electronics

MDPI

Article

IoTFuzzBench: A Pragmatic Benchmarking Framework for Evaluating IoT Black-Box Protocol Fuzzers

Yixuan Cheng [1,2], Wenxin Chen [1,2], Wenqing Fan [1,2], Wei Huang [1,2], Gaoqing Yu [1,2] and Wen Liu [1,2,*]

[1] State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing 100024, China; yixuancheng@cuc.edu.cn (Y.C.)
[2] School of Computer and Cyber Sciences, Communication University of China, Beijing 100024, China
* Correspondence: lw8206@cuc.edu.cn

Abstract: High scalability and low operating cost make black-box protocol fuzzing a vital tool for discovering vulnerabilities in the firmware of IoT smart devices. However, it is still challenging to compare black-box protocol fuzzers due to the lack of unified benchmark firmware images, complete fuzzing mutation seeds, comprehensive performance metrics, and a standardized evaluation framework. In this paper, we design and implement IoTFuzzBench, a scalable, modular, metric-driven automation framework for evaluating black-box protocol fuzzers for IoT smart devices comprehensively and quantitatively. Specifically, IoTFuzzBench has so far included 14 real-world benchmark firmware images, 30 verified real-world benchmark vulnerabilities, complete fuzzing seeds for each vulnerability, 7 popular fuzzers, and 5 categories of complementary performance metrics. We deployed IoTFuzzBench and evaluated 7 popular black-box protocol fuzzers on all benchmark firmware images and benchmark vulnerabilities. The experimental results show that IoTFuzzBench can not only provide fast, reliable, and reproducible experiments, but also effectively evaluate the ability of each fuzzer to find vulnerabilities and the differential performance on different performance metrics. The fuzzers found a total of 13 vulnerabilities out of 30. None of these fuzzers can outperform the others on all metrics. This result demonstrates the importance of comprehensive metrics. We hope our findings ease the burden of fuzzing evaluation in IoT scenarios, advancing more pragmatic and reproducible fuzzer benchmarking efforts.

Keywords: fuzzing evaluation; black-box fuzzing; IoT smart device; benchmark suite

Citation: Cheng, Y.; Chen, W.; Fan, W.; Huang, W.; Yu, G.; Liu, W. IoTFuzzBench: A Pragmatic Benchmarking Framework for Evaluating IoT Black-Box Protocol Fuzzers. *Electronics* **2023**, *12*, 3010. https://doi.org/10.3390/electronics12143010

Academic Editor: Elif Bilge Kavun

Received: 19 June 2023
Revised: 5 July 2023
Accepted: 8 July 2023
Published: 9 July 2023

1. Introduction

The Internet of Things (IoT) smart devices have become a part of the daily life of billions of people and have brought us new lifestyles and convenience [1,2]. Recent statistics show that the number of IoT smart devices will reach 29.42 billion by 2030 [3]. However, IoT smart devices have been reported to be vulnerable to various attacks for a long time [4,5]. According to a report published by Palo Alto in 2020 [6], over 50% of IoT smart devices worldwide are vulnerable to medium-high severity attacks.

Fuzzing has become one of the most successful techniques for finding software security vulnerabilities [7]. Fuzzing generates many test cases, repeatedly tests the target software, and monitors the abnormal conditions of the program [7,8]. Security vulnerabilities in IoT smart devices usually exist in device firmware [9]. The firmware is code embedded in hardware which can provide hardware support for upper-level users [9,10]. Some components in the firmware of these devices usually have security vulnerabilities due to the implementation defects of protocol parse or improper handling of data submitted by users [11]. Since IoT smart devices are computing devices with networking capabilities, they cannot be immune to attacks from the internet as long as users can access them remotely through specific network protocols [12]. This prompted researchers to explore how to perform black-box fuzzing on firmware from outside the device via network protocols, successfully finding many vulnerabilities [11,13–15].

Although black-box protocol fuzzing has successfully improved the firmware quality of IoT smart devices, there are still many challenges to properly evaluating black-box protocol fuzzing techniques. The first challenge is the lack of unified benchmarks for IoT fuzzing evaluation. Fuzzing papers use different benchmark devices and firmware images, making comparing them hard. Moreover, some benchmark devices may be discontinued or updated, making reproducing the experiments hard. Therefore, follow-up research may not be able to obtain the same benchmark devices or firmware images as existing work. The second challenge is the high cost of fuzzing evaluation. It involves comparing fuzzing tools on many benchmark firmware images, which requires time and effort, sometimes months [16]. Setting up all these tools and the benchmark firmware images and ensuring each tool–baseline pair works together (e.g., firmware emulation) also requires much effort. Due to cost concerns, many fuzzing papers may not fully reproduce the existing experiments [17–20]. The third challenge is the lack of a unified seed message. An IoT smart device may have dozens of different network interfaces. Using a seed message from one interface to fuzz another interface may not be effective, as the quality of seeds influences the fuzzing results [4]. However, many fuzzing papers do not disclose the seed messages they use for different interfaces [11,13,14], making it hard to replicate their work. The fourth challenge is the lack of comprehensive metrics. Existing fuzzing papers mainly use the number and time of triggering crashes as evaluation metrics [13,14], but these are not enough. The feedback response message of the IoT smart device can reflect the code coverage of the fuzzer. More unique response messages indicate more paths explored by the fuzzer. In addition, the computational resource consumption of fuzzers is often ignored as an evaluation indicator [20], but it is important for selecting fuzzers under limited resources. All of these challenges hinder the reproducibility and comparability of black-box protocol fuzzers for IoT firmware.

Klees et al. were the first to study fuzzing evaluation [17]. They analyzed 32 fuzzing research papers and found that none provided enough evidence to justify general validity claims [16]. Following their work, FuzzBench [16], Magma [21], and UNIFUZZ [20] conducted further studies on fuzzing evaluation from different focus points. FuzzBench regards code coverage as an essential indicator for evaluating fuzzers, and fuzzers with higher code coverage are considered to perform better. Unlike FuzzBench, Magma tries to introduce actual bugs into genuine software and believes that the number of bugs is an important metric to measure the performance of a fuzzer. They argue that the correlation between the crashes found by coverage guidance and actual bugs is not strong, which means that higher coverage does not necessarily mean better fuzzer effectiveness. UNIFUZZ considers bug count and coverage metrics and suggests other metrics, such as the speed of finding bugs [16]. The benchmark programs for evaluating these fuzzers are software in a general IT environment, and the evaluated fuzzers are general fuzzers under coverage guidance. However, in the IoT scenario, the benchmark program should be the firmware images of the IoT smart devices, and the architectures of these are pretty different (such as x86, MIPS, and ARM) [8]. At the same time, these firmware images usually provide various network communication services [22]. Therefore, software in a general IT environment cannot effectively represent firmware. In addition, since device vendors usually do not provide firmware source code and documentation, the firmware images cannot be recompiled and instrumented [11], and the hardware debugging interface is usually disabled [2], so general-purpose fuzzers based on source code or coverage cannot be directly applied to IoT fuzzing. Therefore, the above fuzzing evaluation methods cannot be directly used to evaluate existing IoT black-box protocol fuzzers. There is an urgent need for a quantitative fuzzing evaluation method to comprehensively and pragmatically evaluate state-of-the-art black-box protocol fuzzers for IoT smart devices on a unified framework.

Our Approach. To address the above challenges, we designed and implemented IoTFuzzBench, a scalable, modular, metric-driven fuzzing evaluation framework for IoT black-box protocols. So far, IoTFuzzBench has included 14 real-world benchmark firmware

images, 30 real-world benchmark vulnerabilities, corresponding fuzzing seeds, 7 popular fuzzers, and 5 categories of performance metrics. We verified the emulation ability of each benchmark firmware image and provided a benchmark bundle to facilitate the automated construction of emulation environments for the firmware images. For each benchmark vulnerability in benchmark firmware images, we manually verified its existence and delivered the original communication message corresponding to the vulnerability as the seed message for fuzzing. We tested the usability of each fuzzer and provided a fuzzer bundle for easy deployment and testing. We also proposed a collection of performance metrics in 5 categories, which can be used to evaluate the performance of fuzzers comprehensively. We present the rationale behind IoTFuzzBench, the challenges encountered during its creation, implementation details, and some preliminary results that show the effectiveness of IoTFuzzBench.

Contributions. In summary, we make the following contributions:

- **New Benchmark Suite**: We constructed a set of practical IoT firmware benchmark suites. The benchmark suite includes 14 real-world benchmark firmware images and their operating environment, 30 verified real-world vulnerabilities that can be triggered in the benchmark firmware environment, and fuzzing seed messages corresponding to each benchmark vulnerability;
- **New Framework**: We described the design of IoTFuzzBench. To the best of our knowledge, this is the first framework to evaluate IoT black-box protocol fuzzers comprehensively and quantitatively. IoTFuzzBench includes the above benchmark suite, 7 popular fuzzers, and 5 categories of performance metrics and can automate the evaluation process of fuzzers in a configurable manner;
- **Implementation and Evaluation**: We implemented the prototype of IoTFuzzBench, conducted a proof-of-concept evaluation of the widely used fuzzers on the above benchmark suite, and presented the evaluation results. In addition, we open-sourced the prototype of IoTFuzzBench and the complete benchmark suite [23].

Roadmap. The remainder of this article is organized as follows. Section 2 reviews the background and related work of IoT smart device fuzzing and fuzzing evaluation. Section 3 details the architecture of IoTFuzzBench. In Section 4, the experiments and evaluation results are introduced. The limitations of the current design are discussed in Section 5. Finally, Section 6 concludes the article.

2. Background and Related Work

In this section, we briefly introduce the background of fuzzing in the IoT smart device scenario and fuzzing evaluation, an approach that has focused on comparing vulnerability discovery performance between different fuzzers.

2.1. IoT Smart Device Fuzzing

The main challenges of discovering vulnerabilities in IoT smart devices include the lack of computing resources of IoT devices, the diversity of device types and protocols, the difficulty in obtaining and emulating firmware, and the limited feedback from devices [7,8]. Fuzzing is a software vulnerability discovery method widely used in academic and industrial fields [22]. Fuzzing repeatedly tests the target program by generating many test cases and monitors the abnormal conditions of the program to trigger a fault [21]. Fuzzing is widely used in vulnerability discovery of IoT smart devices due to its characteristics of easy use, convenient deployment, and easy recurrence of crashes [12–14]. According to the prior knowledge of the program under test during execution, fuzzing methods can be divided into three categories: black-box, grey-box, and white-box fuzzing [22]. Black-box fuzzing does not know the internal state of the program for each execution [13,24]. The program under test is a black box to fuzzers, which usually optimize the fuzzing process by exploiting input formats or different program output states [25,26]. White-box fuzzing usually needs to have all the source code of the target object, to be able to obtain all the execution information of the target object during the fuzzing process [27]. Grey-box fuzzing acquires knowledge

of the execution state between black-box and white-box fuzzing. Grey-box fuzzers do not require access to the source code of the target program and typically use edge coverage as the internal execution state [28,29].

