applied sciences

MDPI

Article

On the Analysis of Coverage Feedback in a Fuzzing Proprietary System

Daehee Jang [1], Jaemin Kim [2,3], Jiho Kim [4], Woohyeop Im [1], Minwoo Jeong [1], Byeongcheol Choi [5] and Chongkyung Kil [2,*]

1 School of Computing, Kyung Hee University, Yongin 17104, Republic of Korea; daehee87@khu.ac.kr (D.J.); pwn@khu.ac.kr (W.I.) p1nkjelly@khu.ac.kr (M.J.)
2 CW Research Inc., Shrewsbury SY2 6BL, UK; woaalsdl12@gmail.com
3 Department of Computer and Information Security, Sejong University, Seoul 05006, Republic of Korea
4 College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA; jkim4050@gatech.edu
5 ETRI, Daejeon 34129, Republic of Korea; corea@etri.re.kr
* Correspondence: contact@cwresearchlab.co.kr

Abstract: Coverage feedback is one of the key mechanisms for improving the effectiveness of fuzzers by measuring and comparing the executed code regions while processing input data. In general, such guidance should always improve the performance of fuzzers to better find unexplored code regions. However, proprietary systems with uncommon I/O interfaces (e.g., UAV system, IoT devices, satellite firmware) require extensive engineering/porting efforts to apply coverage feedback support in developing their fuzzing platform. In this paper, we evaluate the detailed efficacy of *coverage feedback* in fuzzing based on 44 real-world bugs we found using OSS-Fuzz. Our analysis uncovered when and how code coverage information can be helpful, and our experiment demonstrates that although coverage guidance is always helpful to some extent, its effectiveness depends on various external factors. Therefore, such factors should be carefully considered for optimizing the cost and efficiency in designing the fuzzing architecture of proprietary systems.

Keywords: fuzzing; coverage feedback; bug detection

Citation: Jang, D.; Kim, J.; Kim, J.; Im, W.; Jeong, M.; Choi, B.; Kil, C.K. On the Analysis of Coverage Feedback in a Fuzzing Proprietary System. *Appl. Sci.* **2024**, *14*, 5939. https://doi.org/10.3390/app14135939

Academic Editors: Shanling Dong, Meiqin Liu and Yutaka Ishibashi

Received: 28 May 2024
Revised: 28 June 2024
Accepted: 5 July 2024
Published: 8 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Fuzzing automatically generates test data and executes the target application (data parser) repeatedly until an error is detected. Because the idea of fuzzing was first published in 1990 by Miller et al. [1], it quickly became one of the most universal methods for ensuring software reliability and finding bugs in a wide range of software [2]. For example, OSS-Fuzz [3], Google's open-source fuzzing platform, contributed to the discovery and patching of over 10,000 vulnerabilities and 36,000 bugs in approximately 1000 projects.

As fuzzing has been well proven to be effective in practice, researchers have placed significant efforts into enhancing efficacy/performance with various ideas such as combining symbolic execution or tainting techniques with fuzzing algorithms. In the last decade, numerous works [4,5] have suggested state-of-the-art fuzzing methods to enhance the fuzzer's capability to find bugs.

One of the standard approaches for evaluating new fuzzers is by using the *code coverage* as an evaluation metric. Some previous work [6,7] has been solely focused on providing a framework for comparing and evaluating fuzzers. Recently, however, researchers started questioning this evaluation approach [8,9]. The question is: "does achieving better code coverage always guarantee better fuzzing results?". To quote the paper [9]: *"fuzzer best at achieving coverage, may not be best at finding bugs"*. These recent works suggest that achieving maximum coverage with feedback is helpful in general, but it should not be the only metric for evaluating the efficacy of finding bugs.

In general, it is certain that expanding code coverage with feedback is beneficial for fuzzing. However, if such a benefit requires extensive engineering cost, we need to analyze and compare the detailed efficacy relative to the cost. Such costs in fuzzing are especially high in proprietary systems. It is noteworthy that researchers recently started to aggressively apply fuzzers to proprietary systems such as IoT devices, drones, cars, satellite firmware, and various embedded systems [10–12]. These recent works suggest that the porting costs of applying coverage feedback in fuzzing such unique systems is costly and challenging.

To address such recent fuzzing research issues, we quantify and measure the detailed costs of applying coverage feedback in fuzzing and its efficacy for optimizing system design. Our main hypothesis is that the efficacy of coverage guidance in fuzzing will substantially differ based on various factors such as the initially given sample data, type of application parsers, type of bug, and so on. While it is easy to write simple code snippets that highlight the obvious benefits of coverage feedback, our goal is to investigate the efficacy of coverage guidance for various types of fuzzers that have found real-world bugs and reveal what the important factors are in coverage feedback. The goal of our research is to determine and justify when and why we need to apply (or not apply) the coverage feedback feature when fuzzing proprietary systems based on closed sources.

To address the aforementioned research issue, we used 44 unique real-world bugs that we discovered over a year using the OSS-Fuzz and ClusterFuzz [13] fuzzing infrastructures. With this ground-truth dataset, we iterated finding such previous bugs with various configurations regarding coverage guidance: (i) no coverage feedback, (ii) partial coverage feedback, and (iii) full coverage feedback. With this experiment, we were able to determine the relationship between efficacy of coverage guidance and multiple factors. Note that the motivation for our work stems from assessing the efficiency of coverage feedback in closed-source software. However, we used open-source software for our evaluation as an alternative because acquiring a sufficient amount of closed-source software and finding dozens of bugs for evaluation is very challenging in practice. Although the evaluation target is based on open-source software, we believe that the stochastic tendencies we find and analyze in our evaluation should be equally valuable.

In short, our study revealed that the quality of initial seed (initially given the test case data for mutation) has a strong correlation to coverage feedback efficacy. When adequate initial seeds were provided, the efficiency in bug reproduction showed minimal performance variation regardless of whether coverage feedback is given or not. Such an observation suggests that investing in quality initial seed data might sometimes be more beneficial than solely on enhancing coverage feedback with significant porting/engineering efforts in fuzzer development. For the efficient fuzzing of proprietary systems, we emphasize the importance of balanced resource allocation between developing sophisticated coverage feedback mechanisms and generating comprehensive initial test cases.

2. Background

2.1. Code Coverage and Fuzzing

Code coverage encompasses various metrics, including function coverage, statement coverage, branch coverage, condition coverage, and line coverage [14]. These metrics differ in the choice of the unit for measuring coverage. For example, line coverage treats each line as a unit (based on source code), whereas branch or block coverage examines the number of executed branches or blocks in the assembly or binary level. In software testing, code coverage is measured via crafted test cases including data and code altogether (e.g., unit test) to find errors in the software development phase [15]. In fuzzing, code coverage is measured by feeding the target code a randomly generated (mutated) test input [16]. However, according to recent studies [8,9], achieving 100% code coverage does not always guarantee testing quality nor the finding of bugs. In this paper, we figure out more details to this end, considering the system porting costs for applying coverage feedback fuzzing to closed-source systems.

