Fuzzing of Embedded Systems: A Survey

JOOBEOM YUN, FAYOZBEK RUSTAMOV, and JUHWAN KIM, Sejong University, Republic of Korea
YOUNGJOO SHIN, Korea University, Republic of Korea

Security attacks abuse software vulnerabilities of IoT devices; hence, detecting and eliminating these vulnerabilities immediately are crucial. Fuzzing is an efficient method to identify vulnerabilities automatically, and many publications have been released to date. However, fuzzing for embedded systems has not been studied extensively owing to various obstacles, such as multi-architecture support, crash detection difficulties, and limited resources. Thus, the article introduces fuzzing techniques for embedded systems and the fuzzing differences for desktop and embedded systems. Further, we collect state-of-the-art technologies, discuss their advantages and disadvantages, and classify embedded system fuzzing tools. Finally, future directions for fuzzing research of embedded systems are predicted and discussed.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**; **Embedded systems security**; • **Computer systems organization → Firmware;**

Additional Key Words and Phrases: Firmware fuzzing, IoT devices, firmware analysis, fuzzing, embedded systems, software testing, symbolic execution, concolic execution

ACM Reference format:
Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. 2022. Fuzzing of Embedded Systems: A Survey. *ACM Comput. Surv.* 55, 7, Article 137 (December 2022), 33 pages.
https://doi.org/10.1145/3538644

1 INTRODUCTION

The **Internet of Things (IoT)**, a worldwide system comprising many computing devices that communicate mutually [136], has become an indispensable part of our lives. Especially, IoT technology is significantly used in critical infrastructure, the industrial sector, and smart home fields. Critical infrastructures, such as the power plant, water resources, and transportation systems, are very

This work was supported by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2022-2018-0-01423. In addition, this research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1F1A1065539) and the Ministry of Education (No. 2021R1F1A1051251).
Authors' addresses: J. Yun, F. Rustamov, and J. Kim, Sejong University, 209, Neungdong-ro, Gwangjin-gu, Seoul, Republic of Korea, 05006; emails: jbyun@sejong.ac.kr, {ffrustamov, juhwan}@sju.ac.kr; Y. Shin (corresponding author), Korea University, 145, Anam-ro, Seongbuk-gu, Seoul, Republic of Korea, 02841; email: syoungjoo@korea.ac.kr.

0360-0300/2022/12-ART137 $15.00
https://doi.org/10.1145/3538644

important to operate nations. Smart home devices bring convenience to most people, and healthcare systems are vital to patients. In addition, the **industrial Internet of Things (IIoT)**, which refers to a network of connected devices in the industrial sector, has received much attention in the fourth industrial revolution era. Most of these IoT devices are embedded systems having firmware and various applications.

However, security threats from the software vulnerabilities on embedded systems have been increasing along with these advancements. For instance, the Mirai malware [31] infected millions of IoT equipment and commanded them to initiate large-scale network attacks. Due to these attacks, hundreds of thousands of web servers across the globe became in denial of service. According to [93, 123], more than 1.5 billion cyber attacks that target 50 billion embedded devices ranging from cardiac pacemakers, cars, and various IoT devices have been detected in the first half of 2021. These attacks exploit software vulnerabilities in embedded device firmware, potentially discovered by well-crafted embedded fuzzing techniques.

As the increasing number of attacks target embedded devices, the techniques of embedded system fuzzing need to be appropriately studied in the research field. For instance, drones are one of the interesting embedded devices in both the public and private sectors [2, 3]. Unfortunately, drones have several security threats such as GPS spoofing attacks [113, 140], implementation errors, and software vulnerabilities. Several countermeasures [7, 14, 75, 142] against GPS spoofing attacks have been proposed, but other software errors have not received much attention. Drones must keep security and safety regulations to not have destructive impacts on our lives. For this, we must find and remove software errors before they are abused. An efficient technique to eliminate these software vulnerabilities is fuzzing, which was first invented by Miller et al. [85] in 1990.

Fuzzing (i.e., fuzz testing) is "an automated testing method that generates numerous test cases using random data (from files, network protocols, API calls, etc.) as software input to find the presence of exploitable vulnerabilities" [96]. Although fuzzing is an efficient technique for automatically detecting software vulnerabilities, directly applying this technique to embedded devices that are less visible and have strong hardware dependency is challenging [87, 145]. Although the number of embedded system fuzzers is less than that of traditional fuzzers owing to these reasons, several embedded-specific fuzzing tools (i.e., fuzzer) have been developed thus far, and we will analyze and discuss them in this article.

1.1 Motivation

The two main motivations for this survey are as follows:

(1) Many embedded devices have been developed worldwide. Numerous and severe vulnerabilities appear along with these advancements. Consequently, fuzzing for embedded systems or firmware has become increasingly attractive in IoT industries and security research societies. In particular, many embedded system fuzzing tools (e.g., Firm-AFL [145], Avatar2 [86], and IoTFuzzer [22]) have been proposed to identify bugs in embedded devices. However, there is no comprehensive guide for a security analyst to fuzz embedded systems or analyze a firmware. Therefore, we decide to provide an end-to-end guide to the analyst for fuzzing of embedded systems.

(2) No systematic review on **embedded system fuzzing (ESF)** has been conducted thus far. Although some survey articles [74, 76, 82] about traditional software fuzzing are available, they have not mentioned fuzzing for embedded systems or firmware. However, we think that a review of ESF is essential in an IoT era. Although some articles [49, 115, 145] about ESF describe an overview of some related works, they are only selected articles and thus are not comprehensive. Therefore, a comprehensive overview that surveys and analyzes state-of-the-art ESF works should be prepared.

1.2 Outline

The remainder of this article is organized as follows. Section 2 introduces the survey method, and Section 3 provides a general view of ESF. Section 4 classifies fuzzing techniques of embedded systems, and Sections 5–7 describe fuzzing steps, performance comparisons, and their applications, respectively. Section 8 presents several research challenges. Finally, Section 9 presents the conclusions of this article.

2 SURVEY METHOD

To conduct a comprehensive survey on ESF, we collected and selected related works extensively. In this section, we present the research questions, selection criteria, collection strategy, and a summary of the research.

2.1 Research Questions

We present an answer to the following ESF-related research questions.

(1) RQ1: What are the differences between traditional fuzzing and ESF?
(2) RQ2: What are the types of fuzzing techniques for embedded systems, and how do they differ from each other?
(3) RQ3: How do they solve the fuzzing challenges of embedded systems?
(4) RQ4: What are the research challenges and future research trends?

RQ1, described in Section 3, provides us time to consider the differences between traditional fuzzing and ESF and motivates us to write this article. RQ2 and RQ3, which are explored in Sections 4 and 5, enable us to conduct an in-depth investigation on ESF and evaluate the state-of-the-art techniques in this research field. Finally, we provide clues to RQ4 by answering RQ1 and RQ2, which are discussed in Section 8.

2.2 Collection Strategy

We have developed an archive of publications on automatic vulnerability detection for embedded devices; these include more than 105 papers from January 2008 to December 2021. This work aims to conduct a comprehensive survey of all ESF-related literature.

First, we searched for papers related to ESF from IEEE Xplore, ACM Digital Library, USENIX, Internet Society, Elsevier ScienceDirect Library, Springer Online Library, and Wiley InterScience. Thus, we have gathered research papers from these sources using search words such as "embedded/IoT fuzz," "firmware fuzz," "vulnerability embedded/IoT," and "embedded/IoT security" in their titles, keywords, and abstracts. Thereafter, we searched for keywords in online repositories. For example, we started to search for papers from the well-known DBLP computer science bibliography. In particular, DBLP [73] indexes more than 40,000 journals and 39,000 conferences, and contains more than 4.4 million publications and 80,000 monographs.

Second, after we read the abstract of each paper, we excluded unrelated papers. When paper selection could not be decided based on its abstract, we read the entire paper, with the selection criteria as follows:

- Include only computer science area
- Include papers written in English
- Include accessible literature via the Internet
- Include more than six pages by reputable publishers

As a result, we reduced the number of candidate publications to 72 papers based on the scope of our study. We refer to these collected papers as *primary studies* [67]. The four major online

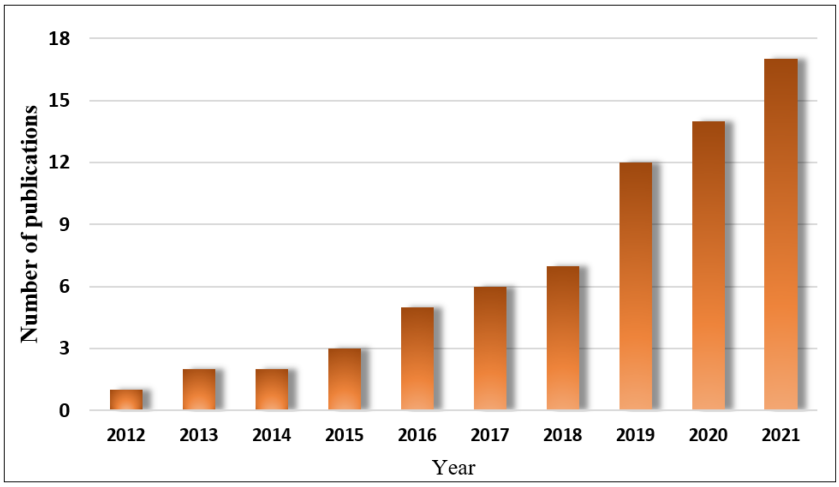

Fig. 1. Number of publications per year (last 10 years).

Table 1. Top Venues on Embedded System Fuzzing

Venue	Papers
Network and Distributed System Security Symposium **(NDSS)**	10
USENIX Security Symposium **(USEC)**	9
ACM Conf. on Computer and Communications Security **(CCS)**	7
IEEE Symposium on Security and Privacy **(S&P)**	5
IEEE ACCESS	5
Workshop on Offensive Technologies **(WOOT)**	3
Concurrency and Computation: Practice and Experience **(CCPE)**	2
International Conference on Testing Software and Systems **(ICTSS)**	2

sources are IEEE Xplore, ACM Digital Library, USENIX, and Internet Society. Although our survey may not cover all relevant papers, we are convinced that we collected enough papers to recognize state-of-the-art works and predict the research trends of ESF.

2.3 Summary on Publications

Figure 1 depicts the number of ESF publications from January 2012 to December 2021 (i.e., last 10 years). The graph demonstrates that the number of research papers related to our topic increases dramatically from around 2019. This curve shows an almost quadratic polynomial rise, which implies thriving attention to the subject. If this trend continues, it is likely to be over 18 papers in 2022.

The 72 selected primary studies were published in 24 different venues. This indicates that the coverage of treating ESF is comprehensive because IoT or embedded devices become more prevalent in our lives, and threats to them have become tremendous. These papers were presented to various venues. The papers presented in conferences and symposiums were 75.4%, academic journals (20%), workshops (3.63%), and technical reports (1.87%), respectively. Table 1 shows a list of top venues wherein at least two papers on ESF are presented.

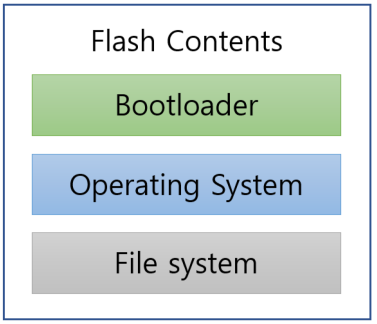

Fig. 2. Firmware architecture.

3 EMBEDDED SYSTEM FUZZING

In this section, we discuss ESF features in the literature to answer RQ1. We investigate papers related to the embedded system and its firmware in Section 3.1, types of embedded devices in Section 3.2, and comparison between traditional fuzzing and ESF in Section 3.3. In addition, we present the taxonomy of fuzzers in Section 3.4.

3.1 Embedded System and Its Firmware

An embedded system is a microprocessor-based system developed to enable a few dedicated functions as a part of a large electrical system [11, 72, 101]. It interacts with the physical environment and communicates with other devices in the environment [52]. This system commonly consists of a processor, memory, and peripherals. Software for operating an embedded system is stored in **read-only memory (ROM)** or flash memory chips of embedded systems [53]. This software stored in ROM or flash memory chips is referred to as *firmware*, which comprises the bootloader, an **operating system (OS)**, and a file system [119]. Firmware controls the hardware of an embedded device and is typically a fuzzing target of embedded systems. Typically, hackers fuzz the entire firmware, including OS and applications. In the case of monolithic firmware (i.e., type 3) devices, hackers input fuzzed test cases into the interfaces such as booting parameters or peripheral I/O channels. For instance, some fuzzing systems [54, 70, 86, 138] in Tables 4 and 5 read a full firmware as an input. In cases of other (i.e., type 1 and type 2) devices, fuzzers identify firmware components described in Figure 2 and I/O interfaces, then fuzz them separately. However, sometimes only applications can be fuzzed owing to target connectivity. We further discuss this in Section 5.2.