Since the first fuzzers were created, fuzzers have evolved significantly and have become one of the most effective methods for discovering software vulnerabilities [7,9]. However, some challenges are encountered when fuzzing technology is applied to the embedded environment of the IoT. On the one hand, compared with the general IT environment, the embedded environment of the IoT has more types of firmware architecture, more scarce computing resources, and less responsiveness of information systems [22]. On the other hand, firmware source code and documentation are usually unavailable [2,9]. Therefore, some out-of-the-box fuzzing methods that depend on the source code or detailed execution status of the target program cannot be directly applied to embedded systems [12,14].

Existing fuzzing methods for IoT smart devices mainly address the above challenges using grey-box fuzzing based on firmware emulation and black-box fuzzing based on the network [7]. Emulators can execute programs that initially ran on IoT firmware without corresponding hardware. Firmware emulation can not only provide researchers with more low-cost research objects, but also enrich the responses obtained by fuzzers. With the help of emulators, fuzzers can test target firmware images in a grey-box manner [30]. However, firmware cannot always be emulated successfully. The firmware emulation success rate of the state-of-the-art firmware emulator Firmadyne is only 16.28% [31]. In practice, since the firmware architecture and hardware dependencies of different manufacturers are very different, the emulation success rate is lower. Therefore, grey-box fuzzers that rely on the success of firmware emulation to obtain device internal code coverage are not suitable for all scenarios. This has motivated researchers to attempt fuzzing IoT smart devices over the network from outside the device [14,15,32]. Since IoT smart devices can communicate with the outside world through the network, the fuzzer automatically sends request messages to a device and waits for the execution response results of the device [2,11–13]. Fuzzers test more execution paths in firmware by exploring more response classes [11].

From the perspective of fuzzer evaluation, the evaluation index of the grey-box fuzzer for IoT devices that relies on firmware emulation is consistent with the evaluation index of the grey-box fuzzer for general IT systems, both of which are coverage. Therefore, a grey-box fuzzer based on firmware emulation can learn from the evaluation method of a grey-box fuzzer for general IT systems. However, there is currently a lack of unbiased evaluation methods for network-level black-box fuzzers for IoT smart devices.

2.2. Fuzzing Evaluation

The rapid addition and improvement of fuzzing techniques mean that different fuzzers must be constantly compared to show that the latest fuzzers displace the previous state-of-the-art fuzzers. Unfortunately, these evaluations have been ad hoc and haphazard [21]. Klees et al. were the first to study the current state of fuzzing evaluation [17]. They analyzed 32 fuzzing research papers and found that some papers did not use many different real-world benchmarks, had too few trials, used short trial times, or lacked statistical testing [16]. Furthermore, making cross-comparisons across all papers is challenging, as they often use different evaluation settings and configurations, different benchmark procedures, and even different coverage metrics.

After Klees et al., a series of studies related to fuzzing evaluation began to appear [16,17,20]. UNIFUZZ [20] is an open-source platform to evaluate fuzzers comprehensively and quantitatively. UNIFUZZ contains 35 popular fuzzers, benchmark suites of 20 real programs, and 6 types of performance metrics [20]. FuzzBench [16] aims to solve the challenges of lack of evaluation rationality and the high cost of time and computing resources in fuzz test evaluation by providing open-source fuzzer benchmark testing services. FuzzBench uses 22 programs from the OSSFuzz [33] project as the default benchmark suites. In terms of evaluation indicators, FuzzBench focuses on using coverage as

an indicator, including code and bug coverage, independent code coverage metric, and differential coverage. FuzzBench believes that fuzzers running more extensive code paths are more likely to find bugs. Unlike the evaluation metrics that FuzzBench focuses on, Magma believes that the correlation between coverage-deduplicated crashes and real bugs is weak. This means that higher coverage does not necessarily mean better fuzzer effectiveness. Therefore, Magma pays more attention to indicators such as the number and time of bugs that the fuzzer can find. Magma uses forward porting to inject dozens of bugs into the latest version of seven benchmark programs. Magma compared the time-to-bug indicators of six fuzzers in three dimensions: bugs reached, triggered, and detected. In the above fuzzing evaluation work, the selected benchmark programs are all software in the general IT environment. The fuzzers evaluated are all white-box or grey-box fuzzers using coverage guidance.

ProFuzzBench [34] is a recent work that provides benchmark programs for stateful fuzzers for network protocols. It offers 10 protocols, and their corresponding implementation programs, and automates the construction process for these benchmark programs. However, it does not present complete evaluation methods and processes. ProFuzzBench has advanced fuzzing evaluation from software in general IT environments to network protocol programs to some extent. However, in the IoT scenario, firmware vulnerabilities are not only in the network protocol implementation programs, but also in many business function codes [11]. Moreover, the network protocol implementation programs in the firmware are usually not the open-source software chosen by ProFuzzBench, but instead the customized programs by many equipment manufacturers [14]. Therefore, the benchmark programs of ProFuzzBench cannot effectively represent the benchmark firmware images required in the IoT fuzzing evaluation. There is still a lack of practical benchmark suites and complete evaluation methods for black-box protocol fuzzers for IoT devices.

3. Design

This section details the architecture and implementation of IoTFuzzBench, a framework for evaluating black-box protocol fuzzers for IoT devices. In this work, we focus on the network-level black-box fuzzing evaluation of IoT devices. Therefore, we assume that all IoT devices under test support the management and control of the device through network protocols, and we evaluate the fuzzers based on their ability to discover vulnerabilities in the firmware via network interactions. IoTFuzzBench supports the description of fuzzers, benchmark firmware images, and vulnerabilities in evaluation experiments through configuration files. Each benchmark firmware may contain one or more vulnerabilities.

Figure 1 shows the high-level architectural design of IoTFuzzBench. The framework consists of four main components: the scheduler, the database, the task queue, and the job driver. The scheduler reads the evaluation configuration file, generates fuzzing tasks, and invokes the analyzer. The database stores the benchmark bundles, the fuzzer bundles, and the fuzzing results. The task queue manages the fuzzing tasks and schedules them for execution. The job driver creates and runs the benchmark and fuzzer containers, drives the fuzzing process, monitors the process, and records the fuzzing results. The scheduler reads the evaluation configuration file when an evaluation process starts. It combines each fuzzer in the configuration file with each vulnerability in each benchmark to form a triple of fuzzer, benchmark firmware image, and vulnerability. Different triples will correspond to fuzzing tasks that can run independently of each other in the subsequent fuzzing process. For each triple, the scheduler retrieves the benchmark bundle for building the benchmark firmware runtime environment and the fuzzer bundle for creating the fuzzer runtime environment from the database. A typical benchmark bundle includes an emulated firmware image, a Dockerfile, start-up scripts, a benchmark configuration file, and a fuzzing seed update script. A standard fuzzer bundle consists of a stand-alone fuzzer and all its dependencies, a Dockerfile, and start-up scripts. The scheduler creates several independent fuzzing tasks according to the triple, benchmark, and fuzzer bundles, and

configures the corresponding task parameters. These fuzzing tasks are added to the task queue and scheduled for execution by the task queue. In each fuzzing task, the job driver creates a set of independent benchmark containers and fuzzer containers according to the task parameters, drives the fuzzing process, monitors the process, and records the fuzzing results. After each fuzzing task, the fuzzing results are added to the database. After all the fuzzing tasks are finished, the scheduler invokes the analyzer to analyze the fuzzing results of all tasks. It outputs the corresponding fuzzing report and stores it in the database.

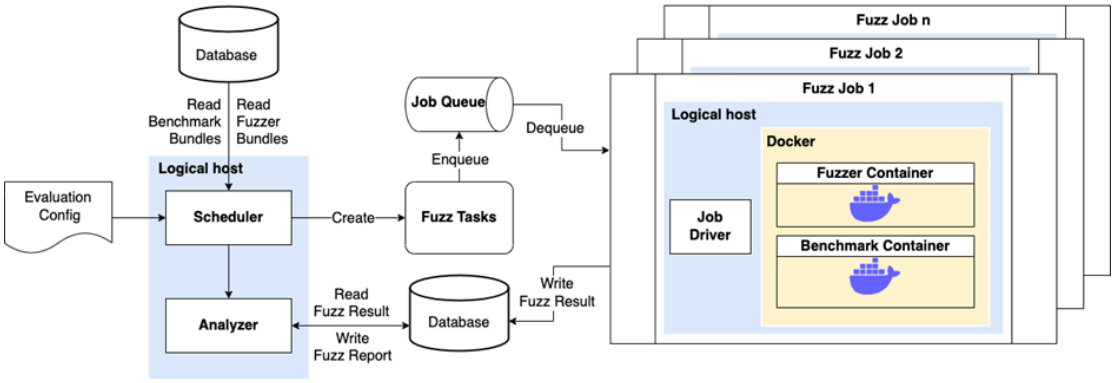


Figure 1. Overview of IoTFuzzBench.

In Section 3.1, we outline the structure of the task driver and the workflow of a single fuzzing job. Section 3.2 discusses the selection principles for benchmark firmware images. Section 3.3 details the process of building benchmark firmware images and addresses specific issues encountered during benchmark environment construction. Section 3.4 provides an overview of the integrated fuzzers and fuzzer bundles used by IoTFuzzBench. Finally, Section 3.5 introduces the evaluation metrics employed by IoTFuzzBench.

3.1. Fuzzing Job Workflow

The workflow of each fuzzing task is shown in Figure 2. Each task runs through the job driver. The driver mainly consists of a benchmark builder, seed updater, fuzzer builder, and monitor.