2.2. Fuzzing Proprietary System

Fuzzing proprietary systems has many challenges compared to open-source-based fuzzing. Open source allows fuzzers to recompile and instrument the target code for measuring various features in their fuzzing process, thus enhancing performance. However, most proprietary applications, such as IoT and UAV firmware, cannot be recompiled and instrumented. One way to apply coverage feedback in such closed-source applications is by using an emulator [17] such as QEMU. Unfortunately, only a few proprietary systems can be emulated in reality. Even worse, some of these firmware are cryptographically signed and encrypted, thus preventing them from being modified, even at the binary or assembly level. Our research can be a reference for designing fuzzers for such systems.

2.3. Coverage Feedback in `libFuzzer`

libFuzzer is one of the most successful coverage-guided fuzzers that is incorporated into compiler frameworks [18]. While libFuzzer is a powerful fuzzing tool, it requires a complete source code that is compliant with specific versions of Clang/GCC compilers, thus making it only available for open-source software that provides specific build configuration based on generic compilers. Algorithm 1 is the simplified libFuzzer's internal logic.

Algorithm 1: libFuzzer Core Fuzzing Algorithm

```
initialize;
while NumberOfRuns < Options.MaxRuns do
    choose unit to mutate from corpus;
    for i ← 1 to Options.MutationDepth do
        if NumberOfRuns >= Options.MaxRuns then
            break;
        end
        mutate;
        run with mutated input;
        if NewCoverage then
            add mutated input to corpus;
            break;
        end
    end
end
```

In libFuzzer's core algorithm, an input unit is selected from the corpus (a set of input data) for random mutation. The mutated test input is then tested against the target code until new coverage has been found or the maximum mutation depth is reached (`libFuzzer`'s fuzzing also halts upon bug detection). If the mutated input reports new coverage, it is considered interesting and is added to the corpus so that it can be potentially mutated again afterwards. In our paper, we use libFuzzer as the baseline evaluation tool for our investigation. We mainly modify libFuzzer in two ways (ignoring coverage feedback and increasing the granularity of the coverage guidance feature).

2.4. Utilizing OSS-Fuzz Project

OSS-Fuzz is a collection of more than 1000 open-source-based fuzzing projects initiated by Google. In numerous academic studies, OSS-Fuzz serves as an experimental baseline for evaluating the efficacy of coverage feedback-based fuzzing. Researchers can utilize OSS-Fuzz to conduct experiments on a variety of open-source fuzzing against real-world software (mostly libraries). In this paper, we utilize libFuzzer stubs in OSS-Fuzz that have actually found real-world bugs to justify our claims in research experiments.

3. Materials and Methods

For the experiment environment setup, we use the Clang compiler from LLVM-15 (git commit hash bf7f8d6fa6f460bf0a16ffec319cd71592216bf4). To set up the experiment environment, we need to understand the internals of libFuzzer's code coverage feedback algorithm and modify them properly for comparison. We use Docker for an automated/fast experiment environment controlled with approximately 1000 lines of Python scripts to manage the experiment system. Figure 1 illustrates the overall architecture design of our experiment system.

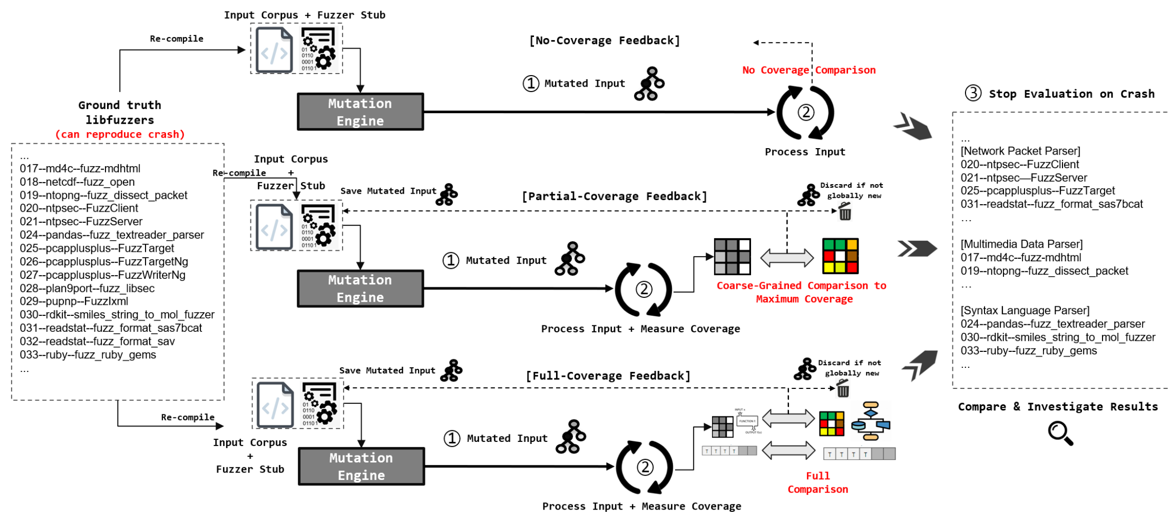

Figure 1. Overall workflow for the experimentation environment setup and analysis. First, we ran OSS-Fuzz for 2 years and found 44 unique crashes based on libFuzzer. Second, we recompiled all libFuzzer source codes based on our customized Clang (no-coverage, partial-coverage, full-coverage) for comparison. Finally, we ran recompiled fuzzers until they reproduced previous crashes and compared/analyzed the detailed contribution of the coverage feedback feature for each application type.

The workflow in our experiment begins with gathering unique real-world bugs based on OSS-Fuzz fuzzers. We have set up our own fuzzing infrastructure/management system based on ClusterFuzz (Although OSS-Fuzz and ClusterFuzz are open-source, crash discovery results found using this infrastructure are undisclosed. Therefore, we used our private infrastructure based on a modified version of ClusterFuzz and gathered unique bugs in OSS-Fuzz for years) and utilized it to find 44 unique memory bugs in OSS-Fuzz projects based on libFuzzer as the fuzzing engine. Following this, a crucial step in our experiment is to recompile the libFuzzer stubs responsible for finding such bugs based on our customized Clang compiler. The Clang compiler customization allows us to change the code coverage feedback mechanism provided by `libFuzzer` (`libFuzzer` is implemented as a part of Clang compiler runtime): (i) no coverage feedback, (ii) partial coverage feedback, and (iii) full coverage feedback. If we cross-reference the results based on the three different versions of fuzzers, we can find the tendency and relationship between the effectiveness of coverage guidance in detail.

