledo, "EtherApe," [Online]. Available: https://etherape.sourceforge.io/. Access on: Jun. 13, 2022.

[94] NETRESEC AB, "NetworkMiner," [Online]. Available: https://www.netresec.com/?page=NetworkMiner. Access on: Feb. 8, 2022.

[95] S. Shah, "ARM-X firmware emulation framework," [Online]. Available: https://firmwaresecurity.com/2019/10/23/arm-x-firmware-emulation-framework/, Accessed on: Oct. 23, 2019.

[96] K. Cong, L. Lei, Z. K. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proc. Int. Symp. Software Testing and Analysis*, Baltimore, USA, 2015, pp. 361–372.

[97] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *Proc. USENIX Conf. Annu. Technical Conf.*, Boston, USA, 2010, pp. 12.

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

[99] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing Greybox fuzz testing coverage," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Software Engineering*, Montpellier, France, 2018, pp. 475–485.

[100] N. Artenstein, "Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets," [Online]. Available: https://www.blackhat.com/us-17/briefings/schedule/#broadpwn-remotely-compromising-android-and-ios-via-a-bug-in-broadcoms-wi-fi-chipsets-7603, Access on: Oct. 20, 2019.

[101] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Security and Privacy*, San Jose, USA, 2015, pp. 797–812.

[102] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Y. Wang, and A. Hanna, "BINARM: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Proc. 15th Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, Saclay, France, 2018, pp. 114–138.

[103] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Security and Privacy*, San Jose, USA, 2015, pp. 709–724.

[104] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. Lecun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "Siamese" time delay neural network," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 7, no. 4, pp. 669–688, Aug. 1993.

[105] T. L. Wang, T. Wei, Z. Q. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *Proc. Network and Distributed System Security Symp.*, San Diego, USA, 2009.

[106] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "BRICK: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proc. 4th Int. Conf. Availability, Reliability and Security*, Fukuoka, Japan, 2009, pp. 208–215.

[107] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The BORG: Nanoprobing binaries for buffer overreads," in *Proc. 5th ACM Conf. Data and Application Security and Privacy*, San Antonio, USA, 2015, pp. 87–97.

[108] S. Harit, "Breaking bad: Stealing patient data through medical devices," 2017. [Online]. Available: https://www.blackhat.com/eu-17/briefings/schedule/#breaking-bad-stealing-patient-data-through-medical-devices-8578, Accessed on: Dec. 6, 2017.

[109] M. R. Yan, J. H. Li, and G. Harpak, "Security research on Mercedes-Benz: From hardware to car control," Aug. 2020. [Online]. Available: https://i.blackhat.com/USA-20/Thursday/us-20-Yan-Security-Research-On-Mercedes-Benz-From-Hardware-To-Car-Control.pdf. Access on: Oct. 20, 2019.

[110] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Proc. 20th Annu. Network and Distributed System Security Symp.*, San Diego, USA, 2013.

[111] V. Kovah, "Finding new Bluetooth low energy exploits via reverse engineering multiple vendors' firmwares," [Online]. Available: https://i.blackhat.com/USA-20/Wednesday/us-20-Kovah-Finding-New-Bluetooth-Low-Energy-Exploits-Via-Reverse-Engineering-Multiple-Vendors-Firmwares.pdf. Access on: Feb. 3, 2022.

[112] Smartthing developer. "Build connected IoT experiences for millions of SmartThings users," [Online]. Available: https://smartthings.developer.samsung.com/. Access on: Sept. 10, 2021.

[113] Philips developer. "The latest from Philips Hue," [Online]. Available: https://www.philips-hue.com/en-us. Access on: Sept. 10, 2021.

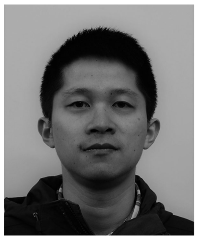
[114] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Y. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. 23rd Annu.*

Network and Distributed System Security Symp., San Diego, USA, 2016.

[115] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, “Directed Greybox fuzzing,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Dallas, USA, 2017, pp. 2329–2344.

[116] H. X. Chen, Y. X. Xue, Y. K. Li, B. H. Chen, X. F. Xie, X. H. Wu, and Y. Liu, “Hawkeye: Towards a desired directed grey-box fuzzer,” in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Toronto, Canada, 2018, pp. 2095–2108.

Xiaotao Feng is a Ph.D. candidate of computer science and engineering at the Swinburne University of Technology, Australia. He received the B.Eng. degree in computer science and engineering from South-West University and Deakin University, Australia in 2018. He is working on the exploitation of security vulnerabilities at Swinburne University of Technology, Australia. His research interests include software testing and IoT firmware security. He has published one paper on ACM Conference on Computer and Communications Security (CCS) 2021 about detecting vulnerabilities in IoT firmware. He also published some papers on ACM ASIA Conference on Computer and Communications Security (ASIACCS) and top journals, such as IEEE Transactions on Dependable and Secure Computing (TDSC) as the second author.