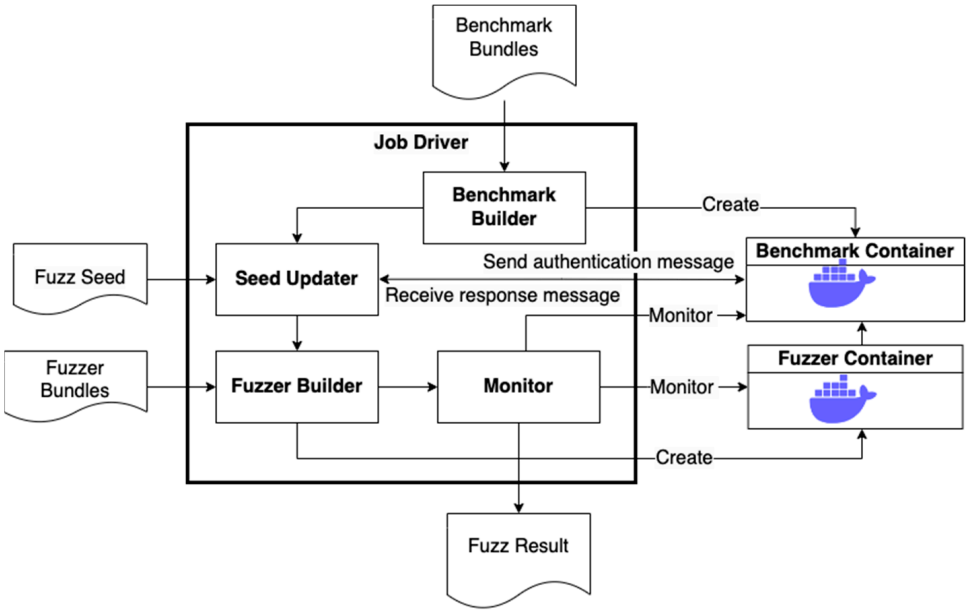


Figure 2. Fuzzing job workflow.

The benchmark builder reads the configuration file in the benchmark bundle to prepare the container runtime parameters. Then, the benchmark builder builds the benchmark image according to the Dockerfile in the benchmark bundle and starts the container according to the runtime parameters to complete the construction of the benchmark container.

The seed updater updates the authentication information in the fuzzing seed. Since the firmware emulation process takes a certain amount of time, when the benchmark container is started, the seed updater cyclically monitors the service survival in the benchmark container and waits for the service to go online. When the service goes online, the seed updater calls the seed update interface implemented in the benchmark bundle to complete the update operation of the fuzzing test seed. Further, the seed updater creates a packet capture process in the container to capture the network traffic during the subsequent fuzzing process.

The monitor is used to monitor the fuzzer container and the benchmark container to judge the fuzzing results and collect primary data for fuzzing evaluation. For the fuzzer container, the monitor periodically checks the survival of the fuzzer container to determine whether the fuzzing process is over. At the same time, the monitor periodically records the consumption of computing resources (such as memory) when the fuzzer is running for subsequent evaluation. For the benchmark container, the emulation tool can create an independent runtime environment to run the benchmark firmware. Therefore, when a bug in the benchmark firmware is triggered, the service of the benchmark firmware will usually be abnormal, and the benchmark container will not exit directly. Therefore, the monitor creates an independent monitoring subprocess, which periodically accesses the benchmark service to monitor whether the bug in the benchmark firmware is successfully triggered. When the benchmark service crashes or the fuzzer container exits or reaches the maximum fuzzing time, the monitor collects all fuzzing results, which include traffic data packets, log information, runtime computing resource consumption, and other information generated during the fuzzing process. These resulting data are written into the database.

3.2. Benchmark Selection Principles

A benchmark suite is essential for evaluating the performance of fuzzers for IoT devices. The benchmark firmware in the suite should be carefully selected to evaluate fuzzers fairly. The selection of benchmark firmware images should follow these principles:

- **Real-world Firmware Images**: Each benchmark firmware image should come from the real world and cover mainstream architectures. According to the findings of Yun [22], firmware images for MIPS and ARM architectures account for more than 90% of all embedded device architectures. Therefore, these benchmark firmware images should include at least MIPS and ARM architectures;
- **Real-world Vulnerabilities**: Each benchmark firmware image should contain at least one real-world vulnerability. Choosing real-world vulnerabilities is more effective for verifying fuzzer performance in practice than artificially implanted vulnerabilities using forward-porting methods [21];
- **Typical Vulnerability Types**: The types of vulnerabilities in the firmware should cover typical IoT device vulnerability types. The existing black-box fuzzing research on IoT devices mainly focuses on typical vulnerability types such as memory corruption, command injection, and denial of service [2,11,12,15]. Therefore, the vulnerabilities in the benchmark firmware images should include these types;
- **Similar Exploit Difficulty and Value of Vulnerabilities**: The vulnerabilities in the firmware should have similar exploit difficulty to ensure that the difficulty of discovering each vulnerability by fuzzers is similar. Since the Common Vulnerability Scoring System (CVSS) considers multiple dimensions such as attack vector, attack complexity, privileges required, and user interaction during calculation; if the CVSS scores of each vulnerability are closer, their exploit difficulty is also relatively similar to some extent. At the same time, the value of different vulnerabilities also affects the evaluation of fuzzers. For example, the value of a high-risk vulnerability may be

higher than that of multiple low-risk vulnerabilities. Therefore, to more easily evaluate the exploit difficulty and value of vulnerabilities discovered by fuzzers, the selected vulnerabilities should have CVSS scores in the same range, such as all being high-risk vulnerabilities with CVSS scores above 7;

- **Determined Seed Messages**: Each vulnerability contained in each benchmark firmware image should have a determined seed message for fuzzing. Since IoT devices usually are managed through network interfaces, a single vulnerability may exist in a message-handling function corresponding to a network interface. Therefore, from a proper perspective, all evaluated fuzzers should have the same standard network interface communication messages as initial seeds for mutation;
- **Emulation and Fidelity of Firmware Images**: Each benchmark firmware should be successfully emulated and easy to use. Unlike programs that can be run directly on x86 general-purpose Linux operating systems, the firmware has more underlying architectures, such as ARM and MIPS. Different firmware has different emulation methods, such as user-mode and system-mode emulation. Therefore, to make benchmark firmware easy to use, researchers should provide rich runtime information for each benchmark firmware, such as architecture information, emulation method, and emulation steps. In addition, a better approach is for developers of evaluation frameworks to provide automated benchmark firmware emulation for each benchmark firmware, but this also means that the workload of developers of evaluation frameworks will increase dramatically. In addition to the emulation of firmware images, the fidelity of firmware images also needs to be verified. For example, many emulated firmware images crash after deep interaction (such as clicking on a web page to set properties), affecting the fuzzer evaluation's validity. Therefore, the fidelity of firmware images also needs to be verified.

Following the 6 principles outlined above, we have constructed a practical benchmark suite consisting of 14 real-world firmware images, 30 real-world vulnerabilities, and corresponding fuzzing seed messages for each benchmark vulnerability. This suite evaluates black-box fuzzers for IoT devices, as shown in Table 1.

3.3. Real-World Benchmark Firmware Images

This section describes the details of the critical steps in constructing a benchmark suite that adheres to the abovementioned six principles. These steps include selecting candidate baseline firmware, matching vulnerabilities with firmware, and addressing specific issues the benchmark suite addresses.

Candidate Benchmark Firmware Images Selection. Since the benchmark firmware needs to be emulated successfully and the success rate of the emulation is relatively low, we first need to screen a batch of firmware images that can be successfully emulated as candidate benchmark firmware images. Considering that the emulation methods that do not depend on hardware are divided into user-mode emulation and system-mode emulation, we mainly screen the firmware that can be successfully emulated from these two aspects. For user-mode emulation, the programs, operating parameters, and dependent resource files in the firmware that need to be emulated by different firmware images may vary significantly. We collect the firmware emulation reports of researchers and manually write Dockerfile to complete the emulation process. The system-mode emulation will emulate the entire operating system when the firmware is running and run the application program in the firmware on the operating system. This method consumes more computing resources than user-mode emulation, but the emulation is more versatile. Some system-mode emulation tools, such as Firmadyne and FirmAE, can directly try to emulate a target firmware image and provide feedback on whether the emulation is successful. Therefore, we first collected some firmware images publicly available on the internet and coded a firmware batch emulation test tool based on Firmware Analysis Toolkit [10] (an automated emulation script based on Firmadyne) to realize automated firmware system-mode emulation testing.

After the emulation work in these two aspects, we have successfully obtained more than 100 candidate benchmark firmware images that can be successfully emulated.

Table 1. The real-world benchmark firmware images and benchmark vulnerabilities.

ID	Firmware Images	Architecture	Vulnerability	Vulnerability Type	CVSS (v3.x)	Protocol	Emulation Tool
1	AC9	ARM	CVE-2018-14558	CI	9.8	HTTP	QEMU
2	AC9	ARM	CVE-2018-16334	CI	8.8	HTTP	QEMU
3	AC9	ARM	CVE-2018-18708	MC	7.5	HTTP	QEMU
4	AC9	ARM	CVE-2020-13390	MC	9.8	HTTP	QEMU
5	AC9	ARM	CVE-2022-25428	MC	9.8	HTTP	QEMU
6	AC9	ARM	CVE-2022-25435	MC	9.8	HTTP	QEMU
7	AC9	ARM	CVE-2022-27016	MC	9.8	HTTP	QEMU
8	AC15	ARM	CVE-2018-5767	MC	9.8	HTTP	QEMU
9	AC15	ARM	CVE-2018-16333	MC	7.5	HTTP	QEMU
10	AC15	ARM	CVE-2020-10987	CI	9.8	HTTP	QEMU
11	DIR806A1	MIPS	CVE-2019-10892	MC	9.8	HNAP	FAT
12	DIR818L	MIPS	CVE-2022-35619	CI	9.8	SOAP	FAT
13	DIR818L	MIPS	CVE-2022-35620	CI	9.8	UPNP	FAT
14	DIR822A1	MIPS	CVE-2019-17621	CI	9.8	UPNP	FAT
15	DIR823GA1	MIPS	CVE-2019-7297	CI	9.8	SOAP	FAT
16	DIR823GA1	MIPS	CVE-2019-7298	CI	8.1	HNAP	FAT
17	DIR823GA1	MIPS	CVE-2020-25366	DoS	9.1	HTTP	FAT
18	DIR823GA1	MIPS	CVE-2020-25367	CI	9.8	SOAP	FAT
19	DIR823GA1	MIPS	CVE-2020-25368	CI	9.8	HNAP	FAT
20	DIR823GA1	MIPS	CVE-2021-43474	CI	9.8	SOAP	FAT
21	DIR825	MIPS	CVE-2020-10215	CI	8.8	HTTP	FAT
22	DIR825	MIPS	CVE-2020-10216	CI	8.8	HTTP	FAT
23	DIR846	MIPS	CVE-2019-17510	CI	9.8	HNAP	FAT
24	DIR865L	MIPS	CVE-2020-13782	CI	8.8	HTTP	FAT
25	DSL3782	MIPS	CVE-2022-34528	MC	8.8	HTTP	FAT
26	HG532	MIPS	CVE-2017-17215	CI	8.8	SOAP	QEMU
27	DIR-859	MIPS	CVE-2022-46476	CI	9.8	SOAP	QEMU
28	TL-WR841N	MIPS	CVE-2020-8423	MC	7.2	HTTP	FAT
29	TL-WR940N	MIPS	CVE-2019-6989	MC	8.8	HTTP	FAT
30	TL-WR940N	MIPS	CVE-2017-13772	MC	8.8	HTTP	FAT

CI: Command Injection. MC: Memory Corruption. DoS: Denial of Service.