More specifically, we run each fuzzer until they replicate the previously identified bugs and measure the *consumed execution count* for this reproduction. Based on this measurement, we analyze the detailed impact and outcomes of fuzzing with different levels of coverage feedback. Through this process, we cross-reference and analyze how the coverage feedback mechanism influences the effectiveness of fuzzing across various factors, including the type of application parser, initial seed data, and so forth. This examination should provide a better understanding and justification of applying coverage feedback for fuzzing proprietary systems that require high porting costs and engineering efforts for integrating such mechanisms.

3.1. Modification to `libFuzzer` *for the Experiment*

We modify and prepare three versions of the Clang compiler for the experiment: (i) the no-coverage version, (ii) the partial coverage version, and (iii) the full coverage version (the original one). Modifying to totally exclude code coverage feedback in `libFuzzer`'s implementation (for the no-coverage version) seems to be a simple task; however, there are a few things to consider. For example, simply ignoring the coverage feedback information for deciding whether to keep or discard the mutated input raises some problems. If we always choose to keep the mutated input by skipping the coverage feedback operations, the corpus set will continuously increase over time, thus quickly exhausting disk/memory space and slowing down the entire fuzzer's performance until it crashes due to an out-of-memory error. Thus, this decision is improper for our experiment design. Also, we need to be careful of compiler optimization erasing important instructions unrelated to coverage feedback. Our modified libFuzzer performs input mutation but always discards the test case after finishing each execution, which is a simple standard implementation in non-coverage feedback fuzzers. Figure 2 shows the relevant code patch (we mainly modified the `RunOne()` function of libFuzzer to only exclude the coverage feedback during the source code modification. We checked the recompiled fuzzer binary considering the possibility of compiler optimization erasing important features other than the feedback). And the result of the modified libFuzzer binary was used to ignore the feedback. Additionally, we note that the `no-coverage` version of our libFuzzer does not eliminate the existing code for measuring code coverage in the fuzzing round. Instead, the modified version ignores the inspected coverage information and processes every input as if it does not increase the overall coverage. Because of this, the execution time of a single fuzzing round for both the original fuzzer and the altered fuzzer would not exhibit a measurable difference in timing performance. However, when a fuzzer discovers new coverage, it stores the in-memory test data to external storage (disk), which involves a significant timing delay. Unfortunately, predicting the pattern and occurrence of such delays is challenging, so we use the execution counts of the fuzzer as a comparison metric in our evaluation.

```
49 compiler-rt/lib/fuzzer/FuzzerLoop.cpp
@@ -505,16 +505,61 @@ static void WriteEdgeToMutationGraphFile(const std::string
505 505
506 506    bool Fuzzer::RunOne(const uint8_t *Data, size_t Size, bool MayDeleteFile,
507 507                        InputInfo *II, bool ForceAddToCorpus,
508     -                   bool *FoundUniqFeatures) {
    508 +                   bool *FoundUniqFeatures, bool seed) {
509 509      if (!Size)
510 510        return false;
511 511      // Largest input length should be INT_MAX.
512 512      assert(Size < std::numeric_limits<uint32_t>::max());
513 513
514 514      ExecuteCallback(Data, Size);
    515 +    auto TimeOfUnit = duration_cast<microseconds>(UnitStopTime - UnitStartTime);
515 516
516 517      UniqFeatureSetTmp.clear();
    518 +
    519 +    if(!seed){
    520 +      PrintPulseAndReportSlowInput(Data, Size);
    521 +      return false;
    522 +    }
    523 +    Printf("[CWFuzz][no_coverage] Attempting to add seed to corpus \n");
```

Figure 2. The key patch for `libFuzzer` to optionally discard coverage feedback. We modified `RunOne` and its dependent code appropriately, considering compiler optimization, and confirmed the resulting binary to double-check correctness.

In case of partially excluding coverage feedback, we need to investigate the detailed implementation of the libFuzzer coverage feedback engine. In our customization, we gather partial coverage by changing the *granularity* of the coverage measurement rather than selectively keeping/discarding the coverage information. To explain this in detail, we first need to discuss details of libFuzzer's coverage measuring system.

3.2. Details of `libFuzzer's` Internal and Customization for Partial Coverage

In libFuzzer, code coverage information is specifically referred to as `Feature`, which is a libFuzzer-specific terminology for measuring code coverage. libFuzzer converts code coverage information into multiple *features*, which include edge coverage, edge counters, value profile, indirect caller/callee pairs, etc. [18]. Each feature is represented as an unsigned 32-bit integer value based on an 8-bit counter array representing edges (possible execution branches between code). For partial coverage measurement customization, we change the granularity of the counter size to lower the precision/accuracy of the measured coverage. Specifically, we modify the 8-bit array counter for feature collection to 3 bits using `counter buckets`. Counters falling into each bucket `1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+` are mapped into corresponding integers ranging from 0 to 7. Also, we change the way libFuzzer handles input via the *ReducedInput* option. The ReducedInput option allows for a new input to replace its parent input in case both inputs exhibit the same code coverage. In partial code coverage customization for lowering the precision, we discard the mutated test case with the same code coverage, even if it is shorter than its parent test case. These modifications give us a customized version of libFuzzer with less coverage feedback capability. We use such an intentionally poor-performing version of libFuzzer for comparison in our experiment.

3.3. Experiment Methodology and Metric

In our evaluation, all of the recompiled libFuzzer harnesses should eventually reproduce crashes at some point in theory. However, we cannot estimate how much time the lack of coverage feedback might additionally consume for reproducing the bug. Therefore, we need some baseline timeout policy for running the customized fuzzers. This maximum can be restricted based on (i) time and (ii) execution count. For example, when running the customized fuzzers, we can limit the maximum execution time for 48H, or similarly, we can limit the count of execution to 10 million per se. This is also an experiment design decision we need to examine. If we choose time as the limit, the comparison experiment could be unfair because excluding/reducing the quality of coverage feedback could make the fuzzer run faster, thus increasing the likelihood of reproducing the bug faster than the original fuzzer. Therefore, we choose *maximum count of execution* for the comparison metric in our evaluation. In a nutshell, we execute original/customized libFuzzer harnesses until they reproduce the crash (or until they reach the limit execution count) and compare their consumed execution counts as the key metric for fuzzing efficiency.