Firmware commonly consists of the kernel, a file system, and applications similar to that presented in Figure 2. The bootloader prepares the execution environment for the operating system (i.e., the kernel). The kernel is the core of the operating system, which controls the entire system, and applications use hardware of a computer via the kernel. The target of most embedded fuzzers is applications because applications are open to the Internet, accessible by an attacker, and liable to have vulnerabilities resulting from their diversity and immaturity. Still, a few fuzzers [49, 70, 115] using full-system emulation can test embedded systems' kernels. The two categories of applications as fuzzing targets are applications accessible from the network and those accessible through emulation. The former category is a few applications, whereas the latter is a lot. Another classification of fuzzing targets depends on the types of embedded devices, as described in Section 3.2.

3.2 Types of Embedded Devices

Embedded devices could be classified based on different criteria, such as performance, the performance of micro-controllers, and operating system types [120], in Figure 3. Based on

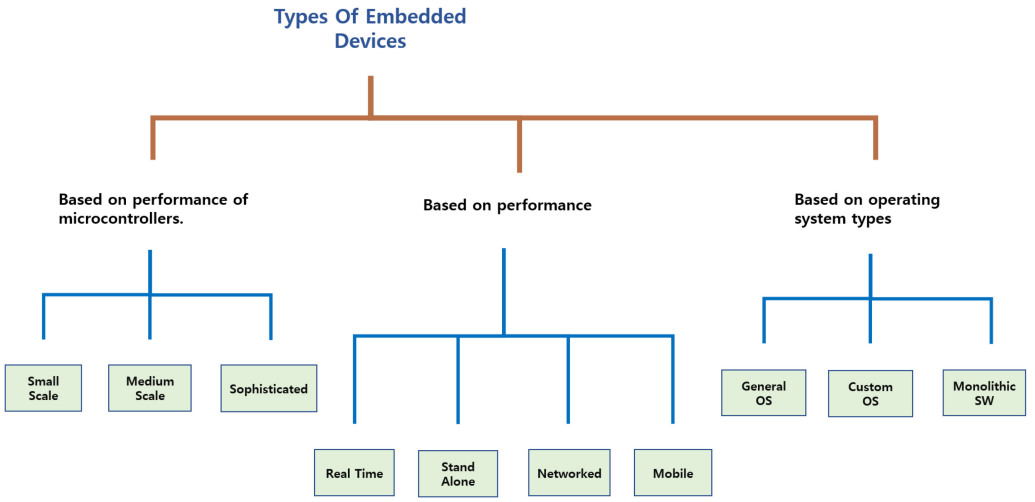

Fig. 3. Types of embedded devices [120].

their performance, embedded devices are classified into four categories: real time, stand-alone, networked, and mobile embedded devices. And based on their performance of micro-controllers, they are divided into small-scale, medium-scale, and sophisticated embedded devices. However, since we are interested in embedded software testing, we follow the operating system types. In terms of software testing, embedded devices can be classified into three categories according to their OS. This classification is based on Muench et al. [87], and we believe that it is reasonable as fuzzing (i.e., security testing) depends on the operating system. An operating system provides both the execution environment for applications and a source of vulnerability caused by its complexity.

3.2.1 Type 1 (T1): Embedded Devices with General-purpose OS. As general-purpose operating systems have a large number of functionality, compatibility, and continuous support from developers, they have also been used for embedded devices. However, the minimal shapes of an operating system are frequently used in embedded devices owing to performance limitations. For example, a combination of busybox [129] or uClibc [5] (i.e., lightweight user environments) and the Linux OS kernel has been widely used in the embedded world.

3.2.2 Type 2 (T2): Embedded Devices with Custom-built OS. The second class of embedded devices has a custom-built OS, such as a **real-time operating system (RTOS)** [122]. The RTOS is used for real-time applications that process data as it does not have buffer delay. These operating systems are suitable for low-power-consuming devices. For example, VxWorks [133] or QNX [15] is a representative RTOS and is widely used in embedded devices. Another example is uClinux (i.e., microcontroller Linux), which has no **memory management unit (MMU)**.

3.2.3 Type 3 (T3): Embedded Devices with Monolithic Software. This embedded device class has monolithic software that functions as a system and an application by compiling them. For example, many small-scale devices, such as SmartCards, GPS receivers, or thermostats, have this form. These devices typically do not have the hardware abstraction level, which hides the physical hardware and supplies programming interfaces.

Table 2. Comparison of Traditional Fuzzing and ESF

	Traditional Fuzzing	ESF
Hardware dependency	Weak	Strong
Crash detection	Easy	Difficult
Instrumentation	Easy	Difficult
Performance	Good	Limited
Scalability	Good	Bad

3.3 Comparison between Traditional Fuzzing and ESF

In this subsection, we compare traditional fuzzing and ESF. Although they have several differences, we identified five significant differences.

3.3.1 Strong Hardware Dependency. Embedded devices have various microcontrollers (i.e., MCU) and OSs. Common MCU types are more than 10 types including ARM, MIPS (32/64bit), Alpha, x86-64, IA-64, MSP430, PowerPC (32/64bit), SPARC-V8, and V9, and OSs are more than 5 including Linux, VxWorks, QNX, uClinux, and TinyOS [4], whereas a desktop system commonly has x86-64 CPU and Linux OS. Desktop fuzzers typically work on the target system, but embedded system fuzzers work on another system as the target system has low power and limited resources. Therefore, most embedded system fuzzers use emulation or re-host firmware. Desktop fuzzers' target programs are real-world datasets, such as Binutils [100], LAVA-M [35], or CGC datasets [114], whereas embedded system fuzzers' targets are executable programs in firmware.

MCU architectures have two main types: **reduced instruction set computing (RISC)** and **complex instruction set computing (CISC)**. Examples of RISC are ARM, MIPS, and PowerPC architectures, whereas an example of CISC is x86-64 architecture [13]. The RISC architecture aims to use simple instructions that are executed in one clock cycle. Conversely, the CISC architecture focuses on using fewer assembly instructions by constructing affluent instruction sets on the target hardware. For example, the CISC "ADD" instruction is divided into three separate instructions in RISC: "LOAD," "ADD," and "STORE." As the instructions are split up, the RISC architecture has the advantages of pipelining and better hardware usage. Thus, many embedded systems use the RISC architecture.

Although desktop fuzzing does not require to consider heterogeneous computer architectures, an embedded system fuzzer must support multi-architectures. Owing to the limited resource environment, the RISC architecture is commonly used for embedded devices. According to Chen et al. [20], 32-bit big/little-endian MIPS architecture makes up 79.4% of all the surveyed firmware images, whereas 32-bit big/little-endian ARM makes up 10.0% of them. In addition, eight other architectures exist, such as PowerPC, Motorola, and x86-64, in firmware images of an embedded system. Consequently, an embedded system fuzzer must support at least MIPS and ARM architectures as they constitute more than 90% of all firmware architectures.

Hundreds of fuzzers exist in the Linux platform, but they cannot be used directly on embedded devices as mentioned above. For example, extracting firmware from a Linux-based embedded device and fuzzing it with a popular fuzzer, such as AFL [139], might not function normally. Therefore, several fuzzing techniques for embedded systems have been proposed. We describe them in Section 4. In summary, traditional fuzzing has no hardware dependency, whereas ESF has strong hardware dependency, as presented in Table 2.

3.3.2 Crash Detection. As indicated in Algorithm 1, monitoring fuzzing results (i.e., crash detection) is a crucial ESF step. Crash detection is not easy in ESF, but it is comparatively easy in desktop fuzzing. According to Muench et al. [87], T1, T2, and T3 crash detection rates are 70%, 40%, and 0%, respectively. Therefore, the crash detection ratio in embedded systems is only 37% to desktop systems. This is because desktop systems are equipped with various detection mechanisms (e.g., error messages, security warnings, and system logs) on a crash or fault. A fuzzer in Linux detects crashes when an executing program is terminated by a fatal signal such as a segmentation fault. This is because a memory bug that overwrites the return address with an arbitrary value produces a segmentation fault or abort when it is accessed. This detection method is simple and efficient as only the fuzzer catches the signal without operating systems' intervention.

However, as embedded operating systems do not provide a fault generation mechanism in most cases, a fuzzer can rarely detect a fault or crash. Even worse, as embedded devices often do not have an output device (e.g., a monitor), a user cannot notice crash detection. Thus, embedded fuzzers use a *liveness check* using heartbeat messages, memory check tools (e.g., MemorySanitizer [116], AddressSanitizer [109], or ThreadSanitizer [110]), or debugging ports (e.g., UART [94] or JTAG [106]). UART is the abbreviation for universal asynchronous receiver transmitter, and JTAG is the abbreviation for joint test action group. (1) The *liveness check* (or probing) checks the embedded devices' states periodically. There are two types of probing [87]. Active probing adds special packets into the communication between a fuzzer and the device. This can affect the communication as the program has to respond to the packets. Meanwhile, passive probing only scans the device's states without modifying them. This is conducted by probing the responses presented by the device to the fuzzer or by identifying crash symbols. (2) Memory checking tools are accompanied by emulation. They check a memory utilization status every time and detect a security violation. Although these techniques are helpful for a desktop fuzzer, they are more beneficial for ESF. (3) The debug ports are used by debugging the device, which indicates that an analyst using a debugger program can have insight into the device. However, debug ports tend to be disabled nowadays.

3.3.3 Instrumentation. In software testing, instrumentation refers to the measure of a program's performance, to diagnose errors and to write trace information [57]. Instrumentation approaches have two types: source code instrumentation performed on the source code during static analysis and binary instrumentation performed on the compiled binary files [48]. Instrumentation is frequently used in desktop systems to gather code coverage information of the supplied inputs or trace taint analysis of interesting variables through source code instrumentation or binary instrumentation [90].

However, typically, the source code of firmware or applications is not available from embedded devices. Even worse, the program cannot be re-compiled as heterogeneous embedded devices have their CPU architecture, operating system, and I/O devices. Consequently, an embedded device tester has difficulty utilizing instrumentation. Rather than using source code instrumentation, the other solution is using dynamic binary instrumentation tools, such as Pin [77] or Valgrind [89]. For example, QSYM [137] uses coverage information when it tests program binary by leveraging the instrumentation techniques of Pin. However, this is only possible for general-purpose OS-based devices, except for embedded OS-specific devices or devices without an OS [87].

3.3.4 Performance and Scalability. Embedded systems have computational performance limitations resulting from low power and limited resources. Several techniques have been developed for current embedded systems to overcome performance limitations. First, embedded systems have a functional upgrade [52]. Since vendors reduce the production cost, they change and upgrade embedded systems by changing the software while keeping the hardware the same. This functional upgrade often includes functionalities to overcome the computational performance

Table 3. Example Function

```
void check(unsigned int input)    {
  if (input == 0xdeadbeef)
      assert();          // target
}
```

limitations. Second, an RTOS or real-time processing, which processes data and events within the time constraints, eliminates computational performance limitations [51]. Third, removing data dependency by the optimized compiler can improve computational performance. Data independency makes parallelization possible and contributes to better computational performance. Finally, having the program use registers, caches, and DMA instead of memory is better for computational performance.

Typically, fuzzing requires re-executing the **program under test (PUT)** to maintain a clean state for every test input. By reverting virtual machine snapshots, this technique is easy for desktop systems. However, this is difficult for embedded devices that require a substantial amount of time to reset the device. In addition, parallel fuzzing execution is possible in desktop systems. However, parallelization is frequently impossible in embedded devices, such as embedded OS devices or devices without an OS. Thus, repeated trials are impossible, or an embedded system takes a long time to fuzz.