Matching CVEs. Each benchmark firmware should contain real-world vulnerabilities in the screening principle of benchmark firmware. Therefore, we need to filter out those firmware images that contain real vulnerabilities among all candidate benchmark firmware images. The straightforward idea is to match all known vulnerabilities against a vulnerability database based on the vendor, model, and version of firmware images. A typical vulnerability database is the National Vulnerability Database (NVD). Nevertheless, this method encounters some challenges in practice.

First, there is an issue of missing information about affected firmware versions in the vulnerability database. Taking NVD as an example, Common Platform Enumeration (CPE) information is used in NVD to describe the vulnerability version, and each piece of CPE information indicates an affected software version. However, unfortunately, in practice, many researchers who reported vulnerabilities did not verify the scope of firmware affected by the reported vulnerabilities, but only verified the existence of reported vulnerabilities on a specific version of the firmware. Typical vulnerabilities that fall into this category include CVE-2022-46570 and CVE-2022-44832. As a result, there is only one piece of CPE information in the vulnerabilities of many IoT devices, and the candidate benchmark firmware cannot be successfully matched to a suitable vulnerability. We try to solve this problem in two ways.

On the one hand, we expanded the scope of the filter by reducing the filter criteria. When inquiring, we only used the keyword combination of manufacturer and model, and did not limit the specific version. Although this increases the number of candidate vulnerabilities, it also means that there are a large number of vulnerabilities that need manual verification. Therefore, on the other hand, we added specific features that can describe the firmware to narrow down the range of candidate vulnerabilities further. We captured the network communication traffic of the successfully emulated firmware and screened some network interface keywords. Experience in software development tells us that, if the communication interfaces of two programs are consistent, their back-end codes may have high similarity. Therefore, the keywords in these network interfaces can better represent the characteristics of the firmware. We combined these network interface keywords as features with manufacturers and models to narrow down the scope of candidate vulnerabilities.

Second, the lack of vulnerability reproduction information prevents candidate vulnerabilities from being successfully reproduced. Simply matching the vulnerability with the firmware through the version number or keywords cannot fully prove that the vulnerability exists in the firmware. The most effective way to prove that a vulnerability exists in a specific firmware is to trigger the target vulnerability in that firmware successfully. A detailed vulnerability description, report, or complete Proof of Concept (PoC) is required for successful vulnerability reproduction. The vulnerability recurrence information in the vulnerability database usually exists in reference links, and each reference link has a corresponding label to identify the type of reference link. For example, there are usually vulnerability reports or PoC in reference links with exploit tags. Unfortunately, not all vulnerabilities have relevant reference links for vulnerability reproduction. The lack of this vulnerability reproduction information leads to a meager success rate of vulnerability reproduction. Therefore, we conducted a secondary screening of all candidate vulnerabilities. Only those with the label links related to vulnerability reproduction were used as candidate vulnerabilities for our subsequent manual reproduction.

After completing the above steps, we successfully obtained each candidate benchmark and the corresponding range of candidate vulnerabilities. We manually verified each candidate vulnerability on its corresponding candidate benchmark. Finally, we successfully validated 30 vulnerabilities on 14 candidate benchmark firmware images. At the same time, for the successfully reproduced vulnerabilities, we recorded the original communication messages of the corresponding network interfaces. This original communication message did not contain the payload that triggered the target vulnerability and was used as a seed message for fuzz testing corresponding to the target vulnerability.

Benchmark Bundle. Since the benchmark firmware needs to be emulated to run, running multiple emulation environments on the same host at the same time may cause conflicts, such as creating conflicts with the same emulated network card or setting the same IP for the emulated firmware. IoTFuzzBench adopts the solution of placing the emulated benchmark firmware in the container and exposing the network communication port. This can ensure the independence of different benchmark emulation environments and ensure that multiple fuzzers do not interfere with each other in the fuzzing process. To enable the emulation environment of the benchmark firmware image to complete automatic construction, a series of resource files are required as support. The combination of resource files used to build a single benchmark firmware emulation environment are named a benchmark bundle. A typical benchmark bundle includes an emulated firmware image, a Dockerfile and start-up scripts, a benchmark configuration file, and a fuzzing seed update script. Below, we introduce how the benchmark bundle plays a role in solving a series of issues while constructing the firmware emulation environment.

The first issue is the construction of the virtual environment for benchmark firmware emulation. Due to different manufacturers, architectures, and hardware dependencies, the emulation methods of each benchmark firmware may vary significantly. Therefore, we coded an independent Dockerfile and corresponding container start-up script for each

benchmark firmware to standardize the construction process of each benchmark firmware emulation environment. The Dockerfile and corresponding container start-up scripts are part of the benchmark bundle.

The second is the access issue of the emulation service. After the benchmark firmware image is emulated successfully in the container, the logical typical network topology of the host where the fuzzing task is located, the benchmark, and the successfully emulated device are as shown in Figure 3. The tap1 network card of the emulation device and the tap0 network card of the benchmark container are in the same independent subnet, and the eth0 network card of the host where the fuzzing task is located and the eth1 network card of the benchmark container are in another independent subnet. Therefore, to enable the network service provided by the emulation device to be accessed by the host where the fuzzing task is located and the subsequent fuzzer, the network service port needs to be forwarded to a particular port on the eth1 network card of the benchmark container. This step can be achieved by creating a separate port forwarding process inside the benchmark container. IoTFuzzBench achieves the above requirements by creating an independent port forwarding process in the benchmark container. Although the default IP assigned by the same firmware is consistent with the listening port after multiple successful emulations, the default IP assigned by different firmware and the listening port may differ after the emulation is successful. When the same firmware image is successfully emulated multiple times, the default assigned IP and listening port are the same each time. However, different firmware images may have different IPs assigned by default and listening ports when the emulation is successful. Therefore, the default assigned IP and open port information of each benchmark firmware need to be defined in a configuration file. This configuration file is included as part of the benchmark bundle.

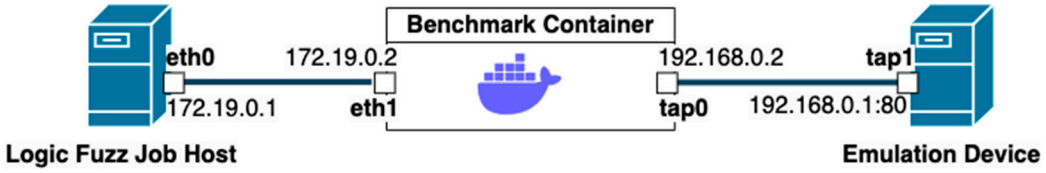


Figure 3. Benchmark firmware runtime logical network topology.

Finally, there is the issue of authentication of fuzzing seeds. Some IoT smart devices require authentication when communicating with the outside world through network interfaces. In this case, the messages used for communication usually carry authentication information to meet the authentication requirements of IoT smart devices. At the same time, due to the timeliness requirement of the authentication information, the authentication information in the communication message used as the seed of the fuzzing test needs to be dynamically updated after the firmware emulation is successful. The fuzzing seed after updating the authentication message is valid. The interface for updating seeds was defined in IoTFuzzBench, and the inputs of this interface were the target IP, target port, and a fuzzing seed message to be updated. The output was a good fuzzing seed with updated authentication information. We coded independent seed update scripts for each benchmark firmware to complete different implementations of this interface. Since all communication interfaces of a single benchmark firmware usually use the same authentication method, the fuzzing seeds of all vulnerabilities in a single benchmark firmware can use the same seed update script. The seed update script is included as part of the benchmark bundle.

3.4. Usable Fuzzers

Fuzzers Selection Principle. To evaluate black-box protocol fuzzers for IoT devices, we need to select representative black-box protocol fuzzers. We mainly follow these principles to select fuzzers:

- **Diverse Sources**: The evaluated fuzzers should have diverse sources, including open-source fuzzers from academic papers, industrial vendors, and an active open-source community;
- **Ease of Use and Extensibility**: Out-of-the-box fuzzers are the most popular type of fuzzer. However, considering that some popular framework-based fuzzers are extensible frameworks rather than specific, out-of-the-box fuzzers—because they do not contain implementations of specific protocols or communication channels—users need to write their own fuzzers using them. However, these fuzzers are also widely used. Therefore, the evaluated fuzzers should also include framework-based fuzzers;
- **Classic Benchmark Fuzzers**: Some classic fuzzers are often used as benchmark fuzzers in academic papers. These fuzzers should also be considered for evaluation.

So far, IoTFuzzBench has integrated seven available black-box fuzzers, including Snipuzz [11], T-Reqs [35], Mutiny [36], Fuzzotron [37], Boofuzz-Default [38], Boofuzz-Byte [38], and Boofuzz-Reversal [38]. These fuzzers all support fuzzing target IoT devices through the network. Among them, Snipuzz and T-Reqs are both articles published in the top security conference ACM CCS 2021 [39] and have been open-sourced. Mutiny is an open-source fuzzer from Cisco with over 500 stars on GitHub. Fuzzotron is an active community fuzzer with more than 350 stars on GitHub that has been continuously updated and maintained over the past six months. Snipuzz, T-Reqs, Mutiny, and Fuzzotron are all out-of-the-box fuzzers. Boofuzz is a typical representative of generative fuzzer frameworks. Similar frameworks include Sulley [40] and KittyFuzzer [41]. Among them, Boofuzz is the successor of Sulley and has better functionality and performance than Sulley. KittyFuzzer is another fuzzer framework inspired by Sulley, but its last update was four years ago and it is no longer maintained. Boofuzz has the most active community and has been continuously updated over the past three months. At the same time, Boofuzz is also widely used as a benchmark fuzzer in various academic papers [11,15,42]. Therefore, we chose Boofuzz as a representative of generative fuzzer frameworks.