4. Results

In this section, we explain and analyze the detailed results of our experiment. For the experiment, we first gathered 44 unique memory bugs by running OSS-Fuzz fuzzers (libFuzzer harnesses) in our private infrastructure similar to ClusterFuzz. Once we found such bugs, we utilized them as the ground-truth source for our evaluation. To evaluate the efficacy of coverage feedback in reproducing such bugs in 44 libFuzzer stubs, we modified libFuzzer's behavior by modifying the LLVM-15 source code. For our evaluation platform, we use an `Ubuntu 22.04 64-bit server` operating system based on an `AMD Ryzen-9 5900` CPU architecture, which supports up to 24 cores. Also, we note that we use the `-workers=5` option (five parallel evaluation) for all libFuzzer harness testing and iterate the evaluation twice to calculate the average.

Table 1 is the summarized result of our evaluation. The `Default` column indicates the fuzzer execution count for reproducing the same bug using the original libFuzzer.

`Partial` indicates the fuzzer execution count for reproducing the same bug based on the partially modified (using coarse-grained coverage feedback information) libFuzzer. `No Cov` indicates the fuzzer execution count for reproducing the same bug with the modified libFuzzer for discarding the coverage feedback information. In the `No Cov` case, the number of fuzzer execution counts is omitted if bug reproduction fails even after running the fuzzer for a sufficiently long time (48 h) and also executing the fuzzing iteration at least twice the original count for discovery. The `Seed` column indicates the existence of the initially given test data set for mutation in fuzzing. If no initial seed is given, libFuzzer automatically generates a series of random byte sequences for its test input data.

Table 1. Evaluation results for using 44 real-world bugs found with libFuzzer. We have reproduced each bug with modified versions of Clang with regard to coverage feedback. `Default`, `Partial`, and `No Cov` columns each indicate running the original/customized version of libFuzzer that measures coverage information with coarse granularity/customized libFuzzer, which does not provide coverage feedback. Each number represents the average execution counts required to reproduce the bug. `Seed` indicates whether the fuzzer has been provided a proper sample test case at the beginning of fuzzing.

No	Project Name	Default	Partial	No Cov	Seed	Bug Type
1	coturn/FuzzStun	33	43	60	Y	Heap BOF
2	example/do_stuff_fuzzer	497	10 K	33 K	Y	Heap BOF
3	hiredis/format_command_fuzzer	4 K	2 K	5 K	-	Heap BOF
4	librdkafka/fuzz_regex	522 K	2 M	-	-	Heap BOF
5	libredwg/llvmfuzz	256 K	279 K	-	-	Heap BOF
6	libyaml/libyaml_dumper_fuzzer	4 M	10 M	-	Y	Heap BOF
7	llvm/llvm-special-case-list-fuzzer	28 M	-	-	-	Heap BOF
8	lzo/lzo_decompress_target	162 K	134 K	196 K	Y	Heap BOF
9	ntpsec/FuzzServer	2 K	6 K	6 K	Y	Heap BOF
10	open62541/fuzz_mdns_message	33 K	11 K	32 K	Y	Heap BOF
11	openbabel/fuzz_obconversion_smiles	402	1 K	701	-	Heap BOF
12	plan9port/fuzz_libsec	46	72	56	-	Heap BOF
13	pupnp/FuzzIxml	33 K	15 K	1 K	Y	Heap BOF
14	readstat/fuzz_format_sas7bcat	285 K	580 K	13 M	Y	Heap BOF
15	readstat/fuzz_format_sav	5 M	6 M	-	Y	Heap BOF
16	ruby/fuzz_ruby_gems	3 M	3 M	-	-	Heap BOF
17	serenity/FuzzILBMLoader	5 M	5 M	-	-	Heap BOF
18	simd/simd_load_fuzzer	29 K	52 K	-	-	Heap BOF
19	vlc/vlc-demux-dec-libFuzzer	262	263	186	-	Heap BOF
20	vulkan-loader/json_load_fuzzer	54 K	84 K	127 K	Y	Heap BOF
21	c-blosc2/decompress_chunk_fuzzer	451 M	67 M	-	Y	Negative Size Param
22	fluent-bit/cmetrics_decode_fuzz	3 K	1 K	4 K	-	Invalid Free
23	vulkan-loader/instance_create_fuzzer	20 K	5 K	4 K	Y	Invalid Free
24	vulkan-loader/instance_enumerate_fuzzer	19 K	3 K	9 K	Y	Invalid Free
25	augeas/augeas_fa_fuzzer	2 K	183	3 K	-	SEGV
26	bloaty/fuzz_target	2 M	2 M	3 M	Y	SEGV
27	fluent-bit/cmetrics_decode_fuzz	731 M	9739 M	-	-	SEGV
28	glog/fuzz_demangle	20 K	15 K	-	-	SEGV
29	haproxy/fuzz_cfg_parser	507 K	616 K	-	-	SEGV
30	hiredis/format_command_fuzzer	12 K	2 K	215 K	-	Heap BOF
31	ibmswtpm2/fuzz_tpm_server	9 K	11 K	2 M	-	SEGV
32	libbpf/bpf-object-fuzzer	120 K	94 K	119 K	Y	SEGV
33	libical/libical_extended_fuzzer	623 K	1 M	-	-	SEGV
34	libredwg/llvmfuzz	561 K	159 K	-	-	SEGV
35	oatpp/fuzz_mapper	7 M	5 M	-	-	SEGV
36	pcapplusplus/FuzzTargetNg	187	128	145	Y	SEGV
37	php/php-fuzz-execute	924 K	625 K	2 M	Y	SEGV
38	php/php-fuzz-function-jit	32 K	42 K	66 K	Y	SEGV
39	wabt/read_binary_interp_fuzzer	353 K	212 K	-	-	SEGV
40	wabt/read_binary_ir_fuzzer	60 M	51 M	-	-	SEGV
41	augeas/augeas_api_fuzzer	67 K	70 K	-	-	UAF
42	augeas/augeas_escape_name_fuzzer	1 M	1 M	-	-	UAF
43	c-blosc2/decompress_frame_fuzzer	168 K	121 K	-	Y	Heap BOF
44	cups/FuzzCUPS	15 K	14 K	21 K	Y	Heap BOF

The evaluation table shows a discernible relationship between the `No Cov` fuzzer's result and the `Seed`. It is noteworthy that the initial seed is manually crafted by the open-source developers and thus contains crucial information regarding data structure and format. The evaluation results based on our test data set suggest that the influence of the initial seed is significantly greater than the detailed mechanism of the coverage feedback

feature. A total of 15 out of 20 cases of bug reproduction failures in the no-coverage experiment lacked initial seed data. However, in cases where a proper initial seed was given (e.g., highlighted with red color in the table), the difference in performance for reproducing the bug was relatively insignificant (typically, all cases found the same bug within the range of x2 to x3 fuzzing execution attempts). On the other hand, in cases where no initial seed was given, the performance of the no-coverage fuzzer was significantly degraded (e.g., highlighted with yellow color in the table). The stochastic tendency in our evaluation table indicates that it would be better to investigate the efforts (engineering, manpower) for providing proper initial seeds for test cases rather than enhancing/adopting better coverage feedback features in some extreme cases.