The ability to handle increased demands is referred to as scalability, and this is parallel execution in the fuzzing field. Parallel fuzzing is typically supported in desktop systems but impossible in embedded devices owing to limited resources and economic reasons. As constructing an environment for parallel fuzzing in embedded systems requires many actual devices, it costs a lot. For example, AFL (i.e., a popular desktop fuzzer) supports parallel fuzzing through single-system parallelization or multi-system parallelization, which depends on the number of systems required. In summary, ESF cannot expect scalability in state-of-the-art works.

3.4 Traditional Taxonomy of Fuzzers

Traditional fuzzing has three categories based on the amount of information they require about the PUT during the test [74, 76, 82]. This information can include instrumentation, code coverage, the number of total paths, path constraints, or anything to steer test case generation. This section discusses these three categories, which can also be one of the taxonomy criteria for ESF.

3.4.1 Black-box Fuzzer. Black-box testing in software engineering only determines the program's interfaces, rather than the details of the PUT, such as data structure or algorithm [92]. Similarly, the black-box fuzzer randomly mutates the seed test cases based on predefined rules without identifying the PUT's inner information. This method has the advantages of simplicity and ease of use, but it is not smart. For example, in Table 3, the probability of reaching assert() function is $1/2^{32}$ as it attempts a brute-force execution (i.e., bit-flip mutation). To render a fuzzer smarter, white-box and gray-box fuzzers have been proposed.

3.4.2 White-box Fuzzer. White-box fuzzing is a technique for generating test cases based on the PUT's internal structure and information generated during execution [45]. Specifically, this fuzzing method leverages program analysis techniques, such as **dynamic symbolic execution (DSE)** [46], to generate a suitable test case. The DSE executes the program initially, generates path constraints, and solves the constraints. For example, DSE executes initially the if statement in

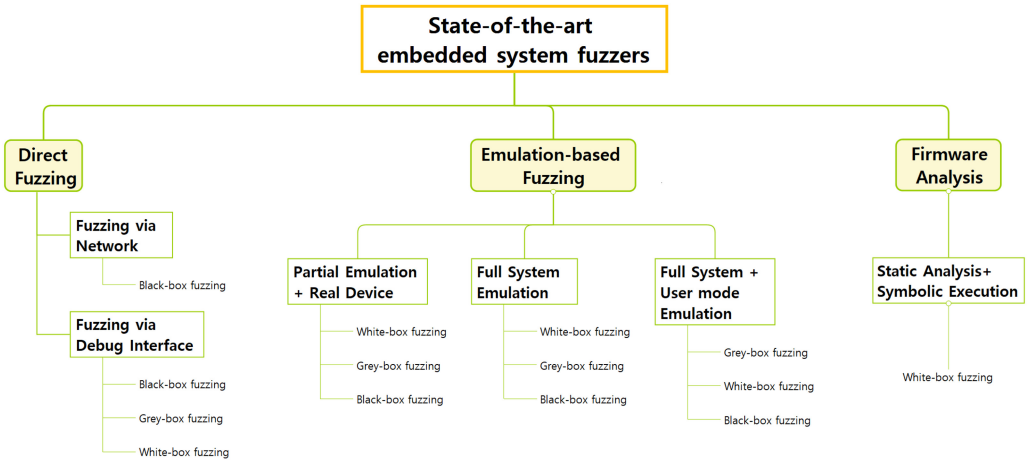

Fig. 4. Taxonomy of embedded system fuzzers.

Table 3, then it identifies that the constraint is "0xdeadbeef," and finally it makes the "0xdeadbeef" integer value to the input variable. Consequently, DSE can reach the assert() function within the second execution; hence, it is more efficient than black-box fuzzing, which requires 2^{32} executions.

The advantage of white-box fuzzing is that, in principle, every new test case will cover a new execution path in the program. Thus, white-box fuzzing provides better code coverage and can discover more software vulnerabilities than the black-box fuzzing approach. It is also able to expose vulnerabilities that are located in the deep area of the code. However, white-box fuzzing has a disadvantage; that is, it requires much time to solve the constraint or sometimes cannot solve the constraints due to various problems such as path explosion [71] or massive resource consumption. In addition, it requires a large amount of information, such as a source code or preliminary work for the information.

3.4.3 Gray-box Fuzzer. The gray-box fuzzer is in the middle of black-box fuzzer and white-box fuzzer, and it only requires partial information about the target program (i.e., PUT). This partial information is commonly the code coverage produced through instrumentation or taint-flow information through taint analysis. For example, the most popular gray-box fuzzer, AFL [139], collects path coverage information at runtime and efficiently utilizes the coverage map to mutate test cases (i.e., test inputs). Specifically, it provides weight to the test cases to find a new path, and the test cases mutate. Thus, the gray-box fuzzer is sufficiently efficient as it does not require preliminary analysis and gathers only partially related information. After AFL, more advanced gray-box fuzzers [16, 17, 23, 102] have emerged continuously, indicating the effectiveness of the gray-box fuzzer.

4 TAXONOMY OF EMBEDDED SYSTEM FUZZERS

From Sections 4 to 7, RQ2 and RQ3 have been answered by analyzing and describing the details of most embedded system fuzzers. Figure 4 depicts the taxonomy of embedded system fuzzers. Tables 4 and 5 summarize the most embedded system fuzzers that we will be analyzing, and Table 6 discusses the details of the embedded system fuzzers.

4.1 Classification

Figure 4 illustrates a taxonomy of state-of-the-art embedded system fuzzers. Traditional taxonomy is based on how much information a fuzzer requires or uses, but our first criterion is based on the

Table 4. Summaries of Embedded System Fuzzers (Closed-source)

Name	Year	Target	Interfaces	Monitoring	Platforms	Emu.[1]	Key Techniques	Tax.[2]
Koscher et al. [69]	2010	T3 Programs	Network (LAN)	Liveness check	Windows	N	Mutation-based network fuzzing	B
Mulliner et al. [88]	2011	T2 Programs	GSM Network	Liveness check, Log message	Linux	N	Generation-based network fuzzing	B
Prospect [61]	2014	T1, T2, T3 Programs	Firmware	Fault (exception) detection	Linux	S	QEMU, Network fuzzing	B
Costin et al. [28]	2014	T1, T2, T3 Programs	Firmware	Static analysis and Correlation	Linux	N	Static analysis, Fuzzy hashing	N/A
Surrogates [70]	2015	T1, T2, T3 Firmware	Firmware & JTAG	Crash detection	Linux	S	JTAG bridge, QEMU, S2E	W
Firmalice [112]	2015	T1, T2 Programs	Firmware	Backdoor detection	Linux, VxWorks	N	Static analysis, Symbolic execution	W
FirmUSB [54]	2017	T3 Firmware	Firmware	Malicious activity	Linux	N	Symbolic execution (FIE, angr)	W
IoTFuzzer [22]	2018	T1, T2, T3 Server apps	Network (LAN)	Liveness check	Android, Linux	N	Network fuzzing	B
Zheng et al. [146]	2019	T1 Programs	Firmware	Crash detection	Linux	F	Static and taint analysis, Coverage-based fuzzing	G
EM-Fuzz [42]	2020	T1, T2 Programs	Firmware	Memory checking	Linux	F	Coverage-based fuzzing	G
Fw-fuzz [43]	2020	T1, T2, T3 Server apps	Network (LAN)	Crash detection, Liveness check	Linux	N	Coverage-based fuzzing, Network fuzzing	G
FIRM-COV [64]	2021	T1, T2 Programs	Firmware	Crash detection	Linux	O	Coverage-based fuzzing	G
Aafer et al. [1]	2021	T1 Programs	APIs	Crash detection, Display and sound corruption	Android	N	Static analysis, Log-guided fuzzing	B
PASAN [66]	2021	T1, T2, T3 Platforms	Firmware	Concurrency bug detection	Linux	N	MMIO address identification, Concurrency analysis	W

[1]Emulation granularity (F: Full system emulation, N: No emulation, O: Optimized emulation, S: System-mode emulation).
[2]Taxonomy (B: Black-box fuzzing, G: Gray-box fuzzing, W: White-box fuzzing, N/A: Not available).

connectivity between the fuzzer and a target embedded system. We found out the connection types were very different and important when we tested the embedded systems. In other words, as the embedded system is hard to operate, emulate, and analyze, we select the connectivity as the first classification criterion. As a result, the first types of our taxonomy are direct fuzzing, emulation-based fuzzing, and firmware analysis. We set the second criterion according to the first type. And then we applied traditional taxonomy (i.e., the dependency on program information) as the third criterion. The classification's first criterion is the connectivity type each method adopts to fuzz a target device. We identify three types of connectivity: direct connection and fuzzing, emulation-based fuzzing, and firmware analysis. We discuss details in the following.

4.1.1 Direct Fuzzing. This approach directly connects the target device and tests the system firmware without intervention. This category has two types of fuzzing. (1) Fuzzing via a network

Table 5. Summaries of Embedded System Fuzzers (Open-source)

Name	Year	Target	Interfaces	Monitoring	Platforms	Emu.[1]	Key techniques	Tax.[2]
FIE [32]	2013	T2, T3 Programs	Firmware	Memory bug detection	Linux	N	KLEE	W
Avatar [138]	2014	T1, T2, T3 Firmware	Firmware & UART, JTAG	Vulnerability and Backdoor detection	Linux	P	S2E [26], QEMU [10], KLEE [19]	W
Firmadyne [20]	2016	T1, T2 Programs	Firmware	Exploit testing	Linux	F	Scanning	B
Muench et al. [87]	2018	T1, T2, T3 Server apps	Firmware & UART, JTAG	Crash detection, Liveness check	Linux	N, F, P	Avatar, PANDA [34], boofuzz	B
Avatar2 [86]	2018	T1, T2, T3 Firmware	Firmware & JTAG	Dynamic instrument, Fault detection	Linux	P	PANDA, QEMU, angr [111]	W
Firm-AFL [145]	2019	T1, T2 Programs	Firmware	Crash detection	Linux	Au	Coverage-based fuzzing	G
FirmFuzz [115]	2019	T1, T2, T3 Web apps	Network (LAN)	Emulation logs	Linux	F	Generation-based network fuzzing	G
Firmcorn [49]	2020	T1, T2 Programs	Firmware	Memory corruption detection	Linux	F	Static analysis, API fuzzing	W
KARONTE [104]	2020	T1, T2 Programs	Firmware	Static analysis and Interactions	Linux	N	Static analysis, Taint analysis	W
P^{2}IM [38]	2020	T1, T2, T3 Programs	Firmware	Crash detection	Linux	Ap	AFL, TriforceAFL	G
HALucinator [27]	2020	T1, T2 Programs	Firmware	Crash detection, Memory checking	Linux	F	AFL, angr, Avatar2	G
FirmAE [65]	2020	T1, T2 Programs	Firmware	Liveness check, Web service availability	Linux	Ar	Scanning, Web fuzzing	B
PGFUZZ [63]	2021	T3 Programs	Programs	Policy violation	Linux	Sim	Policy-guided fuzzing	W
DIANE [103]	2021	T1, T2, T3 Server apps	Network (LAN)	Response monitoring	Android, Linux	N	Network fuzzing	B
Jetset [59]	2021	T1, T2, T3 Programs	Firmware	Crash detection	Linux	F	AFL, angr, Guided symbolic execution	G
μEmu [147]	2021	T1, T2, T3 Programs	Firmware	Crash detection	Linux	F	AFL, S2E, KLEE	G
SNIPUZZ [39]	2021	T1, T2, T3 Server apps	Network (LAN)	Response monitoring	Windows	N	Network fuzzing	B
FIRMWIRE [55]	2022	T1, T2, T3 Firmware	Firmware	Crash detection	Linux	F	Avatar2, AFL++, PANDA [34]	G

[1]Emulation granularity (Ap: Approximate emulation, Ar: Arbitrated emulation, Au: Augmented emulation, F: Full system emulation, N: No emulation, P: Partial emulation, Sim: Simulation).
[2]Taxonomy (B: Black-box fuzzing, G: Gray-box fuzzing, W: White-box fuzzing).