It should be noted that Boofuzz, as a classic fuzzing tool, is usually used for three mutation strategies when it is used in comparative experiments of fuzzing by existing research work, including Boofuzz-Default, Boofuzz-Byte, and Boofuzz-Reversal [11]. In the Boofuzz-Default strategy, each message in the input is set to a whole string, and Boofuzz takes the message as a string for mutation testing. In the Boofuzz-Byte strategy, each message in the input is set as a complete byte stream, and each byte in Boofuzz is mutated individually. In the Boofuzz-Reversal strategy, non-data domains are mutated while data domains remain unchanged. IoTFuzzBench evaluates Boofuzz with these three strategies as three separate fuzzers. As Boofuzz is a flexible and customizable fuzzer, it is necessary to manually code a corresponding fuzzing script that can run independently for each fuzzing seed message. In this script, researchers can customize the format of the message, the fields that need to be mutated, and the mutation method. Performing this process manually is, undoubtedly, very complex and time-consuming, so we implemented the automated generation process of Boofuzz scripts by employing these three strategies based on our published work PDFuzzerGen [9]. Every time IoTFuzzBench conducts a Boofuzz-related evaluation experiment, the corresponding Boofuzz script is dynamically generated, and the fuzzing process is automatically completed after generation;

Fuzzer Runtime Environment: In order to better complete the evaluation experiment, we chose Docker as the primary container tool for fuzzing experiments. Encapsulating the fuzzer in a Docker container has several benefits. First, resource allocation and isolation of Docker containers are more accessible to control than fuzzing on physical machines. At the same time, information, such as the traffic generated by the fuzzer and the computing resource consumption during runtime, is also more convenient to be counted. This provides a reasonable basis for a fair fuzzing evaluation. Second, Docker is more lightweight and uses fewer computing resources than a virtual machine. Finally, Docker can be more convenient for operation, maintenance, and management;

Fuzzer Bundle: Similar to the benchmark bundle, we named all resource files used to build a single fuzzer runtime environment as a fuzzer bundle. A typical fuzzer bundle includes a stand-alone fuzzer and all its dependencies, a Dockerfile, and start-up scripts. Compared with building benchmark firmware, since the fuzzer operating environment does not require firmware emulation steps, building the fuzzer operating environment is more straightforward. For each fuzzer participating in the evaluation, we wrote a Dockerfile that supported its independent operation. It is worth noting that, to enable each fuzzer to have a relatively uniform operation and scheduling method, IoTFuzzBench provides three runtime parameters for each fuzzer container in the form of runtime parameters, which are the operating environment of the benchmark firmware to be tested, i.e., IP, port, and updated fuzzing seeds. The start-up scripts of each fuzzer specifically use these three parameters and implement the start-up process of the fuzzer separately.

3.5. Performance Metrics

Current black-box protocol fuzzers for IoT devices need comprehensive and practical performance metrics when evaluating them. Therefore, we systematically studied performance metrics for fuzzing existing black-box protocols, combined with metrics adopted by grey-box fuzzing evaluation methods [11,13–16,20,21]. Finally, we recapitulated and presented a set of metrics that can be grouped into five classifications: number of discovered vulnerabilities, speed of discovering vulnerabilities, stability of discovery process, number of unique responses, and runtime overhead. Below, we propose specific metrics for each classification and suggest applications in practical evaluations:

Number of Discovered Vulnerabilities: IoTFuzzBench provides the raw seed for the mutation of each vulnerability so that crashes triggered by fuzzers can be associated with specific vulnerabilities. The number of vulnerabilities found by each fuzzer is an essential and valid indicator. Due to the random nature of fuzzing, multiple stable fuzzing experiments must be performed to provide more reliable results [20]. Based on the results of multiple experiments, IoTFuzzBench adds a set of statistical indicators, such as variance, median, and mean, to enrich the results and uses box plots and critical difference diagrams to make more decadent visual displays of these results.

It is worth noting that the critical difference diagram is an essential experimental result for comparing different fuzzers. We show concrete examples in Section 4.2. This diagram was introduced by Demsar [43] and is often used in machine learning to compare algorithms on multiple benchmarks [16]. The diagram compares the fuzzers of all benchmarks by visualizing their average rank and statistical significance [16]. The average rank is calculated from the median number of vulnerabilities found by each fuzzer across all experiments. Groups of fuzzers connected by thick lines are not significantly different;

Speed of Discovering Vulnerabilities: Finding vulnerabilities quickly and efficiently is significant. Since IoTFuzzBench can correlate the crash triggered by the fuzzer with a specific vulnerability, the time for each fuzzer to trigger the crash of the benchmark firmware can be approximately equivalent to the time when the corresponding vulnerability is discovered. On this basis, IoTFuzzBench uses two indicators to measure the speed of discovering vulnerabilities. First, for all the vulnerabilities selected for testing in a single experiment, IoTFuzzBench can create multiple tasks in parallel to complete the testing process and record the time when each fuzzer discovers these vulnerabilities. This is a relatively quantitative indicator. Second, since the maximum fuzzing time of each task is the same (for example, each task runs for 24 h), IoTFuzzBench can draw a curve of the number of vulnerabilities discovered by the fuzzer during the maximum fuzzing time over time. We can qualitatively judge the speed at which each fuzzer finds vulnerabilities through the slope of the curve. A fuzzer with a higher slope over time intervals finds bugs faster;

Stability of Discovery Process: Due to the randomness of fuzzing, there may be differences in the performance of fuzzers in multiple experiments under the same conditions. Therefore, stability is another important indicator to measure the performance of fuzzers.

When a fuzzer is more stable in finding vulnerabilities, its performance in practice is more reliable and predictable. IoTFuzzBench calculates the relative standard deviation (RSD) [20] of the number of vulnerabilities found by each fuzzer in each experiment as a specific quantitative indicator. Lower RSD values mean higher stability;

Number of Unique Responses: Since IoT smart devices usually communicate with the outside world through network interfaces, the number of unique response messages from IoT devices is an important indicator to measure the performance of fuzzers. A higher number of unique responses triggered indicates that the fuzzer may have detected more code execution paths in IoT smart devices. The more code paths detected, the more vulnerabilities the fuzzer has the potential to find. From this perspective, although the number of unique responses cannot be directly equivalent to the precise coverage index used in grey-box or white-box fuzzing methods, it can also represent the coverage effect of the fuzzer on the code execution path to a certain extent. IoTFuzzBench uses two metrics to measure the number of unique responses triggered by a fuzzer. On the one hand, IoTFuzzBench can count the unique responses triggered by each fuzzer to the same vulnerability in a single experiment and the average ranking across multiple experiments. On the other hand, IoTFuzzBench can graph the number of responses over time as each fuzzer fuzzes each vulnerability. The higher the slope of the curve, the faster the fuzzer can find a response;

Runtime Overhead: Runtime overhead is an often-overlooked metric. However, knowing the runtime overhead of a fuzzer during fuzzing is instructive when resources for running fuzzing experiments are limited. When different fuzzers can successfully find the same vulnerability, we believe the fuzzer that spends fewer computational resources performs better. IoTFuzzBench uses memory consumption as a metric to measure the overhead of different fuzzers.

4. Experiments and Results

Using IoTFuzzBench, we performed extensive experiments on a state-of-the-art set of seven black-box protocol fuzzers, including Boofuzz-Default, Boofuzz-Byte, Boofuzz-Reversal, Mutiny, Snipuzz, T-Reqs, and Fuzzotron. Meanwhile, we comprehensively compared these fuzzers on 30 vulnerabilities in 14 benchmark firmware images according to 5 categories of performance metrics and repeated the experiment 10 times.

4.1. Experiment Settings

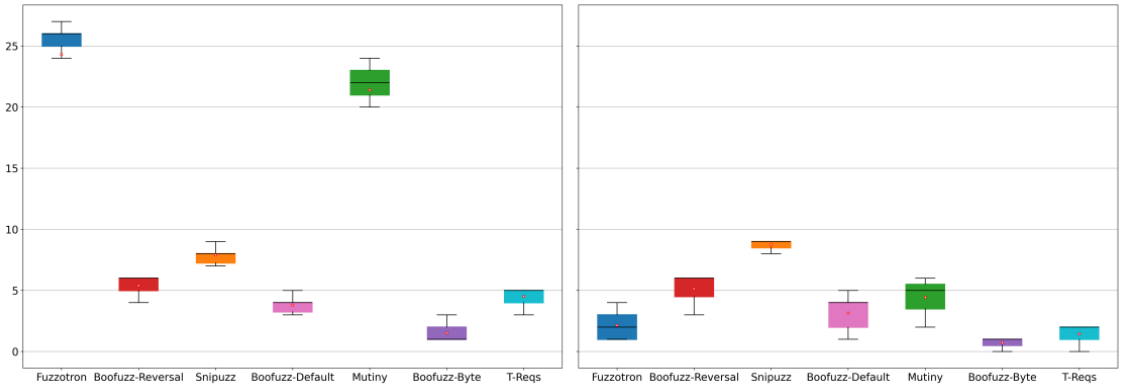
Since IoTFuzzBench establishes the relationship between the benchmark and its vulnerabilities, we treated each vulnerability as an independent target to be tested. IoTFuzzBench creates separate fuzzing tasks for each fuzzer/target combination in a complete fuzzing experiment. In each task, the standard communication message of the communication interface corresponding to the target vulnerability was used as the seed file, and the fuzzing process ran for 24 h. All evaluated black-box protocol fuzzers fuzzed the target IoT device from outside through the network and received the feedback response of the target service to judge the target service status. When the target service did not respond or the response timed out, the target service was considered to have crashed. IoTFuzzBench uniformly implements failure detection as detecting the crash of the target service.

Environments: We conducted all experiments on a single server: 64 Intel Xeon Silver 4314 CPU cores, 2.40 GHz, 256 GB of RAM, 64-bit Ubuntu 22.04 LTS. In each task of each set of experiments, the fuzzer ran in an independent Docker container and fuzzed the target benchmark firmware running in another independent Docker container. Benchmark containers were not shared between tasks. All fuzzers used default or recommended fuzzing parameters.

Next, we present and analyze the evaluation results in detail according to the five categories of performance metrics.

4.2. Number of Discovered Vulnerabilities

IoTFuzzBench mainly monitors whether the target benchmark firmware has crashed as the primary basis for whether the fuzzer triggers the vulnerability. Figure 4a shows the number of crashes that fuzzers triggered in the benchmark firmware provided by IoTFuzzBench in 10 experiments. The experimental results show that Fuzzotron and Mutiny found the most crashes. Among all 30 vulnerabilities, Fuzzotron even found 27 crashes, which is significantly more than the number found by other fuzzers. We were surprised by the excellent performance of Fuzzotron and Mutiny. Therefore, we analyzed in detail the complete data packets of the fuzzing process corresponding to all crashes. Further, we tried to locate the message that triggered the crash and analyze the cause of the crash.



(**a**) (**b**)

Figure 4. The number of crashes detected by fuzzers: (**a**) The number of crashes without regression judgment. (**b**) The number of crashes with regression judgment.

Through extensive analysis, we found that most crashes caused by Fuzzotron and Mutiny were not due to the real vulnerabilities corresponding to the raw seeds. Most crashes caused by Fuzzotron and Mutiny were denial-of-service effects. Specifically, many mutated messages generated by Fuzzotron and Mutiny during the fuzzing caused a denial-of-service attack on the emulation device under test. There were two reasons for this phenomenon. On the one hand, the limited computing resources of the emulation IoT devices led to low processing capacity for many requests. This situation of limited computing resources is widespread even on real devices [22]. On the other hand, some applications in the firmware had defects in processing requests. When they received a malformed variation message simultaneously, their response time was too long, resulting in a denial-of-service during the fuzzing process. However, the emulation device service returned to normal after the fuzzing process.