Other factors, such as the type of bug and data parser category, did not affect the evaluation data in a significant way. Nevertheless, we have investigated such issues as well and provide related information in Appendix A. Interestingly, in some cases, coarse granularity coverage feedback fuzzing showed better efficiency (finding the same bug more quickly) compared with the original. The reason for such a phenomenon might be explained with some example cases that we analyze in Section 5.

Overall, it is evident that coverage feedback enhances the performance of fuzzing and thus increases the likelihood of discovering unknown bugs in a more efficient way. However, our study suggests that if the target system (such as a closed-source proprietary application) requires significant efforts and costs for applying the coverage feedback feature, it would be better to invest the limited resources in other perspectives such as providing high-quality initial test cases that include various structures and logical semantics of the application with static analysis.

5. Discussion

In this section, we discuss special cases where feedback information could be extremely beneficial for finding new code coverage and opposite cases where feedback would be useless for expanding the code coverage. We note that the example cases we discussed in this section are special edge cases to help analyze and interpret our evaluation. However, we note that such cases are relatively common as we can quickly find it from real-world open-source repositories.

5.1. Example Case Where Coverage Feedback Is Beneficial

In Listing 1, a parser code snippet in one of the OSS-Fuzz projects (`ntpsec-secure network time protocol implementation`) demonstrates taking an array of 16-bit array data from the client as a potentially malicious input. In this example, each value in 16 bits of data determines which code to execute next. As 16 bits is a relatively small bit space to explore based on randomness, the fuzzer will quickly test all possible 65,536 cases and find new functions for additional data processing. The new function (e.g., `nts_next_protocol()`) triggered by specific data in this case subsequently processes the same input orthogonal to previously processed data (e.g., bytes of the `buff` array from the previous loop iterations). Therefore, in this case, a randomly discovered test case (that happens to trigger a new function) for invoking such a function is valuable to remember (the genetic algorithm in fuzzing applies more mutation attempts in the future to interesting inputs). Therefore, feedbacking and storing this special test case for future input mutation is beneficial for finding additional code coverage.

Listing 1. An example case where feedback information can effectively expand code coverage (`ntpsec/FuzzClient`).

```
1 #include "libntp.h"
2 #include "ntp.h"
3 #include "ntp_proto.h"
4 ...
5 bool fuzz(uint8_t *buff, int size) {
6     //While the buff has more than 4 bytes, there is data to process
7     while (size >= 4) {
```

```
        uint16_t type = (uint16_t *)buff;
        buff += 2;
        int length = (uint16_t *)buff;
        buff += 2;
        size -= 4;

        switch (type) {
            case nts_next_protocol_negotiation:
                nego_next_protocol(&buff, &size);
                break;
            case nts_algorithm_negotiation:
                nego_algorithm(&buff, &size);
                break;
            case nts_new_cookie:
                new_cookie(&buff, &size);
                break;
            ...
            default:
                //skip current data
                buff += length;
                size -= length;
                break;
        }
    }

    return 0;
}
```

5.2. Example Case Where Fine-Grained Coverage Feedback Is Ineffective

Listing 2 demonstrates a detailed example case in `Apache-APR` from the OSS-Fuzz project, where coverage expansion based on fine-grained (counting the execution attempt of the same code location and considering a greater execution count as a new code coverage) feedback is useless. For example, the `SKIP_WHITESPACE` macro implementation is responsible for counting white space characters from the given input byte stream. In this implementation, libFuzzer will provide feedback as a new coverage for every white-space character appended to the mutated input, as the block execution count will increase proportionate to the number of white-space characters. However, this feedback and coverage expansion do not actually contribute to finding a meaningful code path at all. However, the fuzzer will keep accumulating such mutated test cases because of the coverage feedback result. This could hinder the fuzzer's efficiency and might explain some of the evaluation results in Table 1 where coarse-grained feedback demonstrated better performance.

Listing 2. Example code snippet from the Apache-APR implementation demonstrating useless coverage finding due to inline-counter feedback.

```
#include "apr.h"
#include "apr_strings.h"
#include "apr_private.h"
#include "apr_lib.h"
...

#define SKIP_WHITESPACE(cp) \
    for ( ; *cp == ' ' || *cp == '\t'; ) { \
        cp++; \
    };

#define CHECK_QUOTATION(cp,isquoted) \
    isquoted = 0; \
    if (*cp == '"') { \
        isquoted = 1; \
        cp++; \
    } \
    else if (*cp == '\'') { \
        isquoted = 2; \
```

```
20        cp++; \
21    }
22 ...
```

5.3. Example Case Where Coverage Feedback Is Meaningless

Listing 3 represents an example implementation for checking the integrity of the PNG image header. In this case, checking the code coverage feedback might be totally useless considering that the probability of the fuzzer randomly matching the PNG signature or CRC checksum is practically zero. This is dependent on how the `memcmp` or integer value comparison is implemented. More specifically, more coverage can be measured if the comparison takes different execution cycles upon different input data although their byte length is the same. If the comparison is atomic (e.g., executed instruction sequence for entire comparison is identical even if two given data for comparison partially matches or not), the probability of the fuzzer discovering a new code coverage is 1 over 2 to the 64th in the case of the PNG signature match and 1 over 2 to the 32nd for the CRC check. To handle special cases such as file signature match, libFuzzer typically maintains `dictionary` or supplementary data structure such as the `Table of Recently Compared data (TORC)` to discover such special data, regardless of coverage feedback. Building a good dictionary is more important than coverage feedback in such cases. In the case of a CRC integrity check example, there is no feasible way for the fuzzer to automatically solve the condition solely based on coverage feedback. On the other hand, one may use more detailed and fine-grained program behavior measurement techniques. For example, `cmplog` of AFL++ [19] and `Sanitizer Coverage` in libFuzzer provide fine-grained measurements that allow for the detailed assessment of processed data. With such advanced fuzzing support features, it could be possible to automatically solve such conditions in a small amount of time based on implementation details.