[22, 29] can test only network applications, which is simple, but black-box fuzzing is only possible. (2) The other fuzzing via debugging interfaces, such as the UART [12, 83] or JTAG [106], can test the operating system and applications. Therefore, the second approach is more beneficial. However, it requires intensive labor (i.e., making debug environments) and is not always possible owing to removing the debugging interfaces. UART is a hardware equipment for asynchronous communication between a computer and a peripheral device. Meanwhile, JTAG is a method used

Table 6. Important Fuzzing Steps of Embedded Fuzzers

Name	Firmware Acquisition	Firmware Emulation	Fuzzing	Target Monitoring	Exception Analysis
Koscher et al. [69]	/	✕	/	✓	◯
Mulliner et al. [88]	/	✕	/	✓, /	◯
FIE [32]	◯	✕	◯	◯	◯
Avatar [138]	✓, /	✓	◯	◯	✕
Prospect [61]	✓	◯	✓	◯	◯
Costin et al. [28]	◯	✕	/	✕	◯
Surrogates [70]	✓	✕	◯	◯	✕
Firmalice [112]	◯	✕	◯	✕	◯
Firmadyne [20]	◯	◯	/	/	◯
FirmUSB [54]	✓	✕	◯	✕	/
IoTFuzzer [22]	/	✕	/	✓	◯
Muench et al. [87]	✓, /	◯, ✓	/	◯, ✓	✕
Avatar2 [86]	✓, /	✓	◯	◯	✕
Firm-AFL [145]	◯	/	✓	/	◯
Zheng et al. [146]	◯	◯	✓	/	◯
FirmFuzz [115]	◯	◯	✓	/	◯
Firmcorn [49]	◯	◯	◯	◯	◯
KARONTE [104]	◯	✕	◯	✕	◯
P^2IM [38]	◯	/	✓	◯	◯
HALucinator [27]	✓	◯	✓	◯	◯
EM-Fuzz [42]	✓	◯	✓	◯	◯
Fw-fuzz [43]	/	✕	✓	◯, ✓	◯
FirmAE [65]	◯	/	✓	✓	◯
PGFuzz [63]	◯	✕	◯	△	◯
DIANE [103]	/	✕	/	✓	◯
FIRM-COV [64]	◯	/	✓	/	◯
Aafer et al. [1]	/	✕	/	◯	◯
PASAN [66]	◯	✕	◯	✕	◯
Jetset [59]	◯	◯	✓	◯	◯
μEmu [147]	◯	◯	✓	◯	◯
SNIPUZZ [39]	/	✕	/	✓	◯
FIRMWIRE [55]	◯	◯	✓	◯	◯

The meaning of the symbols in table is as follows:

(1) Firmware acquisition
- ◯: Gathering from websites
- /: Direct connection
- ✓: Firmware extraction

(2) Firmware emulation
- ◯: Full system emulation
- /: Optimized or Customized emulation
- ✓: Partial emulation or Hybrid emulation
- ✕: No emulation

(3) Fuzzing
- ◯: White-box fuzzing
- /: Black-box fuzzing
- ✓: Gray-box fuzzing

(4) Target monitoring
- ◯: Crash detection
- /: Log analysis
- △: Policy violation
- ✓: Network checking
- ✕: Static analysis

(5) Exception analysis
- ◯: Unknown vulnerability found
- ✕: Not mentioned
- /: Backdoor found

for verifying the designs of circuit boards and test circuit boards after manufacturing. In addition, JTAG equipped with in-circuit emulators [25, 62, 84] can be used to diagnose and debug the system.

4.1.2 Emulation-based Fuzzing. Another approach uses emulation with firmware extracted from embedded devices through a UART or JTAG or downloaded from firmware update websites. Chen et al. [20] emulated 9,486 firmware, and Zheng et al. [145] covered 8,556 firmware. They used QEMU [10], which boots up the file system from firmware and runs a corresponding kernel on the file system. This emulation is a system-mode emulation, which executes some applications inside an operating system on an emulated hardware. However, every piece of hardware cannot be emulated owing to some restrictions on supported kernels and supported CPUs. Thus, partial emulation [60, 61, 70, 138] forwards peripheral API requests to the real device. This method has the same effect as system-mode emulation without full support, but intensive work is required to prepare the actual device and forwarding mechanism. User-mode emulation [145] migrates only user-level programs to the host OS by using a shared memory state (i.e., RAM file) with system-mode emulation. This migration is only possible when the kernel is the same in the system-mode emulator and user-mode emulator. The user-mode emulator has the advantage of having high throughput.

4.1.3 Firmware Analysis. The third approach is firmware analysis, which is useful when dynamic analysis that requires firmware execution or emulation is difficult in some embedded systems. For example, Firmalice [112] has three main components: a static program analysis module, a symbolic execution, and an authentication bypass check module. The static program analysis module develops a program dependence graph of the firmware and creates an execution path slice from the start point to the target point. The symbolic execution engine finds execution paths that arrive at the target point and constraint-satisfied inputs. The authentication bypass check module confirms whether the input results in an authentication bypass vulnerability. Another example, KARONTE [104], also analyzes multi-binary interactions (i.e., static analysis) and uses taint information (i.e., flow information). This approach commonly requires comprehensive program information; thus, white-box fuzzing is only possible.

4.2 Summary of Popular Fuzzers

We summarize the popular embedded system fuzzers in Tables 4 and 5. This includes the name, open year, fuzzing target, interfaces, monitoring or detection method, working platforms, emulation granularity, and availability. T1, T2, and T3 in fuzzing targets are three types of embedded devices described in Section 3.2. T1 is an embedded device with a general-purpose OS, T2 is an embedded device with a custom-built OS, and T3 is a monolithic software embedded device. Most embedded fuzzers target T1 and T2 devices as T3 firmware is difficult to obtain and emulate. T3 firmware is difficult to obtain as it is company property or has a customized firmware file format. The most common firmware file format is .bin, but customized firmware file formats are .npk (Mikrotik company), .chk (Netgear), .fw (Cambrionix company), and so on. Also, it is difficult to emulate as it has custom-made peripherals such as UART, **Real-Time Clock (RTC)**, **General-Purpose Input/Output (GPIO)**, **Analog-to-Digital Converter (ADC)**, **Digital-to-Analog Converter (DAC)**, and **Inter-Integrated Circuit (I2C)**. The *target* column has one of firmware, programs, and server applications: *firmware* indicates the firmware itself (i.e., the fuzzer tests the entire firmware), *programs* implies that the fuzzer can test all program files, and *server apps* indicates that the fuzzer targets server applications. The *interfaces* column describes the interfaces between a target device and a fuzzing system. The *monitoring* column shows how to detect or monitor the crash or abnormal behavior of the target device. The *emulation granularity* column is described in Section 5.3.

ALGORITHM 1: Fuzzing of embedded systems

Input: $\mathbb{T}, \mathbb{S}$ ▷ target, seed
Output: $\mathbb{B}$ ▷ a set of bugs

```
1: if (IsDirect()==True) then
2:     Interfaces = IndentifyInterfaces(𝕋)
3: else
4:     Firmware = AcquireFirmware(𝕋)
5:     FS,kernel = Extract(Firmware)
6:     Interfaces = EmulateFirmware(FS,kernel)
7: end if
8: Inputs = InputGen(𝕊)
9: while !abort-signal do
10:    Results = SendInputs(Interfaces, Inputs)
11:    Exception = TargetMonitoring(Results)
12:    fuzzinfos = AnalyzeException(Exception, Inputs)
13:    if (Exception==Bug) then
14:        Report(𝔹)
15:    else
16:        Inputs = Schedule(Inputs, fuzzinfos)
17:    end if
18: end while
```

5 FUZZING STEPS IN DETAILS

Algorithm 1 depicts the algorithm of ESF. ESF requires additional preprocessing steps, such as acquiring firmware or identifying interfaces. It then applies traditional fuzzing steps, such as input generation, sending inputs, and target monitoring. Table 6 presents the fuzzing steps of popular embedded system fuzzers. We provide their details in the following subsections.

5.1 Firmware Acquisition

Although some works [20, 28, 49, 145] gathered many firmware images via a web crawler, all firmware images are difficult to collect as they are companies' property. Sometimes, firmware images can be extracted via UART or JTAG, but device vendors tend to disable debug ports nowadays. Another method, dumping flash memory of embedded devices, is sometimes possible. However, it requires desoldering the flash memory and hence becomes more difficult. This firmware acquisition is critical to emulation-based fuzzing as subsequent works can no longer proceed. In summary, firmware acquisition is an indispensable step in emulation-based fuzzing.

To overcome this limitation, several direct fuzzing techniques [27, 86, 138] have been investigated. IoTFuzzer [22] is targeted at network protocols for IoT devices. In the case of network-based fuzzing, fuzzing targets are mainly network applications visible from the network. This narrow view is a limitation of the network-based fuzzer, whereas fuzzing through firmware emulation can test any firmware applications. In addition, debug ports, such as UART or JTAG, can be used to connect a fuzzing system to the target device. For example, [12, 83] used the UART connection for a fuzzer, and Avatar2 [86] used a debug interface when it orchestrated an emulator with the actual device.

5.2 Interfaces

In this subsection, we explain the fuzzing interfaces to the devices. As presented in Table 4, the fuzzing interfaces of embedded devices are firmware, network, UART, and JTAG. As the fuzzing

strategy varies according to the fuzzing interface, several target connection methods and their advantages should be described.

5.2.1 Firmware. Many embedded system fuzzers have used firmware extracted from actual devices [124] or acquired from the Internet as a fuzzing target. After acquiring firmware, it is used as an input to emulation [20, 86, 145] or further analysis [112]. When the firmware is used as an input to emulation, it can be run and analyzed dynamically in the execution environment. Emulation-based fuzzing [27, 42, 49, 87, 115, 145] benefits from testing much firmware in a short time without difficulties in preparing real devices or creating experimental environments.

Muench et al. [87] explained the effectiveness and efficiency of emulation-based fuzzing for embedded devices. Specifically, through comprehensive experiments, they proved that fuzzing via a fully emulated system is faster and more accurate than that via a physical system. They reported four advantages. First, emulation-based approaches can easily add fault detection, such as heuristic methods. This advantage is beneficial to the embedded fuzzing environment with a crash detection difficulty. Second, as an emulator typically has a higher clock speed than a physical device, emulation-based fuzzing exhibits high throughput. Third, an emulation-based approach has time efficiency as rebooting the physical device is time consuming compared to restarting emulation. Finally, emulation has the advantage of clearing the target system easily as it uses a snapshot and reverts the target to an initial state. Firmware and its emulation are closely related. Thus, we describe popular emulators and their advantages in Section 5.3.

5.2.2 Network. Despite that emulation is the correct option, it requires intensive work, such as firmware acquisition, firmware unpacking, and executable analysis. Moreover, emulation is not always possible, especially when at least one of the works is unsolvable. Another solution is fuzzing embedded devices via a network, commonly referred to as network fuzzer [43]. Network fuzzing is commonly a black-box fuzzing, and it does not analyze the program source code or binary code, but only captures network packets. For instance, Prospect [61] randomizes only 1 byte at a random position per captured packet and replays the communication. When the fuzzer detects an exception, it stores the network packets for further analysis. Meanwhile, IoTFuzzer [22] analyzed the protocol used in the communication between IoT applications and physical devices and performed protocol-guided fuzzing. IoTFuzzer performs a continuous liveness check to detect a fault. However, this type of fuzzing has three limitations: (1) target applications should be accessible from the network, (2) it is often slow, and (3) firmware crashes or faults are difficult to detect due to silent memory corruptions [87].

5.2.3 UART. UART [131] is an interface for asynchronous serial communication, one of the most common protocols found in embedded devices. Hence, it has been used as a fuzzing bridge. For example, beSTORM [12] is an automated dynamic testing tool (i.e., fuzzer) to verify the security of any software or product that uses high-speed UART. Although this tool is a black-box fuzzer, it typically produces satisfactory results as it includes many protocol standards, such as UART, and supports auditing the specific protocol, thereby performing smart fuzzing. Other researchers [83] used an Arduino [6] to fuzz the debug pins on an embedded device and then used the Arduino as a USB-to-serial converter to communicate to the device and obtain a shell. Further, UART is commonly used as a fuzzing bridge and is often used as a connecting medium to facilitate emulation. In [87, 138], UART was used as a communication channel to transmit the memory state. In addition, as UART supports serial communication, it can be used as a data channel, such as sending interrupts or data.

5.2.4 JTAG. The JTAG is a standard interface to verify the design and test pins of printed circuit boards [130]. As the JTAG is designed to access the hardware resources of the target device, to our

knowledge, JTAG fuzzing does not exist. However, the JTAG can be used as a connecting medium to facilitate emulation similar to the UART. In several research studies [70, 86, 87, 138], there is forwarding peripheral input and output to the actual device through a JTAG debugger with an in-memory stub. Additionally, JTAG interfaces support accessing the CPU state, registers, and memory when communication is required between emulation and the physical device. In summary, the JTAG is commonly used for connecting an emulator and a real device, rather than a fuzzing target interface.