From the above results, Fuzzotron and Mutiny have significant advantages in finding the denial-of-service problems of IoT devices. However, at the same time, this phenomenon that causes a denial-of-service effect in the fuzzing process and service recovery after the fuzzing is over is not an actual vulnerability. Therefore, we further enriched the monitor and increased the regression judgment of service availability. After a crash was found, the monitor performed a secondary regression judgment on the availability of the emulation device at a specific interval to avoid the interference of the above denial-of-service effect on the results.

Then, we conducted a complete experiment again and counted the results, as shown in Figure 4b. The critical difference ranking of each fuzzer is shown in Figure 5. The experimental results show that the denial-of-service effect caused by Fuzzotron and Mutiny can be effectively detected, and this situation was not considered as the vulnerability found. Among all fuzzers, Snipuzz performed best in finding vulnerabilities, ranking first in the number of vulnerabilities found in each experiment. The performances of Boofuzz-

Reversal and Mutiny were slightly weaker than that of Snipuzz. Boofuzz-Byte was the worst performer. This shows that simply changing the communication message into a byte stream cannot effectively discover the vulnerabilities in the intelligent devices of the Internet of Things.

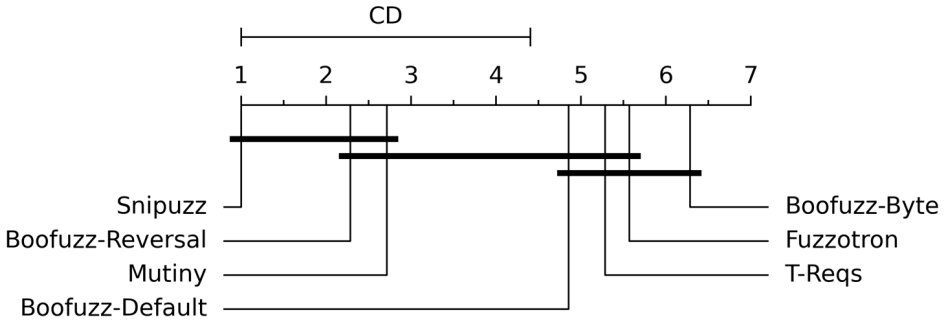


Figure 5. Critical difference diagram of the number of vulnerabilities found by the fuzzer in the experiment.

Due to the randomness of fuzzing, if each fuzzer can find a specific vulnerability in any experiment, we believe the fuzzer can successfully find this vulnerability. Table 2 provides details of the vulnerabilities that each fuzzer can successfully discover. Among the 30 verified vulnerabilities, the evaluated fuzzers triggered 14, and no fuzzer triggered more than 9 vulnerabilities. It is worth noting that, although Snipuzz found the most vulnerabilities, other fuzzers could still find some vulnerabilities that Snipuzz could not. For example, Fuzzotron and Mutiny successfully discovered CVE-2020-10215, Fuzzotron successfully discovered CVE-2018-5767 and CVE-2020-10987, Boofuzz-Reversal successfully discovered CVE-2022-25428, and Mutiny successfully discovered CVE-2020-10216. This shows that different fuzzers may have their own advantages in discovering vulnerabilities, and a better-performance fuzzer cannot directly replace all other fuzzers of the same type.

4.3. Speed of Discovering Vulnerabilities

Table 2 lists the average time for the fuzzer to discover each vulnerability. The vulnerability detection time of each fuzzer in Table 2 varies from a few seconds to thousands of seconds. Figure 6 visualizes the curve of the number of vulnerabilities found by each fuzzer over time during each fuzzing task. From the experimental results, we can see the speed of each fuzzer to discover vulnerabilities. Snipuzz quickly discovered multiple vulnerabilities within tens of seconds after the experiment began and performed the best. However, the speed of the discovered vulnerabilities slowed down and stopped growing after 330 s. About 10 to 50 s after the start of the experiment, Boofuzz-Default performed the best. The number of vulnerabilities found by Boofuzz-Default increased rapidly but then stopped growing. In about 50 s to 100 s, Mutiny found the fastest vulnerabilities, and the number of vulnerabilities found after more than 70 s exceeded Boofuzz-Default. Boofuzz-Default was the fastest to discover vulnerabilities between 1000 s and 1600 s. The experimental results show that different fuzzers significantly differ in the speed of discovering vulnerabilities in different periods. When researchers need a fuzzer that can quickly find vulnerabilities in a few minutes, but there is no requirement for the number of vulnerabilities found, Snipuzz, Mutiny, and Boofuzz-Default may be better choices. Snipuzz is a better choice when researchers need a fuzzer to find many vulnerabilities over a long time.

Table 2. Average time to discover vulnerabilities per fuzzer, in seconds.

Vulnerability	Snipuzz	Boofuzz Reversal	Mutiny	Boofuzz Default	T-Reqs	Fuzzotron	Boofuzz Byte
CVE-2017-13772	**329.3**	1490.8	-	-	-	-	-
CVE-2017-17215	-	-	-	-	-	-	-
CVE-2018-5767	-	-	-	-	-	**573.8**	-
CVE-2018-14558	-	-	-	-	-	-	-
CVE-2018-16333	**14.1**	-	-	-	-	-	-
CVE-2018-16334	-	-	-	-	-	-	-
CVE-2018-18708	-	-	-	-	-	-	-
CVE-2019-6989	**12.4**	1524.6	-	-	-	-	-
CVE-2019-7297	-	-	-	-	-	-	-
CVE-2019-7298	-	-	-	-	-	-	-
CVE-2019-10892	234.3	1666.5	73.5	**33.3**	78.4	666.4	643.3
CVE-2019-17510	**18.3**	-	-	-	-	-	-
CVE-2019-17621	-	-	-	-	-	-	-
CVE-2020-8423	-	-	-	-	-	-	-
CVE-2020-10215	-	-	**104.7**	-	-	2785.8	-
CVE-2020-10216	-	-	**39.8**	-	-	-	-
CVE-2020-10987	-	-	-	-	-	**176.8**	-
CVE-2020-13390	167.4	-	89.3	**36.3**	-	-	-
CVE-2020-13782	-	-	-	-	-	-	-
CVE-2020-25366	-	-	-	-	-	-	-
CVE-2020-25367	-	-	-	-	-	-	-
CVE-2020-25368	-	-	-	-	-	-	-
CVE-2021-43474	-	-	-	-	-	-	-
CVE-2022-25428	-	**1278.4**	-	-	-	-	-
CVE-2022-25435	107.6	1186.4	69.6	**34.9**	-	-	-
CVE-2022-27016	122.4	1234.2	68.4	**43.4**	-	-	-
CVE-2022-34528	-	-	-	-	-	-	-
CVE-2022-35619	-	-	-	-	-	-	-
CVE-2022-35620	-	-	-	-	-	-	-
CVE-2022-46476	-	-	-	-	-	-	-

"-" indicates that the vulnerability was never found during the entire evaluation process.

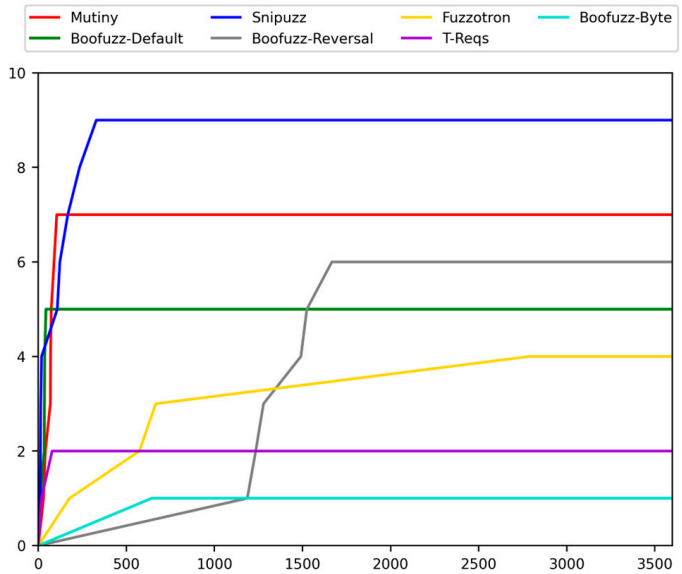


Figure 6. The number of vulnerabilities found by each fuzzer varies over time.

4.4. Stability of Discovery Process

Figure 7 shows the RSD of the number of vulnerabilities found by each fuzzer across ten experiments. A lower RSD indicates a better stability of the fuzzer [20]. We can find the

following observations. First, all fuzzers differed in the number of vulnerabilities found in each set of experiments. This shows that the fuzzer has randomness in the fuzzing process. This also illustrates the importance of repeating fuzzing experiments multiple times. Secondly, Snipuzz and Boofuzz-Byte had lower RSD, while Boofuzz-Reversal, Boofuzz-Default, and Fuzzotron had higher RSD values, with little difference. Mutiny had the highest RSD value. This shows that Snipuzz and Boofuzz-Byte have good stability, while the stability of Mutiny's vulnerability discovery process is relatively poor. It is important to note that the stability of discovery vulnerabilities process indicators is adjunct to the number of discovered vulnerabilities indicators, since finding more vulnerabilities is more critical than consistently finding fewer vulnerabilities.

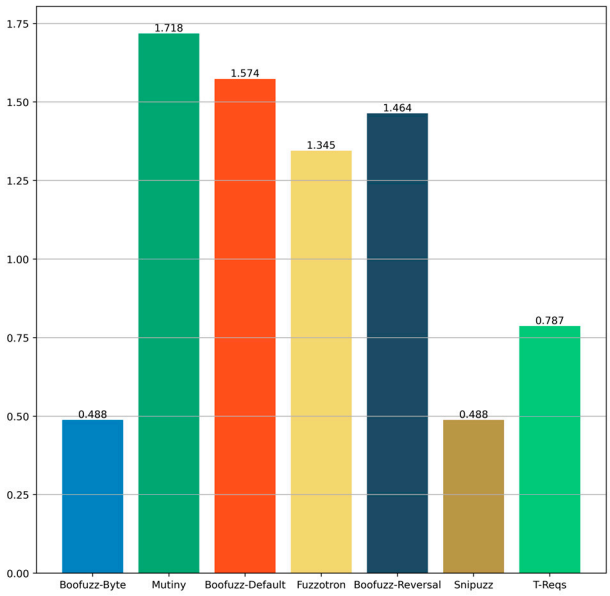


Figure 7. The RSD of the number of vulnerabilities.

4.5. Number of Unique Responses

Figure 8 shows the average ranking of each fuzzer for the number of triggered responses across 10 experiments. Figure 9 shows the curve of the average number of responses triggered by each fuzzer in the process of fuzzing some vulnerabilities in the 10 experiments. The above experimental results show that the Fuzzotron fuzzer triggered the most significant unique number of responses and the highest average ranking, while Boofuzz-Default and Boofuzz-Byte performed the worst. The average ranks of the rest of the fuzzers were relatively close. The number of responses can reflect the code execution path triggered by the fuzzer from the side. Executing more code paths means the fuzzer is more likely to find a vulnerability. The large number of responses that Fuzzotron and Mutiny can trigger also explains why they can find the problems related to the denial-of-service of the target device.

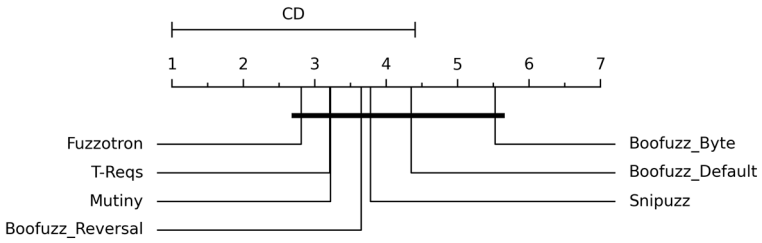


Figure 8. Critical difference diagram of the number of responses triggered by fuzzers in the experiment.

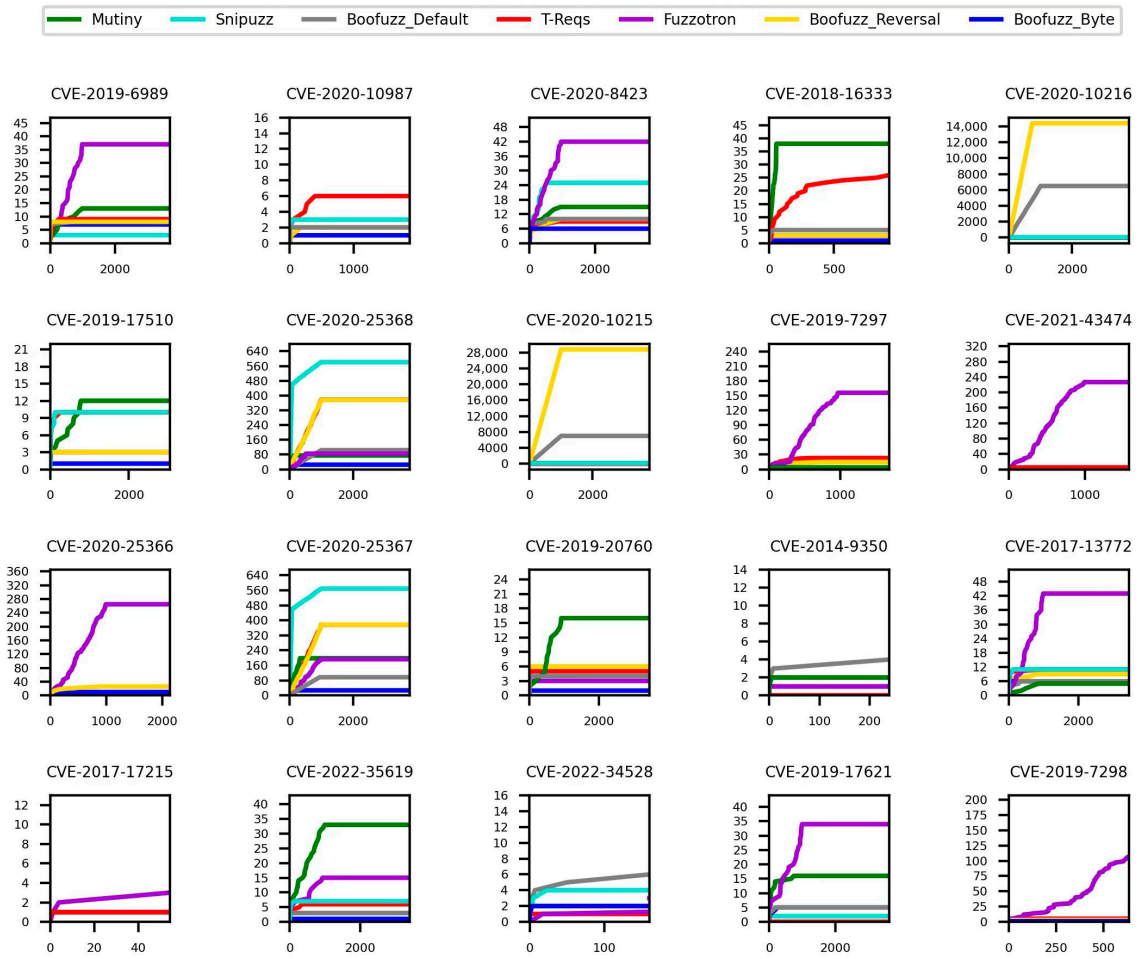


Figure 9. The average number of unique responses found by each fuzzer over time for a subset of typical vulnerabilities across 10 experiments, in seconds.

4.6. Runtime Overhead

Figure 10 shows the average memory consumption of each fuzzer in the process of fuzzing different vulnerabilities. Fuzzotron averaged less memory consumption than other fuzzers during fuzzing, but its maximum memory consumption was higher than Boofuzz-Byte, Mutiny, and T-Reqs. The memory consumption of Boofuzz-Byte, Mutiny, and T-Reqs was relatively close, and the memory consumption of Boofuzz-Default and Boofuzz-Reversal was the most. Although the memory consumption of each fuzzer was different, the average memory consumption values were within 200 MB. Compared with the memory of several GBs or dozens of GBs in the current server, the memory consumption of the fuzzer is not the bottleneck that limits the performance of the fuzzer to discovering vulnerabilities. However, if researchers need to deploy dozens or even hundreds of fuzzers with fewer computing resources, Fuzzotron, Boofuzz-Byte, Mutiny, and T-Reqs, with an average memory consumption of less than 30 MB, may be better choices.

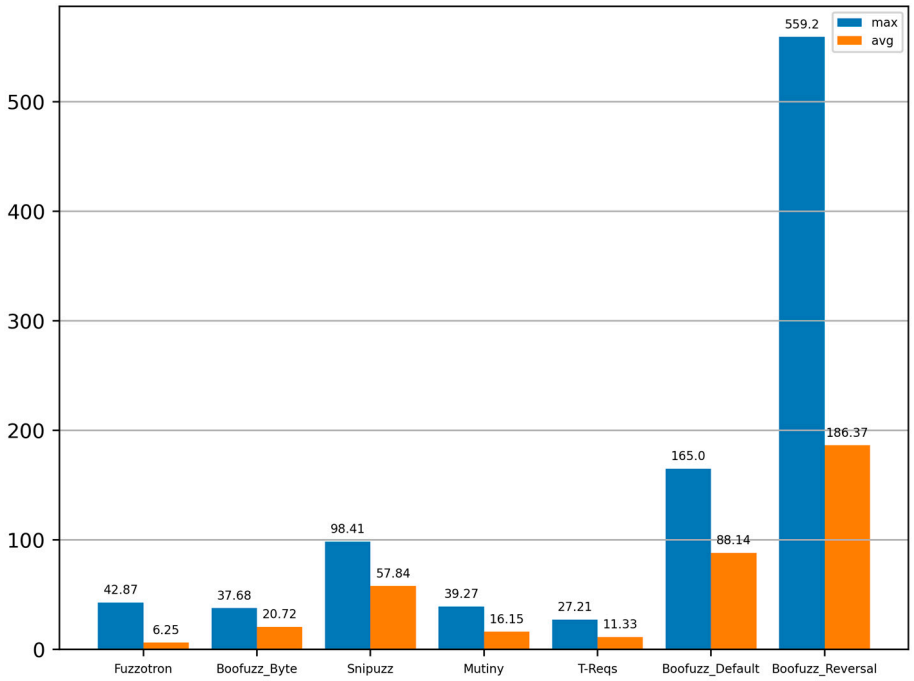


Figure 10. Average and maximum memory consumption (MB) of each fuzzer.

5. Discussion and Future Work

Although IoTFuzzBench can facilitate the complete process of fuzzing evaluation of black-box protocols for IoT smart devices, since this is the first step in an automated, comprehensive fuzzing evaluation, we would like to discuss current design limitations and explore directions for future improvements.

More Benchmark Vulnerability Types: The current benchmark vulnerabilities have room for further expansion, mainly reflected in the types of vulnerabilities in the benchmark firmware. The currently chosen types of vulnerabilities mainly include memory corruption and command injection vulnerabilities. Although these two types of vulnerabilities account for the highest proportion of all vulnerabilities in IoT smart devices in the vulnerability database and are highly representative, more types of vulnerabilities will undoubtedly further reflect the differences in different types of vulnerabilities among the performance of fuzzers. Enriching the type and number of vulnerabilities in IoTFuzzBench is part of our ongoing work to improve IoTFuzzBench;

More Fuzzer Types: IoTFuzzBench is currently focused on evaluating IoT black-box protocol fuzzers. Nevertheless, the extensibility of IoTFuzzBench allows it to be further improved for evaluating other types of IoT protocol fuzzers, such as grey-box protocol fuzzers. The key to achieve this step is collecting the coverage of emulated firmware images and successfully feeding it back to the target grey-box fuzzer. Meanwhile, for a single fuzzer, repeated experiments with different fuzzing parameters can help fuzzer developers debug and optimize their fuzzers. This will be one of our next steps;

Vulnerability Score: Most of the vulnerabilities selected in IoTFuzzBench are high-risk or critical-risk vulnerabilities in the CVSS. Although these vulnerabilities are very representative, the harm and value of each vulnerability are relatively close. In practice, many vulnerabilities still vary widely in harm and value. A fuzzer that finds one high-value bug is likely to perform better than a fuzzer that finds multiple low-value bugs. Therefore, the quality of the vulnerabilities found by the fuzzer can also be used as a class of indicators to further measure the performance of the fuzzer;

Comprehensive Scoring Method: IoTFuzzBench currently evaluates fuzzers on five categories of metrics that reflect different aspects of fuzzing performance. So far, these metrics are independent and do not provide a single score that can measure the combined performance of individual fuzzers. However, we also recognize that some researchers may

want to measure the combined performance of individual fuzzers using a single score. In such cases, it may be necessary to devise a composite scoring method. A composite scoring method could aggregate indicators of different dimensions and calculate a specific value to evaluate the fuzzer performance quantitatively. Such a method must also consider the priority and weight relationship between various indicators, which may vary depending on the fuzzing objectives and scenarios. Designing a comprehensive scoring method is one of our next steps;

Fuzzer Performance Analysis: Different fuzzers may have different performances on various performance metrics, and analyzing the root causes of this differential performance would help researchers to optimize and improve their fuzzers. The factors that may affect the fuzzing performance include the fuzzing parameter setting, the test case generation strategy, the target firmware characteristic, and the inherent randomness of fuzzing. So far, IoTFuzzBench has focused on improving the efficiency of fuzzing evaluation for researchers and reducing the burden of fuzzing evaluation in the IoT scenario. Providing further analysis methods is another of our next steps;

Formal Methods: Using formal methods can help improve the accuracy, completeness, scalability, and applicability of the system [44,45]. Integrating formal methods with the IoTFuzzBench may be approached from several aspects, for example, modeling part of IoTFuzzBench precisely, using modularity and compositionality to divide the whole system into smaller subsystems, and considering data independence in the fuzzing process. Integrating formal methods into the IoTFuzzBench framework is one of our future directions.

6. Conclusions

In this paper, we present and implement IoTFuzzBench, a scalable, modular, metrics-driven IoT black-box protocol fuzzer evaluation framework for evaluating fuzzers comprehensively and fairly. IoTFuzzBench implements an automated and standardized fuzzer evaluation process, including 7 fuzzers, 14 real-world benchmark firmware images, 30 verified real-world vulnerabilities in benchmark firmware, whole seeds corresponding to each vulnerability, and 5 categories of evaluation metrics. Using IoTFuzzBench, we comprehensively compared state-of-the-art fuzzers. The experimental results show that IoTFuzzBench can effectively evaluate the differential performance of different fuzzers on every indicator. This demonstrates the importance of composite metrics. With the adoption of real-world benchmark evaluation frameworks such as IoTFuzzBench, IoT black-box protocol fuzzer evaluations will become reproducible, allowing researchers to demonstrate the actual contribution of new fuzzing methods.

Author Contributions: Conceptualization, Y.C. and W.H.; methodology, Y.C. and W.F.; software, Y.C. and W.C.; validation, Y.C., W.C. and G.Y.; resources, W.F.; data curation, W.C.; writing—original draft preparation, Y.C.; writing—review and editing, Y.C., W.L. and W.H.; visualization, W.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the major project of Science and Technology Innovation 2030, "The next generation of Artificial Intelligence", under Grant Number 2021ZD0111400, and the Fundamental Research Funds for the Central Universities under Grant Number 3132018XNG1814 and 3132018XNG1815.

Data Availability Statement: The data used to support the findings of this study are available from the corresponding author upon request. The disclosed security vulnerabilities used and found in this paper can be accessed in the CVE (https://cve.mitre.org/, accessed on 8 July 2023).

Acknowledgments: We would like to sincerely thank the reviewers for their insightful comments, which helped us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Friha, O.; Ferrag, M.A.; Shu, L.; Maglaras, L.; Wang, X. Internet of Things for the Future of Smart Agriculture: A Comprehensive Survey of Emerging Technologies. *IEEE/CAA J. Autom. Sin.* **2021**, *8*, 718–752. [CrossRef]
2. Redini, N.; Continella, A.; Das, D.; Pasquale, G.D.; Spahn, N.; Machiry, A.; Bianchi, A.; Kruegel, C.; Vigna, G. Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 484–500.
3. Number of Internet of Things (IoT) Connected Devices Worldwide from 2019 to 2021, with Forecasts from 2022 to 2030. Available online: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (accessed on 18 June 2023).
4. Travel Routers, NAS Devices among Easily Hacked IoT Devices. Available online: https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/ (accessed on 18 June 2023).
5. Lack of IoT Security Could Undermine Growth. Available online: https://www.rsaconference.com/library/blog/lack-of-iot-security-could-undermine-growth (accessed on 18 June 2023).
6. 2020 Unit 42 IoT Threat Report. Available online: https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf (accessed on 18 June 2023).
7. Zhu, X.; Wen, S.; Camtepe, S.; Xiang, Y. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv. (CSUR)* **2022**, *54*, 1–36. [CrossRef]
8. Feng, X.; Zhu, X.; Han, Q.L.; Zhou, W.; Wen, S.; Xiang, Y. Detecting Vulnerability on IoT Device Firmware: A Survey. *IEEE/CAA J. Autom. Sin.* **2022**, *10*, 25–41. [CrossRef]
9. Cheng, Y.; Fan, W.; Huang, W.; Yu, G.; Han, Y.; Dong, H.; Liu, W. PDFuzzerGen: Policy-Driven Black-Box Fuzzer Generation for Smart Devices. *Secur. Commun. Netw.* **2022**, *2022*, 9788219. [CrossRef]
10. Toolkit to Emulate Firmware and Analyse It for Security Vulnerabilities. Available online: https://github.com/attify/firmware-analysis-toolkit (accessed on 18 June 2023).
11. Feng, X.; Sun, R.; Zhu, X.; Xue, M.; Wen, S.; Liu, D.; Nepal, S.; Xiang, Y. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 15–19 November 2021; pp. 337–350.
12. Shu, Z.; Yan, G. IoTInfer: Automated Blackbox Fuzz Testing of IoT Network Protocols Guided by Finite State Machine Inference. *IEEE Internet Things J.* **2022**, *9*, 22737–22751. [CrossRef]
13. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, San Diego, CA, USA, 18–21 February 2018.
14. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering Vulnerabilities in COTS IoT Devices through Blackbox Fuzzing Web Management Interface. *Secur. Commun. Netw.* **2019**, *2019*, 5076324. [CrossRef]
15. Zhang, Y.; Huo, W.; Jian, K.; Shi, J.; Liu, L.; Zou, Y.; Zhang, C.; Liu, B. ESRFuzzer: An enhanced fuzzing framework for physical SOHO router devices to discover multi-Type vulnerabilities. *Cybersecurity* **2021**, *4*, 24. [CrossRef]
16. Metzman, J.; Szekeres, L.; Simon, L.; Sprabery, R.; Arya, A. Fuzzbench: An open fuzzer benchmarking platform and service. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 23–28 August 2021; pp. 1393–1403.
17. Klees, G.; Ruef, A.; Cooper, B.; Wei, S.; Hicks, M. Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 15–19 October 2018; pp. 2123–2138.
18. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by program transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 697–710.
19. Li, Y.; Chen, B.; Chandramohan, M.; Lin, S.; Liu, Y.; Tiu, A. Steelix: Program-state based binary fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 4–8 September 2017; pp. 627–637.
20. Li, Y.; Ji, S.; Chen, Y.; Liang, S.; Lee, W.; Chen, Y.; Lyu, C.; Wu, C.; Beyah, R.; Cheng, P.; et al. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), virtually, 11–13 August 2021; pp. 2777–2794.
21. Hazimeh, A.; Herrera, A.; Payer, M. Magma: A ground-truth fuzzing benchmark. In Proceedings of the ACM on Measurement and Analysis of Computing Systems, New York, NY, USA, 30 November 2020; Volume 4, pp. 1–29.
22. Yun, J.; Rustamov, F.; Kim, J.; Shin, Y. Fuzzing of Embedded Systems: A Survey. *ACM Comput. Surv.* **2022**, *55*, 1–33. [CrossRef]
23. IoTFuzzBench. Available online: https://github.com/a101e-lab/IoTFuzzBench (accessed on 18 June 2023).
24. Lee, S.; Han, H.S.; Cha, S.K.; Son, S. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In Proceedings of the 29th USENIX Conference on Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2613–2630.
25. Han, H.S.; Oh, D.H.; Cha, S.K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2019, San Diego, CA, USA, 24–27 February 2019.
26. Dinh, S.T.; Cho, H.; Martin, K.; Oest, A.; Zeng, K.; Kapravelos, A.; Ahn, G.; Bao, T.; Wang, R.; Doupe, A.; et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2021, virtually, 21–25 February 2021.
27. Huang, H.; Yao, P.; Wu, R.; Shi, Q.; Zhang, C. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1613–1627.

28. Aschermann, C.; Schumilo, S.; Blazytko, T.; Gawlik, R.; Holz, T. REDQUEEN: Fuzzing with Input-to-State Correspondence. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2019, San Diego, CA, USA, 24–27 February 2019; pp. 1–15.
29. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. CollAFL: Path Sensitive Fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 679–696.
30. Zheng, Y.; Davanian, A.; Yin, H.; Song, C.; Zhu, H.; Sun, L. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 1099–1114.
31. Chen, D.D.; Egele, M.; Woo, M.; Brumley, D. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2016, San Diego, CA, USA, 21–24 February 2016.
32. Zhang, Y.; Huo, W.; Jian, K.; Shi, J.; Lu, H.; Liu, L.; Wang, C.; Sun, D.; Zhang, C.; Liu, B. SRFuzzer: An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In Proceedings of the 35th Annual Computer Security Applications Conference, New York, NY, USA, 9–13 December 2019; pp. 544–556.
33. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Available online: https://github.com/google/oss-fuzz (accessed on 18 June 2023).
34. Natella, R.; Pham, V.T. Profuzzbench: A benchmark for stateful protocol fuzzing. In Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, New York, NY, USA, 11–17 July 2021; pp. 662–665.
35. Jabiyev, B.; Sprecher, S.; Onarlioglu, K.; Kirda, E. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 15–19 November 2021; pp. 1805–1820.
36. Mutiny Fuzzing Framework. Available online: https://github.com/Cisco-Talos/mutiny-fuzzer (accessed on 18 June 2023).
37. Fuzzotron: A TCP/UDP Based Network Daemon Fuzzer. Available online: https://github.com/denandz/fuzzotron (accessed on 18 June 2023).
38. Boofuzz: Network Protocol Fuzzing for Humans. Available online: https://github.com/jtpereyda/boofuzz (accessed on 18 June 2023).
39. ACM CCS 2021. Available online: https://www.sigsac.org/ccs/CCS2021/ (accessed on 18 June 2023).
40. Sulley: A Pure-Python Fully Automated and Unattended Fuzzing Framework. Available online: https://github.com/OpenRCE/sulley (accessed on 18 June 2023).
41. KittyFuzzer: Fuzzing Framework Written in Python. Available online: https://github.com/cisco-sas/kitty (accessed on 18 June 2023).
42. Zhang, H.; Lu, K.; Zhou, X.; Yin, Q.; Wang, P.; Yue, T. SIoTFuzzer: Fuzzing Web Interface in IoT Firmware via Stateful Message Generation. *Appl. Sci.* **2021**, *11*, 3120. [CrossRef]
43. Demšar, J. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* **2006**, *7*, 1–30.
44. Krichen, M. Improving formal verification and testing techniques for internet of things and smart cities. *Mob. Netw. Appl.* **2019**, *2019*, 1–12. [CrossRef]
45. Fortas, A.; Kerkouche, E.; Chaoui, A. Formal verification of IoT applications using rewriting logic: An MDE-based approach. *Sci. Comput. Program.* **2022**, *222*, 102859. [CrossRef]