Listing 3. Example implementation of PNG header integrity checking. Coverage feedback information does not help the fuzzer find a new code path (integrity check success) in such a case.

```
1  #include "png.h"
2  #include "crc.h"
3  ...
4
5      // Check the PNG signature
6      const uint8_t png_signature[8] = {137, 80, 78, 71, 13, 10, 26, 10};
7      if (memcmp(png_data, png_signature, 8) != 0) {
8          // Not a PNG file
9          return -1;
10     }
11
12     // next 17 bytes to header (IHDR chunk bytes).
13     uint32_t file_crc = *(uint32_t *)(png_data + 28);
14     uint32_t memory_crc = calculate_crc32(png_data + 8, 17);
15
16     if (file_crc != memory_crc) {
17         // Integrity Check Failed
18         return -1;
19     }
20
21     // parser continues...
```

6. Limitations and Future Work

In this paper, we have investigated the efficacy of coverage feedback and fuzzing results to provide a guideline for assessing the cost-effectiveness of applying coverage feedback features to closed-source systems such as proprietary firmware in IoT or UAV devices. In industry, many closed-source UAV systems, such as the DJI drone, have their dedicated SoC hardware, which requires an extensive engineering effort to apply fuzzing to [20]. Researchers trying to apply fuzzing to such systems struggle to receive feedback, such as code coverage increments and crash events. Although there are a few ways to measure code

coverage without utilizing source code via emulation or debugging features [21], applying such techniques to other proprietary systems remains challenging. Despite the evaluation and analysis conducted in this paper, quantifying the efficacy of fuzzing and the cost of altering a system (e.g., to support coverage-guided fuzzing) cannot be performed in a straightforward manner, as various factors must be considered, including CPU architecture support, operating system support, hardware interfaces, and so forth. Considering all of these factors to determine the cost of applying coverage-guided fuzzing would be a challenging future task. Although we do not provide a straightforward answer regarding the efficacy of fuzzing and the cost of coverage measurement, we hope our analysis and evaluation results can aid future research directions and be utilized as a ground-truth reference for this purpose.

7. Related Works

7.1. Analyzing Coverage Metrics

Wang et al. introduced the concept of *sensitivity* in coverage-guided fuzzing, which can be utilized to compare different coverage metrics in fuzzing [22]. This previous research systematically examined the impact of various coverage metrics on fuzzing performance. Using a formally defined concept of sensitivity, their research evaluated a wide range of metrics, including basic branch coverage, context-sensitive branch coverage, and memory-access-aware branch coverage, with the AFL tool on extensive datasets. The evaluation results indicate that more sensitive coverage metrics generally lead to quicker and more frequent bug discoveries. Additionally, different metrics tend to find distinct sets of bugs, and the optimal metric may vary at different stages of the fuzzing process. The main difference between such previous works and ours is that we intend to investigate the cost versus gain of utilizing coverage feedback in fuzzing in terms of closed-source systems, which impose significant engineering challenges for developing coverage-guided fuzzers.

7.2. Fuzzing Closed Systems

As fuzzing became a popular and standard way for discovering memory errors in software, researchers started to adopt fuzzing to special systems such as closed-source embedded systems (e.g., UAV firmware, satellite firmware, IoT device firmware, etc.). Applying fuzzing to such systems raises significant challenges in measuring code coverage due to the impossibility of source-code instrumentation. Measuring code coverage for binary-only applications is feasible based on QEMU-AFL [23]; however, applying such fuzzers still requires a compatible operating system and CPU architecture (e.g., standard Linux and Intel/ARM architecture). Special systems such as satellite firmware are usually based on the SPARC Leon-3 [24] CPU architecture with the RTEMS [25] real-time operating system or VxWorks [26] systems. UAV firmwares such as the DJI drone firmware use proprietary systems with multiple SoC environments [20]. Moreover, such firmwares are often signed/encrypted to prevent any modification at the binary level [27]. Applying fuzzing techniques to such systems is feasible; however, supporting coverage feedback as in popular open-source-based fuzzers requires massive system porting and engineering. Jang et al. [10] recently proposed a system for applying coverage feedback-based fuzzers to embedded firmwares and partially recovered code fragments. In the study, the authors defined a concept called *fuzzability* to test if a fragment of binary could be fuzzed via conventional feedback-based fuzzers. However, their evaluation suggests that applying coverage feedback-based fuzzing to such systems/environments is highly costly (in terms of system engineering) and inefficient. This paper addresses such research problems and conducts an extensive fuzzing evaluation based on 92 real-world bugs found by OSS-Fuzz.

7.3. Previous Fuzzing Works

Recent advancements in fuzzing have significantly improved the efficiency and effectiveness of finding bugs. This section summarizes related works regarding various fuzzing approaches, highlighting key contributions and their methodologies.

Coverage-based fuzzing is a standard approach where the fuzzer aims to maximize the coverage of code paths during execution. The key idea in coverage fuzzing is using genetic algorithms to selectively preserve mutated inputs based on their contribution to expanding code coverage, thus significantly increasing the likelihood of discovering new code paths. To this end, various researchers have suggested novel approaches. Böhme et al. [9] discuss the reliability of coverage-based fuzzer benchmarking, emphasizing the need for consistent and reliable benchmarks in evaluating fuzzers. Similarly, ref. [28] introduced coverage-based greybox fuzzing as a Markov chain, laying the groundwork for other subsequent fuzzing techniques.

Application-aware fuzzing techniques, such as `VUzzer` [5], adapt fuzzing strategies based on application-specific knowledge, leading to a more effective identification of vulnerabilities. `CollAFL` [29] and `FairFuzz` [30] propose path-sensitive and targeted mutation strategies, respectively, to enhance coverage and effectiveness. `Hawkeye` [31] focuses on specific program paths or vulnerabilities, directing the fuzzer's efforts toward areas of interest. `JITfuzz` [32] specifically targets Just-in-Time compilers in JVMs, highlighting the applicability of fuzzing in diverse software environments.

Integrating fuzzing with other analysis techniques has also shown promise. `SAFL` [33] combines symbolic execution with guided fuzzing to improve testing coverage, while `Zeror` [34] accelerates fuzzing with coverage-sensitive tracing and scheduling. Hardware-assisted fuzzing, as demonstrated by kAFL [35], leverages hardware features to improve the feedback mechanism and efficiency of fuzzing kernels and operating systems. Similarly, though based on software-only techniques, full-speed fuzzing [21] aims to reduce overhead through coverage-guided tracing, enhancing the speed and efficiency of fuzzing processes based on the debug interrupt feature.

Network protocol-specific fuzzing has also gained attention with tools like `AFLNET` [36], which adapts greybox fuzzing for network protocols, and `Fw-fuzz` [37], focusing on firmware network protocols. These tools address the unique challenges of fuzzing network interactions and embedded systems. For IoT devices and firmwares, `FIRM-COV` [38] introduces a high-coverage greybox fuzzing framework optimized for IoT firmware through process emulation. This paper explores the effectiveness of coverage feedback in fuzzing considering their expensive porting cost in IoT and embedded firmware systems.

8. Conclusions

This paper uncovers the details of how and when code coverage feedback is beneficial or not. Our research revealed that while coverage-guided fuzzing benefits most scenarios to some extent, its effectiveness is substantially influenced by multiple factors such as the *existence of an initial seed corpus*, type of data parser, and amount of code base (line of code). Such information is especially valuable in situations requiring substantial effort to implement coverage feedback fuzzing, such as with closed-source or embedded system firmwares. By evaluating 44 real-world bugs from OSS-Fuzz, we found detailed correlations between coverage feedback and external factors. We hope researchers can reference our findings in designing fuzzing systems for closed-source applications, such as UAV/IoT firmware.

Author Contributions: Conceptualization, D.J. and C.K.; methodology, J.K. (Jaemin Kim); software, J.K. (Jiho Kim); validation, W.I., M.J. and B.C.; writing—original draft preparation, D.J.; writing—review and editing, D.J.; visualization, J.K. (Jamin Kim); supervision, C.K.; project administration, D.J.; funding acquisition, B.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Convergence Security Core Talent Training Business Support Program grant number IITP-2024-RS-2023-00266615, and the APC was funded by the KRIT (Korea Research Institute for defense Technology planning and advancement) grant funded by the Defense Acquisition Program Administration (DAPA) (KRIT-CT-22-074) and was also supported by a grant from Kyung Hee University in 2023 (KHU-20230880).

Data Availability Statement: The original data presented in the study are openly available in oss-fuzz-lab at https://github.com/Jminis/oss-fuzz-lab (acessed on 20 March 2024).

Conflicts of Interest: Authors Jaemin Kim and Chongkyung Kil were employed by the company CW Research Inc. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Appendix A

Table A1 is a table summarizing the entire libFuzzer harness we utilized for our evaluation. We manually analyzed related source codes to identify and categorize parser types and other information and cross-checked the relevance to coverage feedback.

Table A1. Summarized information of bugs we used in the evaluation. `LoC` stands for `Line of Code` (C/C++). To check the LoC, we manually downloaded all relevant github source codes from `Dockerfile` provided by OSS-Fuzz project. We have categorized each parser type based on their fuzzed target API implementation.

No	Project Name	Fuzzer Name	LoC	Category/Description
1	coturn	FuzzStun	40,351	Network Packet Parser/Parsing and validating the STUN protocol message
2	example	do_stuff_fuzzer	67	Syntax Language Parser/Checking for mismatches with certain strings
3	hiredis	format_command_fuzzer	9552	Syntax Language Parser/Parsing redis commands that include format specifiers
4	librdkafka	fuzz_regex	138,892	Syntax Language Parser/Interpreting and compiling regular expressions
5	libredwg	llvmfuzz	873,276	Multimedia Data Parser/Parsing dwf files in different formats, such as binary, dxj, and json
6	libyaml	libyaml_dumper_fuzzer	10,529	Syntax Language Parser/Parsing and dumping a YAML data
7	llvm	llvm-special-case-list-fuzzer	8,261,434	Syntax Language Parser/Creating SpecialCaseList
8	lzo	lzo_decompress_target	13,127	Stream Data Parser/Perform decompression based on the Lempel–Ziv–Oberhumer algorithm
9	ntpsec	FuzzServer	62,158	Network Packet Parser/Interpreting a packet according to the NTS protocol
10	open62541	fuzz_mdns_message	152,191	Network Packet Parser/Parsing network packets according to the mDNS protocol
11	openbabel	fuzz_obconversion_smiles	306,793	Multimedia Data Parser/Interpreting the input as a chemical data format
12	plan9port	fuzz_libsec	410,170	Syntax Language Parser/Parsing and printing the input as an ASN.1 element
13	pupnp	FuzzIxml	38,138	Syntax Language Parser/Parsing the input as an XML document
14	readstat	fuzz_format_sas7bcat	32,281	Multimedia Data Parser/Parsing the structure and content of SAS catalog files
15	readstat	fuzz_format_sav	32,281	Multimedia Data Parser/Parsing the input as SAV file format
16	ruby	fuzz_ruby_gems	411,797	Syntax Language Parser/Call ruby gems and library functions (date, regexp, json, etc.) with the input
17	serenity	FuzzILBMLoader	845,258	Multimedia Data Parser/Parsing the ILBM image format in the Serenity operating system
18	simd	simd_load_fuzzer	265,725	Multimedia Data Parser/Interpreting the input data as an image
19	vlc	vlc-demux-dec-libFuzzer	626,259	Multimedia Data Parser/Handling the input using VLC's demuxing functionalities
20	vulkan-loader	json_load_fuzzer	54,897	Syntax Language Parser/Parsing the data as the json format
21	c-blosc2	decompress_chunk_fuzzer	117,051	Stream Data Parser/Validating and decompressing the input with a Blosc compressor
22	fluent-bit	cmetrics_decode_fuzz	1,277,129	Stream Data Parser/Decoding the data with different decoders (OpenTelemetry, Msgpack, Prometheus)
23	vulkan-loader	instance_create_fuzzer	54,897	Syntax Language Parser/Parsing the data as a json config file and creating the Vulkan instance
24	vulkan-loader	instance_enumerate_fuzzer	54,897	Syntax Language Parser/Parsing the data as a json config file and enumerating instance extensions
25	augeas	augeas_fa_fuzzer	26,553	Syntax Language Parser/Interpreting and compiling regular expressions
26	bloaty	fuzz_target	11,553	Multimedia Data Parser/Parsing the input to analyze the structure and size
27	fluent-bit	cmetrics_decode_fuzz	1,277,129	Stream Data Parser/Decoding the data with different decoders (OpenTelemetry, Msgpack, Prometheus)
28	glog	fuzz_demangle	9729	Syntax Language Parser/Interpreting and converting mangled names (symbols) into demangled forms
29	haproxy	fuzz_cfg_parser	239,548	Syntax Language Parser/Interpreting the input as a HAProxy's configuration file
30	hiredis	format_command_fuzzer	9552	Syntax Language Parser/Parsing redis commands that include format specifiers
31	ibmswtpm2	fuzz_tpm_server	46,963	Network Packet Parser/Interpreting the input as the commands and responding by the TPM protocol
32	libbpf	bpf-object-fuzzer	124,043	Stream Data Parser/Parsing the input as a BPF object
33	libical	libical_extended_fuzzer	47,640	Syntax Language Parser/Interpreting the input as the iCalendar data format
34	libredwg	llvmfuzz	873,276	Multimedia Data Parser/Parsing dwf files in different formats, such as binary, dxj, and json
35	oatpp	fuzz_mapper	32,794	Syntax Language Parser/Interpreting the input as json data and converting it into an oatpp object
36	pcapplusplus	FuzzTargetNg	89,528	Network Packet Parser/Interpreting the input as a PCAP file and parsing packets
37	php	php-fuzz-execute	1,182,120	Syntax Language Parser/Interpreting the input as a PHP code and executing it
38	php	php-fuzz-function-jit	1,182,120	Syntax Language Parser/Interpreting the input as a PHP code and executing it
39	wabt	read_binary_interp_fuzzer	763,941	Syntax Language Parser/Parsing the input conforming to the WebAssembly binary format
40	wabt	read_binary_ir_fuzzer	763,941	Syntax Language Parser/Parsing the input conforming to the WebAssembly binary format
41	augeas	augeas_api_fuzzer	26,553	Syntax Language Parser/Interpreting the input as the config file and parsing it with node operation APIs
42	augeas	augeas_escape_name_fuzzer	26,553	Syntax Language Parser/Escaping the input and finding matches of a path expression
43	c-blosc2	decompress_frame_fuzzer	117,051	Stream Data Parser/Creating chunks with the input and decompressing with a Blosc compressor
44	cups	FuzzCUPS	170,349	Syntax Language Parser/Interpreting the input as a PostScript code and executing it

References

1. Miller, B.P.; Fredriksen, L.; So, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* **1990**, *33*, 32–44. [CrossRef]
2. Manès, V.J.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331. [CrossRef]
3. Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Available online: https://github.com/google/oss-fuzz (accessed on 20 March 2024).
4. Fu, Y.F.; Lee, J.; Kim, T. autofz: Automated Fuzzer Composition at Runtime. *arXiv* **2023**, arXiv:2302.12879.
5. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.

6. Dolan-Gavitt, B.; Hulin, P.; Kirda, E.; Leek, T.; Mambretti, A.; Robertson, W.; Ulrich, F.; Whelan, R. Lava: Large-scale automated vulnerability addition. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 110–121.
7. 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, Athens, Greece, 23–28 August 2021; pp. 1393–1403.
8. Poet, M. Achieving 100% Code Coverage: Is It Worth It? Available online: https://methodpoet.com/100-code-coverage/ (accessed on 28 February 2024).
9. Böhme, M.; Szekeres, L.; Metzman, J. On the reliability of coverage-based fuzzer benchmarking. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1621–1633.
10. Jang, J.; Son, G.; Lee, H.; Yun, H.; Kim, D.; Lee, S.; Kim, S.; Jang, D. Fuzzability Testing Framework for Incomplete Firmware Binary. *IEEE Access* **2023**, *11*, 77608–77619. [CrossRef]
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, Virtual, 15–19 November 2021; pp. 337–350.
12. 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 NDSS, San Diego, CA, USA, 18–21 February 2018.
13. Google. ClusterFuzz: Coverage-Guided vs. Blackbox Fuzzing. Available online: https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox/ (accessed on 20 March 2024).
14. Atlassian. Code Coverage: What Is It and How Do You Measure It? Available online: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage (accessed on 28 February 2024).
15. Buechner, F. Is 100% Code Coverage Enough? 2008. Available online: https://www.agileconnection.com/sites/default/files/article/file/2013/XUS252268366file1_0.pdf (accessed on 20 March 2024)
16. Ivanković, M.; Petrović, G.; Just, R.; Fraser, G. Code coverage at Google. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 955–963.
17. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. {AFL++}: Combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), Online, 11 August 2020.
18. LLVM LibFuzzer Documentation. Available online: https://llvm.org/docs/LibFuzzer.html (accessed on 28 February 2024).
19. Wiebing, S.J.; Rooijakkers, T.; Tesink, S. Improving AFL++ CmpLog: Tackling the Bottlenecks. In *Science and Information Conference*; Springer: Cham, Switzerland, 2023; pp. 1419–1437.
20. Schiller, N.; Chlosta, M.; Schloegel, M.; Bars, N.; Eisenhofer, T.; Scharnowski, T.; Domke, F.; Schönherr, L.; Holz, T. Drone Security and the Mysterious Case of DJI's DroneID. In Proceedings of the NDSS, San Diego, CA, USA, 27 February–3 March 2023.
21. Nagy, S.; Hicks, M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA USA, 19–23 May 2019; pp. 787–802.
22. Wang, J.; Duan, Y.; Song, W.; Yin, H.; Song, C. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Beijing, China, 23–25 September 2019; pp. 1–15.
23. 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.
24. Hou, Z.; Sanan, D.; Tiu, A.; Liu, Y.; Hoa, K.C. An executable formalisation of the SPARCv8 instruction set architecture: A case study for the LEON3 processor. In Proceedings of the FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, 9–11 November 2016; Proceedings 21; Springer: Cham, Switzerland, 2016; pp. 388–405.
25. Cederman, D.; Hellström, D.; Sherrill, J.; Bloom, G.; Patte, M.; Zulianello, M. Rtems smp for leon3/leon4 multi-processor devices. *Data Syst. Aerosp.* **2014**, *180*. Available online: https://gedare.github.io/pdf/cederman_rtems_2014.pdf (accessed on 20 March 2024).
26. Clements, A.A.; Carpenter, L.; Moeglein, W.A.; Wright, C. A case study in re-hosting VxWorks control system firmware. Available online: https://bar2021.moyix.net/bar2021-preprint6.pdf (accessed on 20 March 2024).
27. Vasselle, A.; Maurine, P.; Cozzi, M. Breaking mobile firmware encryption through near-field side-channel analysis. In Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, London, UK, 15 November 2019; pp. 23–32.
28. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
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, 21–23 May 2018; pp. 679–696.
30. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Melbourne, Australia, 21–25 September 2020; pp. 475–485.

31. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108.
32. Wu, M.; Lu, M.; Cui, H.; Chen, J.; Zhang, Y.; Zhang, L. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 56–68.
33. Wang, M.; Liang, J.; Chen, Y.; Jiang, Y.; Jiao, X.; Liu, H.; Zhao, X.; Sun, J. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden , 27 May–3 June 2018; pp. 61–64.
34. Zhou, C.; Wang, M.; Liang, J.; Liu, Z.; Jiang, Y. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 December 2020; pp. 858–870.
35. Schumilo, S.; Aschermann, C.; Gawlik, R.; Schinzel, S.; Holz, T. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In Proceedings of the 26th USENIX security symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 167–182.
36. Pham, V.T.; Böhme, M.; Roychoudhury, A. AFLNet: A greybox fuzzer for network protocols. In Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 24–28 October 2020; pp. 460–465.
37. Gao, Z.; Dong, W.; Chang, R.; Wang, Y. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e5756. [CrossRef]
38. Kim, J.; Yu, J.; Kim, H.; Rustamov, F.; Yun, J. FIRM-COV: High-coverage greybox fuzzing for IoT firmware via optimized process emulation. *IEEE Access* **2021**, *9*, 101627–101642. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.