5.3 Firmware Emulation

5.3.1 Emulation Granularity. Emulation has several types. In system-mode emulation, firmware images are emulated similar to a virtual machine, and all peripherals should be supported. This emulation is a basic emulation provided by QEMU [10]. However, it is heavyweight and slower than user-mode emulation. User-mode emulation executes processes compiled for one **central processing unit (CPU)** on another CPU to provide convenience and efficiency. Hence, it supports system call translation, signal handling, and threading. System call translation indicates that the system calls of a PUT can be converted to those of the host. Further, user-mode emulation can redirect all signals in the PUT to the host's signal handler. It can also emulate the clone system call and create a real host thread for each emulated thread. In summary, system-mode emulation emulates firmware as an entire system, whereas user-mode emulation emulates a process on another CPU for convenience.

However, system-mode emulation requires manual work, such as file system extraction, customized kernel configuration, and running system-mode QEMU. Thus, the Firmadyne [20] authors conducted full-system emulation by adding automatic works to system-mode emulation. The added automatic works are extracting the file system from the firmware, configuring the customized kernel, developing virtual network interfaces, and running system-mode QEMU. Meanwhile, some works [86, 138] used partial emulation as the emulations mentioned above are not perfect and accurate. Partial emulation, also referred to as hybrid emulation, combines an emulation environment and an actual device, and it forwards all items that emulation cannot handle to actual devices. Consequently, partial emulation has the advantages of accuracy and scalability, but it has a disadvantage of requiring an actual device.

Another hybrid emulation is the augmented process emulation used by Firm-AFL [145]. Augmented process emulation combines system-mode emulation with user-mode emulation to enhance fuzzing throughput. Thus, they implemented memory mapping, a RAM file that is a memory-mapped file shared between system-mode and user-mode emulator, and system call redirection. Consequently, Firm-AFL showed high-throughput fuzzing results and found two unknown vulnerabilities. Similarly, the authors of P^2IM [38] introduced an approximate emulation that feeds acceptable inputs to the firmware. Specifically, the emulator provides firmware with suitable inputs from the emulated peripherals when requested. These inputs do not need to be the same as the output from a real peripheral, but they should satisfy the firmware's checking function to execute firmware successfully. In addition, P^2IM automatically generates approximate emulators for IoT devices equipped with various peripherals. This emulator renders it possible to roughly execute firmware, which is sufficient for fuzzing and analyzing firmware's control or data flows. However, approximate emulation cannot ensure functional accuracy. Hence, it cannot be used for a program that requires high accuracy.

5.3.2 Popular Emulators. Emulation-based fuzzers must have an emulator. Considering the additional costs of creating an emulator from scratch, developers often utilize an existing emulator. Although QEMU [10] is the most popular emulator, few other emulators are available. Table 7

Table 7. Comparison of Emulators

	QEMU	Unicorn	Simics
Emulation	CPU & devices	CPU	CPU & devices
Complexity	Heavy	Light	Heavy
Flexibility	Less flexible	Flexible	Less flexible
Security	No	Yes	Yes
Availability	Open source	Open source	Closed source

compares the popular emulators QEMU, Unicorn [91], and Simics [79]. QEMU and Simics emulate a full system, including the CPU and peripherals, so they are heavy. Meanwhile, Unicorn emulates only the CPU, so it is lightweight. As Unicorn emulates the CPU without the execution environment, it is flexible. However, it requires other implementations when more accurate emulation is necessary. Although QEMU has many related vulnerabilities, Unicorn and Simics have no vulnerability to date.

QEMU (i.e., Quick Emulator) is a generic and open-source emulator with unique features to emulate the processor and peripheral devices, to support multi-architectures and multi-platforms, and to be maintained eagerly. It translates several basic blocks simultaneously, provides a set of virtual hardware and devices, and runs various operating systems. QEMU can also emulate user-level processes, which execute processes compiled for one CPU on another CPU. Since QEMU [10] was invented, many fuzzing researchers have used QEMU as an emulation engine. Despite the insufficient functionalities of QEMU, several researchers combined it with other features, such as actual hardware support [70, 138] or augmented process emulation [145]. Authors of Firmadyne [20] added four features to address the limitations of current emulation. First, they modified QEMU to support NVRAM-related functions as the original QEMU did not provide such support. Second, QEMU booted up the extracted filesystem with their pre-compiled kernels as it could not support every kernel extracted from all firmware. With these pre-compiled kernels for ARM little-endian, MIPS little-endian, and MIPS big-endian platforms, they can cover 90.8% of their dataset. Third, their QEMU had a learning mode, in which their modified kernels recorded all system interactions. Finally, Firmadyne launched the extracted firmware image and performed network connectivity checks.

Unicorn Engine [80, 91] is a lightweight multi-architecture CPU emulator framework. It is derived from QEMU, but it has several advantages over QEMU, as presented in Table 7. First, Unicorn is a framework wherein anyone can develop tools on it. Second, it emulates only the CPU, so it is lightweight and flexible. It can emulate raw binary code without an execution environment, but QEMU requires an entire system image or an executable binary. Third, QEMU cannot present dynamic instrumentation, whereas Unicorn supports customized handlers for several types of CPU events. This support renders it possible for an expert to create an instrumentation tool for emulation. Finally, QEMU has had many vulnerabilities, according to the CVE website [30], and all of them are from subsystems such as peripherals and input/output devices. However, Unicorn does not have any revealed vulnerabilities to now. Therefore, it is more secure than QEMU.

Simics [79, 132] is a commercial full system simulator that simulates complex digital systems' hardware and software. As Simics provides instruction-level fidelity and hardware models [135], it can support more than 10 processors, including ARM, MIPS (32/64bit), Alpha, x86-64, IA-64, MSP430, PowerPC (32/64bit), SPARC-V8, and V9. It also supports common operating systems (i.e., Windows, Linux, Solaris, and so on) as well as real-time operating systems (i.e., VxWorks and QNX). As Simics is a fast enough commercial simulator with sufficient fidelity and accuracy [105], it has been widely used in various industries. On the other hand, QEMU is an open-source machine

Table 8. Comparison of Network Fuzzers for Embedded Systems

	Prospect	FirmFuzz	IoTFuzzer
Preprocessing	No	Static analysis	Dynamic analysis
Taxonomy	Black-box	Gray-box	Black-box
Performance	High	Medium	Low
Output	Not good	Good	Good

emulator and is widely used in academia. Most emulation-based fuzzers in Table 4 use QEMU for their emulation platform.

In addition to the emulators mentioned above, other emulators exist, such as libemu [33], PyEmu [99], libCPU [44], IDA-x86emu [58], and Ghidra [95]. However, these emulators have limitations: for example, libemu only emulates x86 and is used to detect a shellcode. PyEmu is also an x86 emulator in Python, and IDA-x86emu is the x86 emulator plugin for IDA Pro [107]. These three are not multi-architecture emulators, but only x86 emulators. libCPU is an open-source library that emulates several CPU architectures, but it is incomplete and has not been updated for a long time. Ghidra is an open-source reverse engineering tool released by the **National Security Agency (NSA)**. It includes a suite of software analysis tools on various platforms, including Windows, Mac OS, and Linux, and supports a wide variety of processor instruction sets and executable formats. However, as the time that the Ghidra project was released is not long, only a few embedded system fuzzers [65] use the Ghidra emulator to date. In summary, other emulators, except the four (i.e., QEMU, Unicorn, Simics, and Ghidra), are difficult to use and are no longer maintained.

5.4 Fuzzing

Many publications and surveys [74, 76, 82] on traditional fuzzing have intensively described fuzzing algorithms and techniques. Meanwhile, most ESF publications have briefly described them as setting up a fuzzing environment, such as firmware acquisition or emulation, is more important. Therefore, we briefly discuss fuzzing techniques in this section, based on the fuzzing features described in ESF articles.

The first type of ESF is network fuzzing. As depicted in Table 4, Prospect, FirmFuzz, and IoTFuzzer include network fuzzers. Prospect [61] uses black-box fuzzing, which sets up a proxy to connect an emulation environment to an actual embedded system. The proxy captures network packets, and the inside fuzzer generates random traffic. Specifically, it takes packets from the captured network traffic, randomizes 1 byte at an arbitrary position, and replays the communication without a skip. This is a typical black-box fuzzing operation, but it is not smart. Accordingly, a smart fuzzer, FirmFuzz [115], was proposed. FirmFuzz adapts a generation-based gray-box fuzzer. It conducts static analysis and uses a command-line browser free from creating standard HTTP packets, which easily creates a generation-based fuzzer. IoTFuzzer [22] conducted a dynamic analysis to recognize the message of the IoT application and mutated the message to formulate test cases for the target device. This mutation is both a simple method without complex protocol analysis and a useful technique as it does not require consideration of encryption.

Table 8 presents a comparison of network fuzzers for embedded systems. Prospect uses simple black-box fuzzing and does not need preprocessing, so it is fast, but the result is unsatisfactory. Meanwhile, FirmFuzz leverages static analysis for smart fuzzing to obtain satisfactory results. Finally, IoTFuzzer requires dynamic IoT application analysis; thus, it is relatively slow, but it exhibits promising results. In summary, every network fuzzer has pros and cons depending on its policy and purpose.

The second type of embedded fuzzing is symbolic execution [9]. Symbolic execution regards a variable as a symbol. When it meets a path constraint, the symbol can be any value that satisfies the constraint. Consequently, theoretically, symbolic execution explores all paths in a program. For example, FIE [32] uses the KLEE [19] symbolic execution engine to present an extensible inspection of firmware programs. However, symbolic execution has the disadvantage of *state explosion*, which produces numerous program paths that it should explore. FIE addresses this problem through state pruning and memory smudging [32]. Several solutions for the state explosion problem have been proposed to date, but they are beyond the scope of this article.

Hybrid fuzzing combines fuzzing and concolic (or symbolic) execution to achieve broader and deeper program testing coverage [56, 81, 97, 117, 137, 144]. Thus far, no hybrid fuzzer is available in the ESF domain. However, several hybrid fuzzers in traditional fuzzing have shown promising results. For instance, QSYM [137] implemented a fast concolic execution engine that leveraged the native execution with symbolic emulation to overcome the bottlenecks of the concolic executors. As hybrid fuzzers suffer from finding vulnerabilities in real-world applications, QSYM proposed such a solution. Driller [117], another example, augmented fuzzing with selective concolic execution to find deeper bugs. It used fuzzing as an exerciser executing blocks of an application and concolic execution as an input generator that satisfies the path constraints. Consequently, Driller avoided path explosion and found the vulnerabilities successfully.

As coverage-based gray-box fuzzers [17, 139] have shown exemplary performance in traditional fuzzing, they can be adapted to ESF. The coverage-based fuzzer measures the code coverage by calculating the proportion of executed basic blocks and total basic blocks of the program, and then it uses this information to identify unvisited program blocks (i.e., widen the code coverage). The coverage-based fuzzer aims to test every path branch of the program (i.e., PUT). A typical example of these fuzzers is the Firm-AFL [145]. This fuzzer presents high throughput as its augmented process emulation renders it possible to test a target program in user-mode emulation quickly. However, as better coverage-based fuzzers exist in the traditional fuzzing area, we expect better coverage-based embedded fuzzers be proposed in the future.

While the feature of coverage-based fuzzers is to expand the code coverage, directed fuzzers [16, 21, 41, 126] spend most of their execution time on arriving at specific target points (e.g., bug-suspicious area). For example, FirmCorn [49] conducts directed fuzzing using a vulnerable code search algorithm. Owing to the effectiveness of the vulnerable code search algorithm, FirmCorn could achieve a very efficient time to crash. Directed fuzzing in traditional fuzzing research has two types: directed gray-box fuzzing and directed white-box fuzzing.

The authors of AFLGo [16] introduced directed gray-box fuzzing, which generates inputs guiding the fuzzer to the target points. They proposed a new power scheduler that assigns more power to test inputs that lead to the target points. In another example, Hawkeye [21] conducted a static analysis to gather information on the program and target locations and then executed the program along with the information. This strategy renders the Hawkeye fulfill fuzzing toward the target and shows better results.

BuzzFuzz [41] leveraged dynamic taint tracing to fuzz an instrumented program and then ran the program on the generated inputs to determine whether the inputs contained any bugs. This method enabled random fuzzing to explore a deep program code while preserving the semantic form of the input. This method is useful when a user identifies the program attack points, but it cannot detect unexpected bugs and takes a long time to test many points. In another example, TaintScope [126] used dynamic taint analysis and symbolic execution to bypass checksum mechanisms that block program execution from reaching the deep program code section. The execution monitor in TaintScope identified which input bytes control the arguments and which input bytes are related to the checksum. Then, it generated a bypass input. This method is helpful when a

program has checksum mechanisms, but this requires preprocessing overhead (i.e., dynamic taint tracing). Moreover, white-box fuzzing has a disadvantage in that it takes a long time to explore all possible paths.

5.5 Target Monitoring

As described in Section 3.3, crashes in the embedded system are difficult to detect. Therefore, several response-monitoring methods have been proposed. First, network heartbeat checking is widely used for network-based fuzzers to detect some embedded systems' hang or no effect. Second, crash detection methods in traditional fuzzing can be used in ESF, which are signal handlers or memory sanitizers [109, 116] in an emulator. For example, signals such as signal 11 (segmentation violation) or signal 10 (SIGUSR) are issued when a fuzzer finds crashes. The monitoring module in a fuzzer catches and handles this signal. Third, some embedded devices reboot when they encounter memory corruption. This phenomenon commonly occurs in Type 3 devices, but the fuzzer does not handle it automatically.

Table 4 shows how related pieces of research monitor the crash. First, network-based fuzzers, such as IoTFuzzer [22], check the liveness of a target device. It guesses program liveness by sending an arbitrary live check message. Muench et al. [87] also used a liveness check. Notably, they analyzed a target device's response behaviors thoroughly and classified the responses into six types: observable crash, reboot, hang, late crash, malfunctioning, and no effect. FirmFuzz [115] detects vulnerabilities by monitoring the logs generated by emulating firmware. This technique is advantageous when it emulates the target firmware. Further, it can detect other vulnerabilities such as command injection [118], null pointer dereference [37], and cross-site scripting [127] as these vulnerabilities can be detected by execution logs. Another detection method of memory corruption is conducted by a signal handler [61, 145]. They monitored the program execution in an emulator and detected signals such as segmentation fault. This method has been used widely in traditional fuzzing and is better used with **AddressSanitizer (ASAN)** [109]. ASAN can also help detect a memory error by using compiler instrumentation and runtime libraries.

5.6 Exception Analysis

After a fuzzer detects crashes or bugs of the PUT, it then confirms whether the discovered bugs are vulnerable. This step is performed manually by an analyst or automatically by program tools included in the fuzzer [121]. If a fuzzer finds a lot of crashes, a fuzzer or an analyst checks data duplication. Thus, an analyst finds out unique vulnerabilities. Note that an analyst uses a GNU debugger and a gdbserver when debugging embedded system applications [141]. This has the advantage of reducing the performance overhead by performing real debugging in a host system remote from the target system. In addition, most fuzzers use the result of the exception analysis as feedback to the next fuzz runs. For example, coverage-based fuzzers [145] use code coverage information to select the next seed. Meanwhile, taint-based directed fuzzers [98, 126] use the dynamic taint result or distance information from the program entry point to the targets [16]. All this information is included in *fuzzinfos* in Algorithm 1.

5.7 Seed Scheduling

After analyzing the exceptions, fuzzers generate seeds and select the next seed. Coverage-based fuzzers generate new seeds to expand the code coverage, and directed fuzzers generate new seeds to keep closer to the target points. For instance, Firm-AFL [145] uses lightweight instrumentation to widen the branch coverage, and this instrumentation includes coarse branch-taken hit counts in calculating the branch coverage [139]. Another fuzzer, FirmCorn [49], calculates vulnerability feature ranking of the API functions in the static analysis stage, hooks the suspicious functions,

and mutates the functions' inputs in order to widen the code coverage. In particular, FirmCorn uses heuristic-based mutations such as the bit-flip operation. Meanwhile, AFLGo [16] (i.e., a directed gray-box fuzzer) instruments the distance between the chosen seed and the set of target points. Then, it calculates the shortest path to the target points in the **control flow graph (CFG)** and executes the program along the path with a triggering seed. This triggering seed guides the fuzzer to a specific program point.

6 PERFORMANCE COMPARISONS

In this section, we discuss performance comparisons. Due to the variety of embedded systems, operating environments, and execution results, it is impossible to compare every device with one criterion. Instead, we should compare embedded fuzzers based on three groups according to our taxonomy; direct fuzzing, emulation-based fuzzing, and firmware analysis. However, as there are few fuzzers using firmware analysis, and their operating environments are very different, we discuss performance comparisons of two groups: direct fuzzing and emulation-based fuzzing.

We depict a performance comparison of network-based embedded fuzzers in Table 9, whose data is derived from the SNIPUZZ [39] paper. BooFuzz [98] is the basic network fuzzer using the protocol-based mutation, which requires manual input of protocol specification. This is inconvenient and shows bad experimental results in Table 9. Doona [134] is also a protocol-based fuzzer that does not analyze network packets but uses a pre-defined protocol format. Boofuzz mutates input messages starting from the protocol specification, but Doona uses a pre-defined overflow string. Doona also shows bad results in Table 9. **Network message syntax analysis (NEMESYS)** [68] leverages the internal message structure, which is the bitwise similarity of bytes in two consecutive bytes of the message. It generates fuzzed messages by the deterministic mutation within the internal structure of messages. This tool requires several resources, such as a network traffic monitor and a protocol analyzer, to show a good coverage result. IoTFuzzer [22] analyzes the protocol used in the communication between IoT applications and physical devices and performs protocol-guided fuzzing. This fuzzer shows good performance and results. SNIPUZZ [39] is a black-box fuzzer based on IoT devices' responses. It shows that the response message snippet reflects executed code blocks of the firmware. Therefore, its mutation is determined by the response message snippet, and it shows that the code coverage widens.

Table 10 shows a performance comparison of emulation-based embedded fuzzers. We performed 24 hours of fuzzing for eight network-related IoT devices and summarized the average values in Table 10. We used a desktop environment with an Intel i5 processor with 32-GB RAM, and the operating system was Ubuntu 16.04 LTS. The version of QEMU was 2.1.0, and the AFL version was 2.52b. Firm-AFL [145] proposes augmented process emulation, which combines full-system and user-mode emulation. It shows basic performance; it detected 52 crashes and found 492 paths. We implemented Firm-AFLFast by replacing AFL fuzzer [139] with AFLFast fuzzer [17]. Firm-AFLFast shows better performance than Firm-AFL because it widens the code coverage by using the power schedule that gravitates toward low-visited paths. We write the number of zero-day vulnerabilities according to each corresponding paper, but there is no paper of Firm-AFLFast. Therefore, it is not available in Table 10. FIRMCORN [49] uses optimized virtual execution and heuristic algorithms so that it shows good code coverage and crash detection results. FIRM-COV [64] uses firmware pre-analysis and optimized process emulation to offer better code coverage and crash detection.

7 TOOLS IN TERMS OF APPLICATION

In this section, we discuss the applicability of the embedded system fuzzers classified by their targets. Using this discussion, practitioners can determine how they use an embedded system fuzzer.

Table 9. Performance Comparison of Network-based Embedded Fuzzers

Name	Number of Crashes (24 hours)	Number of Vulnerabilities	Coverage (24 hours)	Required Resources	Features
NEMESYS [68]	0	0	51.35	◯, ✓, /, ✕	Deterministic mutation
BooFuzz [98]	0	0	28.85	◯, ✓	Protocol-based mutation
Doona [134]	0	0	14.5	◯, ✓	Protocol-based brute force
IoTFuzzer [22]	2	2	26.75	◯, ✓, /	API hooking, Mutation
SNIPUZZ [39]	13	5	59.4	◯, ✓, /	Determination, Snippet mutation

The meaning of the symbols in table is as follows:
(1) Number of crashes (24 hours)
(2) Number of zero-day vulnerabilities
· These numbers are based on each corresponding paper.
(3) Coverage: Average number of response categories (24 hours)
(4) Required resources
◯: IoT devices
✓: Fuzzing system
/: Network traffic monitor
✕: Protocol analyzer
(5) Features with overhead
· Deterministic mutation: It mutates input messages within the internal structure of messages.
· Protocol-based mutation: It mutates input messages based on communication protocol.
· Brute force: It uses brute-force exploit detector tool (BED).
· API hooking: It uses function hooking of mobile applications.
· Determination: It mutates input messages based on snippets determined by responses.
· Snippet mutation: It mutates message snippets rather than a single byte in a message.

They can develop a fuzzer for a framework, firmware, file system programs, and server applications. We classify embedded system fuzzers into four types by the target software level.

7.1 Fuzzing Framework

Over the past decade, although many embedded system fuzzers have been proposed, most of them are tightly coupled with their analysis system and are difficult to integrate with other systems [86]. This coupling renders it difficult for analysts to develop various tools to create a more potent fuzzing tool. Thus, proving a useful fuzzing framework in embedded systems is important as various embedded systems exist. To this end, Avatar [138] and Avatar2 [86] proposed a multi-target orchestration framework that combines dynamic analysis systems such as emulators or debuggers with real devices. This framework is beneficial when an analyst makes a debugging or testing system for an embedded system. However, it is not fully automated and requires some work that connects the real device to the framework.

7.2 Fuzzers for Firmware

Although many fuzzing systems receive the entire firmware as an input (i.e., *interfaces* column in Tables 4 and 5 has firmware), only a few systems [70, 86, 138] test parts such as the bootloader or operating systems outside the file system programs. For example, Avatar [138] analyzes the masked ROM bootloader and the bootloader of a hard disk drive. Avatar2 [86] can record the execution of firmware. These two examples are possible as they use partial emulation that executes the firmware in an emulator with physical devices. Consequently, this fuzzing framework is advantageous when an attacker is interested in the booting step or hardware hacking.

Table 10. Performance Comparison of Emulation-based Embedded Fuzzers

Name	Number of Crashes (24 hours)	Number of Vulnerabilities	Coverage (24 hours)	Emulation	Overhead
Firm-AFL [145]	52	2	492	Augmented process emulation	System call redirection
Firm-AFLFast [64]	133	N/A	1118	Augmented process emulation	System call redirection
FIRMCORN [49]	105	2	870	Optimized virtual execution	Heuristic algorithms
FIRM-COV [64]	335	2	2321	Optimized process emulation	Firmware pre-analysis

The meaning of the symbols in table is as follows:
(1) Number of crashes (24 hours)
(2) Number of zero-day vulnerabilities
· These numbers are based on each corresponding paper.
(3) Coverage: Average number of found paths (24 hours)
(4) Emulation method (required resources)
· Augmented process emulation: It combines full-system emulation and user-mode emulation.
· Optimized process emulation: It adds heuristic optimization to the augmented process emulation.
· Optimized virtual execution: It uses heuristic algorithms to optimize the virtual execution process.
(5) Overhead
· Firm-AFL and Firm-AFLFast require system call redirection. When user-mode emulation cannot handle system calls, it hands over the request to the full-system emulation.
· FIRMCORN requires heuristic algorithms during optimized virtual execution.
· FIRM-COV needs firmware pre-analysis because of optimized process emulation.

7.3 Fuzzers for File System Programs

Many embedded fuzzers can test every program in the file system of an embedded device. In Tables 4 and 5, many fuzzers [20, 28, 32, 112, 145] can analyze executable programs in the firmware. As they extract the firmware from the device and execute it in the emulator, they can access every program in the file system. If a fuzzer can access some programs, the file fuzzer that receives the program files can begin executing the programs. However, an attacker is especially interested in server programs that present network services to the Internet as they are accessible from the outside of the network (i.e., the attacker). Hence, although a fuzzer can test every program in a system, fuzzing articles typically indicate that vulnerabilities are mostly found in the server programs. Although attackers are not interested in vulnerabilities except server programs, security experts need to address these vulnerabilities as they can be abused by other attacks, such as local privilege escalation.

7.4 Fuzzers for Server Programs

Note that access methods to fuzz server programs (i.e., daemons) have three types. First, some fuzzers extract and run the file system from the firmware in an emulator and then fuzz the server programs. This method has the advantage of high performance and satisfactory results; thus, many fuzzers presented in Tables 4 and 5 use this method. Second, network protocol-based fuzzers [22, 43, 61, 115] are the most suitable type for fuzzing server programs. This method is intuitive and convenient but has the disadvantage that target programs are limited. Considering that a hacker typically attacks from the Internet, this scope is enough. Finally, some fuzzers [70, 87] using debugging interfaces such as JTAG or UART can test server programs. This method has the advantage in that it can fuzz server programs with debugging information. However, this method requires manual work that creates a debugging environment or is often impossible in

recent trends. In summary, network-based fuzzing is intuitive and convenient, whereas extraction and fuzzing provide the advantage of high throughput and plenty of outputs.

7.5 Fuzzers for Industrial Embedded Devices

Industrial embedded devices such as IIoT are usually controlled by **supervisory control and data acquisition (SCADA)** systems [18]. The SCADA system uses mostly a telephone network or third-party networks, which provide low-speed and poor-quality communication [40], and it requires an operator console or **human-machine interface (HMI)** to monitor and control the SCADA devices [128]. However, SCADA-based IIoT is increasingly connected to TCP/IP networks in order to provide convenience. This allows attackers to compromise the software vulnerabilities of SCADA systems from the Internet.

Security flaws of industrial embedded devices are more often found than general embedded systems because they use special software (i.e., SCADA software) and are rarely tested for security. Thus, fuzzing for industrial embedded devices is inevitable. Even though some fuzzers [63, 78, 125, 143] or countermeasures [36, 108] have been proposed so far, industrial embedded devices are numerous and varied. Thus, more interest and study are required in this research area. Additionally, note that fuzzing must not be applied to the SCADA system in operation because this can cause malfunction of the system. Before testing, an analyst should set up a test-bed and conduct fuzzing to industrial embedded systems.

8 FUTURE RESEARCH DIRECTIONS

Thus far, we have investigated state-of-the-art ESF. Based on the investigated contents, we answer RQ4 by discussing research trends and challenges in the fuzzing technique for embedded systems. We will discuss future challenges in the following subsections, hoping that the following would help motivate other researchers and developers.

8.1 Custom Embedded System Fuzzer

A vast number of IoT devices have emerged nowadays, and they have become increasingly diverse. Keeping pace with this phenomenon, many embedded system fuzzers will be available also. Some fuzzers are dedicated to only one platform architecture. For example, FIE [32] is a symbolic execution tool that analyzes firmware programs of MSP430-family microcontrollers. When this tool was published, these microcontrollers were very popular and dominant. Another fuzzer by Mulliner et al. [88] tests feature phones through a **global system for mobile communication (GSM)** network. This fuzzing system is customized only for the GSM network. In summary, all these fuzzers work well only with a limited architecture or platforms.

We believe that this trend will increase due to diversified IoT devices. Thus, there will be an increasing number of custom embedded system fuzzers. For instance, real-time embedded devices using VxWorks are predominant in weapon systems, critical infrastructures, medical devices, and so on, as VxWorks is an RTOS designed for an environment that requires real-time, availability, safety, and security certification. Although VxWorks is widely used in critical industries, to the best of our knowledge, a useful fuzzer for it does not exist to date. Therefore, we predict that many custom embedded system fuzzers, such as a fuzzer for VxWorks or QNX, are necessary.

8.2 Higher Code Coverage

Although state-of-the-art embedded coverage-based fuzzers such as Firm-AFL [145] show good performance and results, there is room for improvement. For example, Firm-AFL generates new test cases by using a basic mutation algorithm, in which a new test case is inserted into a seed pool only when the new test input finds a new path, or it is ignored. According to AFLFast [17],

this **coverage-based gray-box fuzzer (CGF)** shows low-frequency paths and needs to explore considerably more paths. In addition, recently improved coverage-based fuzzing techniques [8, 23, 102] can be adapted to this framework.

In traditional fuzzing research, several optimization techniques have been proposed to improve the code coverage. First, AFLFast [17] adapts a power scheduler and search strategy that guides the fuzzer to less visited program paths. The power scheduler allocates high energy to the seeds exploring low-frequency paths and low energy to the seeds exploring high-frequency paths. This implies that AFLfast focuses on less visited paths. The search strategy of AFLFast involves choosing seeds in the queue. It first chooses the seeds that produce low-frequency paths and have been chosen less frequently. Second, VUzzer [102] conducted static and dynamic analyses to extract control-flow and data-flow features. Using this information, it accomplishes application-aware evolutionary fuzzing. This fuzzer is more lightweight than symbolic execution and thus exhibits better results.

8.3 Toward Perfect Emulation

As described in Section 5.3, emulation is an essential process for testing a variety of embedded systems. As all types of IoT devices cannot be developed in an experimental environment, emulation is the best solution. Thus, perfect emulation renders the fuzzer test target devices correctly. Several fuzzers [86, 138] exercise a partial emulation that combines emulation with physical devices to prepare perfect emulation. Meanwhile, many fuzzers [20, 49, 61, 70, 115, 146] exercise full system emulation. Augmented emulation [145] and approximate emulation [38] have recently been reported.

However, emulation requires improvement to date. QEMU, which is frequently used in firmware emulation, does not support NVRAM emulation and floating-point instruction, although most embedded devices have NVRAM nowadays. Even worse, sometimes it produces incorrect results compared with real devices [49]. In addition, there is huge time overhead and unstable fuzzing due to incomplete emulation resulting from the above facts. All these issues are open problems and deserve to be studied in depth in the future.

8.4 Hybrid Fuzzer for Embedded Systems

The state-of-the-art fuzzers have two limitations: (1) a fuzzer cannot test every path branch, and (2) it cannot generate every possible test case. Owing to these limitations, other techniques tend to be combined with fuzzing. For example, several previous works [28, 49, 104, 146] applied static and taint analyses before fuzz testing. Through this preparation process, they proposed unique methodologies and produced better experimental results. We believe that this trend will continue as the popularity of IoT devices works up various improved techniques. In particular, a hybrid fuzzer for the embedded system is not available to date. Thus, designing and implementing a more effective and efficient fuzzing technique is an open problem in this research area.

8.5 ESF with AI Techniques

AI technology has become popular to solve state-of-the-art security and engineering problems. For example, in the ESF field, Pretender [50] used a machine learning technique to re-host embedded systems' firmware. It was trained from interaction data between the CPU and peripherals and provided virtual peripheral models to re-host firmware. Based on these models, firmware samples are flexibly executed in an emulated environment, and post-emulation analyses such as fuzzing are performed well. These models are useful as they provide interactive and stable emulation environments without actual devices. However, they require enough and well-organized training data for a successful emulation.

Other desktop fuzzing methods also use AI techniques. For example, Learn&Fuzz [47] proposed a fuzzing input generation technique based on machine learning. This technique learned an input format by processing many sample inputs and automatically generated a suitable fuzzing input. They insisted that their fuzzer achieved better code coverage and found an unknown bug. Another example, MEUZZ [24], proposed a smart seed scheduling algorithm for hybrid fuzzing. This method measured which seeds produced better code coverage based on a large amount of previous data, achieved 27.1% more code coverage, and discovered unknown vulnerabilities.

These examples also can be applied to ESF. For instance, the new ESF learns an input format by processing many sample inputs and automatically generates an effective fuzzing input. Another possible example measures the code coverage of many seed inputs and learns a smart seed scheduling algorithm from them. Or emulated peripherals [147] know typical execution outputs and error outputs and imitate normal peripheral behaviors. Hence, applying AI techniques to ESF is an open challenge.

9 CONCLUSION

ESF is an automatic testing technique for embedded system firmware. It is more complicated than traditional fuzzing as its target is heterogeneous embedded devices with a strong dependency on hardware, difficulty in detecting crashes, and performance limitation. Thus, these limitations should be considered when using and developing ESF. ESF can be classified into direct fuzzing, emulation-based fuzzing, and firmware analysis depending on how it accesses the target program. Compared with traditional fuzzing, it requires additional preprocessing steps, such as firmware extraction, system emulation, or device connection. Along with the comparison, we presented features and taxonomy of state-of-the-art embedded system fuzzers. In particular, we noted that most fuzzers for embedded systems were emulation based; thus, we emphasized several emulation types. We described fuzzing techniques and their usability in terms of testing embedded systems and discussed open challenges.

Considering recent trends in the ESF field, we expect that much related research will be studied. Thus, this survey provides information and guidelines for practitioners who select an embedded system fuzzer and researchers who study ESF. We provided key techniques and usability for practitioners and open challenges for researchers. We believe that future works are developing customized fuzzers for various embedded systems, improving fuzzing techniques, rendering emulation more perfect, and adapting AI technology to ESF. We hope that our work will motivate researchers to study ESF for a secure IoT world.

APPENDIX

A ACRONYMS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
DSE	Dynamic Symbolic Execution
ESF	Embeded System Fuzzing
IoT	Internet of Things
IIoT	Industrial Internet of Things
JTAG	Joint Test Action Group
NVRAM	Non-Volatile Random Access Memory
OS	Operating System
PUT	Program Under Test

RISC	Reduced Instruction Set Computing
RTOS	Real-Time Operating System
SCADA	Supervisory Control And Data Acquisition
SW	Software
UART	Universal Asynchronous Receiver Transmitter

ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. 2021. Android $SmartTVs$ Vulnerability discovery via $Log-Guided$ fuzzing. In *30th USENIX Security Symposium (USENIX Security'21)*. 2759–2776.

[2] Amazon. [n.d.]. Amazon Prime Air. Retrieved February 1, 2022, from https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011.

[3] Amazon. [n.d.]. Google X-wing. Retrieved February 1, 2022, from http:https://x.company/projects/wing/.

[4] Muhammad Amjad, Muhammad Sharif, Muhammad Khalil Afzal, and Sung Won Kim. 2016. TinyOS-new trends, comparative views, and supported sensing applications: A review. *IEEE Sensors Journal* 16, 9 (2016), 2865–2889.

[5] Erik Andersen. [n.d.]. uClibc. Retrieved June 10, 2020, from https://www.uclibc.org/.

[6] Arduino. [n.d.]. arduino. Retrieved June 10, 2020, from https://www.arduino.cc/.

[7] Sandra Pérez Arteaga, Luis Alberto Martínez Hernández, Gabriel Sánchez Pérez, Ana Lucila Sandoval Orozco, and Luis Javier García Villalba. 2019. Analysis of the GPS spoofing vulnerability in the drone 3DR solo. *IEEE Access* 7 (2019), 51782–51789.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In *Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[10] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[11] Arnold Berger. 2001. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CRC Press.

[12] beSTORM. [n.d.]. Dynamic, Black Box Testing on the High Speed Universal Asynchronous Receiver/Transmitter (High Speed UART). Retrieved April 5, 2020, from https://beyondsecurity.com/dynamic-fuzzing-testing-high-speed-universal-asynchronous-receiver-transmitter-uart.html?cn-reloaded=1.

[13] Dileep Bhandarkar and Douglas W. Clark. 1991. Performance from architecture: Comparing a RISC and a CISC with similar hardware organization. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. 310–319.

[14] Mohammad Shameel bin Mohammad Fadilah, Vivek Balachandran, Peter Loh, and Melissa Chua. 2020. DRAT: A drone attack tool for vulnerability assessment. In *Proceedings of the 10th ACM Conference on Data and Application Security and Privacy*. 153–155.

[15] BlackBerry. [n.d.]. QNX. Retrieved June 10, 2020, from https://blackberry.qnx.com/en/software-solutions/embedded-software/qnx-neutrino-rtos.

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as Markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[18] Hugh Boyes, Bil Hallaq, Joe Cunningham, and Tim Watson. 2018. The industrial internet of things (IIoT): An analysis framework. *Computers in Industry* 101 (2018), 1–12.

[19] Cristian Cadar, Daniel Dunbar, Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX conference on Operating Systems Design and Implementation (OSDI)*, Vol. 8. 209–224.

[20] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, Vol. 16. 1–16.

[21] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[22] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *NDSS*.

[23] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 711–725.

[24] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'20)*. 77–92.

[25] P. C. Ching, Y. H. Cheng, and M. H. Ko. 1994. An in-circuit emulator for TMS320C25. *IEEE Transactions on Education* 37, 1 (1994), 51–56.

[26] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–49.

[27] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Sec'20)*. 1–18.

[28] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security'14)*. 95–110.

[29] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.

[30] CVE. [n.d.]. Common Vulnerabilities and Exposures. Retrieved March 1, 2020, from https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=QEMU.

[31] The New Jersey Cybersecurity and Communications Integration Cell (NJCCIC). [n.d.]. Mirai Botnet. Retrieved January 17, 2022, from https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet.

[32] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Presented as Part of the 22nd USENIX Security Symposium (USENIX Security'13)*. 463–478.

[33] Angelo Dell'Aera. [n.d.]. libemu. Retrieved June 10, 2020, from https://github.com/buffer/libemu.

[34] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 1–11.

[35] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 110–121.

[36] Gregory Falco, Carlos Caldera, and Howard Shrobe. 2018. IIoT cybersecurity risk modeling for SCADA systems. *IEEE Internet of Things Journal* 5, 6 (2018), 4486–4495.

[37] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[38] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*.

[39] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 337–350.

[40] Brendan Galloway and Gerhard P. Hancke. 2012. Introduction to industrial control networks. *IEEE Communications Surveys & Tutorials* 15, 2 (2012), 860–880.

[41] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 474–484.

[42] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. 2020. EM-Fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3420–3432.

[43] Zicong Gao, Weiyu Dong, Rui Chang, and Yisen Wang. 2020. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. *Concurrency and Computation: Practice and Experience* 34, 16 (2020), e5756.

[44] Github. [n.d.]. libcpu. Retrieved June 10, 2020, from https://github.com/libcpu/libcpu.

[45] Patrice Godefroid. 2007. Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing: Co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 1–1.

[46] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20–27.
[47] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, 50–59.
[48] Anjana Gosain and Ganga Sharma. 2015. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA'14)*, Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J. K. Mandal (Eds.). Springer International Publishing, Cham, 113–122.
[49] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. 2020. FIRMCORN: Vulnerability-oriented fuzzing of IoT firmware via optimized virtual execution. *IEEE Access* 8 (2020), 29826–29841.
[50] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. 135–150.
[51] Prasanna Hambarde, Rachit Varma, and Shivani Jha. 2014. The survey of real time operating system: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. IEEE, 34–39.
[52] Steve Heath. 2002. *Embedded Systems Design*. Elsevier.
[53] Thomas A. Henzinger and Joseph Sifakis. 2007. The discipline of embedded systems design. *Computer* 40, 10 (2007), 32–40.
[54] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin R. B. Butler. 2017. Firmusb: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2245–2262.
[55] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. 2022. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In *NDSS*, Vol. 22. 1–19.
[56] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. PANGOLIN: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP'20)*. IEEE, 1613–1627.
[57] IBM. [n.d.]. Source Code Instrumentation Overview. Retrieved March 7, 2020, from https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html.
[58] IDA. [n.d.]. IDA-x86emu. Retrieved June 10, 2020, from http://www.idabook.com/x86emu/index.html.
[59] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security'21)*. 321–338.
[60] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE'16)*.
[61] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: Peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. 329–340.
[62] Chung-Fu Kao, Hsin-Ming Chen, and Jer Huang. 2008. Hardware-software approaches to in-circuit emulation for embedded processors. *IEEE Design & Test of Computers* 25, 5 (2008), 462–477.
[63] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-guided fuzzing for robotic vehicles. In *Network and Distributed System Security Symposium*.
[64] Juhwan Kim, Jihyeon Yu, Hyunwook Kim, Fayozbek Rustamov, and Joobeom Yun. 2021. FIRM-COV: High-coverage greybox fuzzing for IoT firmware via optimized process emulation. *IEEE Access* 9 (2021), 101627–101642.
[65] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Annual Computer Security Applications Conference*. 733–745.
[66] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave Jing Tian. 2021. PASAN: Detecting peripheral access concurrency bugs within Bare-Metal embedded applications. In *30th USENIX Security Symposium (USENIX Security'21)*. 249–266.
[67] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.
[68] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *12th USENIX Workshop on Offensive Technologies (WOOT'18)*.

[69] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 447–462.
[70] Karl Koscher, Tadayoshi Kohno, and David Molnar. 2015. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT'15)*.
[71] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. 2010. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*. IEEE, 59–64.
[72] Edward Ashford Lee and Sanjit A. Seshia. 2017. *Introduction to Embedded Systems: A Cyber-physical Systems Approach*. MIT Press.
[73] Michael Ley. [n.d.]. The dblp Computer Science Bibliography. Retrieved April 5, 2020, from https://dblp.org/faq/What+is+dblp/.
[74] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A survey. *Cybersecurity* 1, 1 (2018), 1–13.
[75] Chen Liang, Meixia Miao, Jianfeng Ma, Hongyan Yan, Qun Zhang, Xinghua Li, and Teng Li. 2019. Detection of GPS spoofing attack on unmanned aerial vehicle system. In *International Conference on Machine Learning for Cyber Security*. Springer, 123–139.
[76] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (Sept. 2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476
[77] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices* 40, 6 (2005), 190–200.
[78] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function code aware fuzz testing of ICS protocol. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.
[79] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.
[80] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT'19)*.
[81] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.
[82] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[83] MDSec. [n.d.]. An Arduino UARTFuzzer. Retrieved April 5, 2020, from https://github.com/mdsecresearch/UARTFuzz.
[84] Charles Melear. 1997. Emulation techniques for microcontrollers. In *Wescon/97 Conference Proceedings*. IEEE, 532–541.
[85] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279
[86] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.'18)*, Vol. 18. 1–11.
[87] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*.
[88] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of death: From analyzing to attacking mobile phones on a large scale. In *USENIX Security Symposium*, Vol. 168.
[89] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices* 42, 6 (2007), 89–100.
[90] James Newsome and Dawn Xiaodong Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, Vol. 5. Citeseer, 1–17.
[91] Anh Quynh Nguyen and Hoang Vu Dang. 2015. Unicorn: Next generation CPU emulator framework. In *Proceedings of the 2015 Blackhat USA Conference*.
[92] Srinivas Nidhra and Jagruthi Dondeti. 2012. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* 2, 2 (2012), 29–50.
[93] Amy Nordrum. [n.d.]. Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. Retrieved December 10, 2021, from https://spectrum.ieee.org/techtalk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated.
[94] J. Norhuzaimin and H. H. Maimun. 2005. The design of high speed UART. In *2005 Asia-Pacific Conference on Applied Electromagnetics*. IEEE, 5–pp.
[95] NSA. [n.d.]. Ghidra. Retrieved August 10, 2020, from https://ghidra-sre.org/.

[96] P. Oehlert. 2005. Violating assumptions with fuzzing. *IEEE Security Privacy* 3, 2 (March 2005), 58–62. https://doi.org/10.1109/MSP.2005.55
[97] Brian S. Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University* (2012).
[98] J. Pereyda. [n.d.]. boofuzz. Retrieved March 1, 2020, from https://github.com/jtpereyda/boofuzz.
[99] Cody Pierce. [n.d.]. PyEmu. Retrieved June 10, 2020, from https://github.com/codypierce/pyemu.
[100] GNU project. [n.d.]. GNU Binutils. Retrieved August 10, 2020, from https://www.gnu.org/software/binutils/.
[101] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. 2004. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 3 (2004), 461–491.
[102] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *NDSS*, Vol. 17. 1–14.
[103] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. DIANE: Identifying fuzzing triggers in Apps to generate under-constrained inputs for IoT devices. In *2021 IEEE Symposium on Security and Privacy (SP'21)*. IEEE, 484–500.
[104] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2015. KARONTE: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP'15)*. 1544–1561.
[105] Wind River. [n.d.]. Wind River Simics. Retrieved December 3, 2021, from https://resources.windriver.com/i/1183368-wind-river-simics-product-note/0?.
[106] Kurt Rosenfeld and Ramesh Karri. 2010. Attacks and defenses for JTAG. *IEEE Design & Test of Computers* 27, 1 (2010), 36–47.
[107] Hex-Rays SA. [n.d.]. IDA Pro. Retrieved June 10, 2020, from https://www.hex-rays.com/products/ida/index.shtml.
[108] Jayasree Sengupta, Sushmita Ruj, and Sipra Das Bit. 2020. A comprehensive survey on attacks, security issues and blockchain solutions for IoT and IIoT. *Journal of Network and Computer Applications* 149 (2020), 102481.
[109] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as Part of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 309–318.
[110] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71.
[111] Y. Shoshitaishvili. [n.d.]. angr. Retrieved June 10, 2020, from http://fuse.sourceforge.net.
[112] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*.
[113] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. 2015. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th USENIX Security Symposium (USENIX Security'15)*. 881–896.
[114] Jia Song and Jim Alves-Foss. 2015. The darpa cyber grand challenge: A competitor's perspective. *IEEE Security & Privacy* 13, 6 (2015), 72–76.
[115] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: Automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 15–21.
[116] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. IEEE, 46–55.
[117] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, Vol. 16. 1–16.
[118] Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in web applications. *ACM Sigplan Notices* 41, 1 (2006), 372–382.
[119] Ed Sutter. 2002. *Embedded Systems Firmware Demystified*. CMP Books.
[120] Trenton Systems. [n.d.]. What Are Embedded Systems? Retrieved December 10, 2021, from https://www.trentonsystems.com/blog/what-are-embedded-systems.
[121] Ari Takanen, Jared D. Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House.
[122] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern Operating Systems*. Pearson.
[123] TechRepublic. [n.d.]. IoT Device Attacks Double in the First Half of 2021, and Remote Work May Shoulder Some of the Blame. Retrieved January 17, 2022, from https://www.techrepublic.com/article/iot-device-attacks-double-in-the-first-half-of-2021-and-remote-work-may-shoulder-some-of-the-blame/.

[124] Sebastian Vasile, David Oswald, and Tom Chothia. 2018. Breaking all the things–A systematic survey of firmware extraction techniques for IoT devices. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 171–185.
[125] Artemios G. Voyiatzis, Konstantinos Katsigiannis, and Stavros Koubias. 2015. A modbus/TCP fuzzer for testing internetworked industrial systems. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA'15)*. IEEE, 1–6.
[126] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.
[127] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 171–180.
[128] Joseph Weiss. 2010. *Protecting Industrial Control Systems from Electronic Threats*. Momentum Press.
[129] Nicholas Wells. 2000. Busybox: A Swiss army knife for Linux. *Linux Journal* 2000, 78es (2000), 10–es.
[130] Wikipedia. [n.d.]. jtag. Retrieved June 10, 2020, from https://en.wikipedia.org/wiki/JTAG.
[131] Wikipedia. [n.d.]. uart. Retrieved June 10, 2020, from https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.
[132] Windriver. [n.d.]. simics. Retrieved June 10, 2020, from https://www.windriver.com/products/simics/.
[133] Windriver. [n.d.]. VxWorks. Retrieved June 10, 2020, from https://www.windriver.com/products/vxworks/.
[134] wirehoul. [n.d.]. Doona. Retrieved December 10, 2021, from https://github.com/wireghoul/doona.
[135] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. 2021. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–36.
[136] L. D. Xu, W. He, and S. Li. 2014. Internet of Things in industries: A survey. *IEEE Transactions on Industrial Informatics* 10, 4 (Nov. 2014), 2233–2243. https://doi.org/10.1109/TII.2014.2300753
[137] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security'18)*. 745–761.
[138] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, Vol. 14. 1–16.
[139] Michal Zalewski. [n.d.]. American Fuzzy Lop. Retrieved March 1, 2020, from http://http://lcamtuf.coredump.cx/afl/.
[140] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. 2018. All your GPS are belong to us: Towards stealthy manipulation of road navigation systems. In *27th USENIX Security Symposium (USENIX Security'18)*. 1527–1544.
[141] Yunjia Zhang, Bin Liu, and Qing Zhou. 2011. A dynamic software binary fault injection system for real-time embedded software. In *Proceedings of the 2011 9th International Conference on Reliability, Maintainability and Safety*. IEEE, 676–680.
[142] Zhenghao Zhang, Matthew Trinkle, Lijun Qian, and Husheng Li. 2012. Quickest detection of GPS spoofing attack. In *2012 IEEE Military Communications Conference (MILCOM'12)*. IEEE, 1–6.
[143] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST'19)*. IEEE, 59–67.
[144] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*.
[145] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.
[146] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. 2019. An efficient greybox fuzzing scheme for linux-based IoT programs through binary static analysis. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC'19)*. IEEE, 1–8.
[147] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In *USENIX Security Symposium*. 2007–2024.

Received 12 February 2021; revised 4 May 2022; accepted 16 May 2022