Xiaogang Zhu (Member, IEEE) received the Ph.D. degree in computer science and engineering at Swinburne University of Technology. He received the B.Eng. degree from Xidian University in 2012 and M.Eng degree from Xi’an Jiaotong University in 2015. Currently, he is a Research Fellow at Swinburne University of Technology, and focuses on searching vulnerabilities in programs. His research interests include detecting techniques such as fuzzing, machine learning and symbolic execution. He has published papers on top journals, such as *IEEE Transactions on Dependable and Secure Computing (TDSC)*, and conferences such as ACM Conference on Computer and Communications Security (CCS), and IEEE/ACM International Conference on Software Engineering (ICSE).

Qing-Long Han (Fellow, IEEE) received the B.Sc. degree in mathematics from Shandong Normal University, in 1983, and the M.Sc. and Ph.D. degrees in control engineering from East China University of Science and Technology, in 1992 and 1997, respectively.

Professor Han is Pro Vice-Chancellor (Research Quality) and a Distinguished Professor at Swinburne University of Technology, Australia. He held various academic and management positions at Griffith University and Central Queensland University, Australia. His research interests include networked control systems, multi-agent systems, time-delay systems, smart grids, unmanned surface vehicles, and neural networks.

Professor Han was awarded the 2021 Norbert Wiener Award (the Highest Award in Systems Science and Engineering, and Cybernetics) and the 2021 M.A. Sargent Medal (the Highest Award of the Electrical College Board of Engineers Australia). He was the recipient of the 2021 IEEE/CAA Journal of Automatica Sinica Norbert Wiener Review Award, the 2020 IEEE Systems, Man, and Cybernetics (SMC) Society Andrew P. Sage Best Transactions Paper Award, the 2020 IEEE Transactions on Industrial Informatics Outstanding Paper Award, and the 2019 IEEE SMC Society Andrew P. Sage Best Transactions Paper Award.

Professor Han is a Member of the Academia Europaea (the Academy of Europe). He is a Fellow of the International Federation of Automatic Control and a Fellow of the Institution of Engineers Australia. He is a Highly Cited Researcher in both Engineering and Computer Science (Clarivate Analytics). He has served as an AdCom Member of IEEE Industrial Electronics Society (IES), a Member of IEEE IES Fellows Committee, and Chair of IEEE IES Technical Committee on Networked Control Systems. He is Co-Editor-in-Chief of *IEEE Transactions on Industrial Informatics*, Deputy Editor-in-Chief of *IEEE/CAA Journal of Automatica Sinica*, Co-Editor of *Australian Journal of Electrical and Electronic Engineering*, an Associate Editor for 12 international journals, including *the IEEE Transactions on Cybernetics, IEEE Industrial Electronics Magazine, Control Engineering Practice, and Information Sciences*, and a Guest Editor for 14 Special Issues.

Wei Zhou received the B.Eng. and M.Eng. degrees from Central South University, in 2005 and 2008, respectively, all in CS, and the Ph.D. degree from the School of Engineering and IT, University of New South Wales, Australia in 2016. She is now a Research Fellow at Swinburne University of Technology. Her research interests include computer networks, network security, and fingerprint biometric.

Sheng Wen (Senior Member, IEEE) received the degree in computer science from the Central South University of China in 2012, and the Ph.D. degree from the School of Information Technology, Deakin University, Australia, in 2015. Since then, he has been a Lecturer with Deakin University, and then Associate Professor in Swinburne University of Technology, Australia. His research interests include modeling of virus spread, information dissemination, and defense strategies for the internet threats. He is also interested in the techniques of AI security and fuzzing on software. Since 2018, he is serving as the deputy director of Swinburne Cybersecurity Laboratory, Australia. He is also serving as the editors for several academic journals such as soft computing.

Yang Xiang (Fellow, IEEE) received the Ph.D. in computer science from Deakin University, Australia. He is currently a Full Professor and the Dean of Digital Research, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is also leading the Blockchain initiatives at Swinburne. In the past 20 years, he has published more than 300 research papers in many international journals and conferences. He is the Editor-in-Chief of the *Springer Briefs on Cyber Security Systems and Networks*. He serves as the Associate Editor of *IEEE Transactions on Dependable and Secure Computing, IEEE Internet of Things Journal*, and *ACM Computing Surveys*. He served as the Associate Editor of *IEEE Transactions on Computers* and *IEEE Transactions on Parallel and Distributed Systems*. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP).