Understanding and Improving Coverage Tracking with AFL++ (Registered Report)

Vasil Sarafov
vasil.sarafov@unibw.de
μCSRL—Munich Computer Systems Research Lab
University of the Bundeswehr Munich, Germany

David Markvica
david.markvica@unibw.de
μCSRL—Munich Computer Systems Research Lab
University of the Bundeswehr Munich, Germany

Felix Berlakovich
felix.berlakovich@unibw.de
μCSRL—Munich Computer Systems Research Lab
University of the Bundeswehr Munich, Germany

Matthias Bernad
matthias.bernad@unibw.de
μCSRL—Munich Computer Systems Research Lab
University of the Bundeswehr Munich, Germany

Stefan Brunthaler
brunthaler@unibw.de
μCSRL—Munich Computer Systems Research Lab
University of the Bundeswehr Munich, Germany

Abstract

Coverage-based fuzzers track which program parts they visit when executing a specific input as a proxy measure to (1) guide the fuzzing process, and (2) explore the Program Under Test (PUT)'s state space. One way to record coverage progress is to enumerate basic block pairs (i.e., edges in the control-flow graph) and use them to index into a hash table that holds counters. The counter is incremented every time a fuzzer's input exercises the corresponding edge. Traditionally the coverage map has been a compact bitmap that fits the L2 CPU cache to reduce runtime overhead and boost fuzzing throughput. In such a design where space is traded for speed, two sources of imprecision can arise: (1) collisions, and (2) arithmetic inaccuracies.

Collisions refer to the situation when two *different* basic block pairs hash to the same entry. Imprecision arises since one pair is now counted together, but the fuzzer cannot tell one apart from the other.

Arithmetic inaccuracies refer to errors in the counting strategy. For example, a monotonically incrementing counter inside the hash table can overflow. This indicates a situation where high-frequency control-flow exceeds the predefined, expected maximum counter size (e.g., in loops). Due to execution frequencies obeying exponential power laws, such overflows will affect a small number of hash table entries. Another arithmetic inaccuracy results from range-based counters that capture only predefined frequency intervals (e.g., logarithmic counters).

In 2018, CollAFL examined how collisions impact precision, and presented a new hashing scheme to reduce the number of collisions. CollAFL did not address the problem of arithmetic inaccuracies. Furthermore, CollAFL considered only a single-core virtual machine, a limited set of benchmark programs, and did not explore hardware-specific effects (e.g., cache utilization for concurrent fuzzing processes).

This registered report aims at providing new insights of how collisions and arithmetic inaccuracies affect coverage tracking for fuzzing. We propose experiments for multiple hardware architectures with different cache topologies, and a more diverse set of benchmark programs. Leveraging the evaluation data, our aim is to determine precise *architecture-aware* settings for AFL++. Furthermore, we plan to demonstrate an *adaptive optimization strategy* that optimizes the coverage map w.r.t. to collisions and counting strategies for a specific combination of the CPU architecture and PUT.

CCS Concepts

• **Software and its engineering → Empirical software validation**.

Keywords

Software Engineering, Testing, Fuzzing, Coverage-Guided Fuzzing, Coverage Tracking, Hashmap, Hitmap, Imprecision, Collisions, Overflows, Empirical Study

ACM Reference Format:
Vasil Sarafov, David Markvica, Felix Berlakovich, Matthias Bernad, and Stefan Brunthaler. 2024. Understanding and Improving Coverage Tracking with AFL++ (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3678722.3685537

FUZZING '24, September 16, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685537

1 Introduction

Coverage-guided fuzzers have shown to be useful for identifying vulnerabilities in complex software. This type of fuzzer records coverage of the Program Under Test (PUT) for specific inputs to determine more effective *seed scheduling* and *mutation*. Well known

representatives of coverage-based fuzzers derive from AFL, such as aflpp [6].

Both AFL and AFL++ record coverage by (1) instrumenting the PUT, and (2) supplying a data structure to keep track if a seed reaches a certain program point. The instrumentation step (usually) adds recording instructions into the lowest level of compound program structure: basic blocks. The recording data structure is commonly referred to as a *coverage map*. To reduce the overhead from instrumentation and coverage tracking, the coverage map has been traditionally kept small so that it fits the L2 CPU cache (e.g., 64 KB in AFL's initial design [20]).

Fuzzing research of the past decade has focused primarily on improving the seed scheduling and mutation stage using coverage data [2, 5, 9, 13]. This focus did not, however, address the shortcomings in coverage recording, namely imprecision due to *collisions* and *arithemtic inaccuracies*.

Collision imprecision refers to the problem of having multiple program locations index into the *same* coverage map recording slot. Such a collision leads to imprecision because the fuzzer cannot separate multiple different program locations by only considering the counts inside the coverage map slot.

Arithmetic inaccuracies refer to the problem of the counting strategy. Consider, for example, a recording slot size of one byte. Exceeding the execution count of 255, the recorded count value will overflow back to zero. Such a collision leads to imprecision because the fuzzer cannot reliably tell which input parts were executed more or less often modulo the 256 countable executions.

Figure 1 illustrates both sources of imprecision on a conceptual coverage map data structure. We note, however, that in the general case and depending the coverage type, a certain degree of imprecision is inherent to every fuzzer. If a fuzzer were to track detailed coverage of every possible control-flow and data state, fuzzing would reduce to (bounded) model checking [3]. The problem then becomes *tracking coverage optimally—reducing imprecisions (e.g., collisions and arithmetic inaccuracies), and maintaining high fuzzing throughput.*

Summing up, this paper makes the following contributions:

- We analyze existing implementations for coverage recording mechanisms in AFL and AFL++. Our analysis focuses specifically on highlighting existing mechanism's impotence to prevent both collision and arithmetic imprecision.
- We devise a set of experiments aimed at identifying *and* quantifying the imprecision of both sources in real-world programs.
- We present results of a preliminary study that shows as much as half of a PUT's coverage map slots can overflow (e.g., 49.5% for `zlib`).
- We propose a detailed evaluation across multiple hardware architectures to (1) highlight shortcomings, and (2) identify ideal parameters for each hardware architecture (e.g., w.r.t. CPU cache utilization).

2 Background

This section briefly explains (1) how AFL [20] and AFL++ [6] implement coverage-guided fuzzing, and (2) what Link-time Optimization (LTO) is.

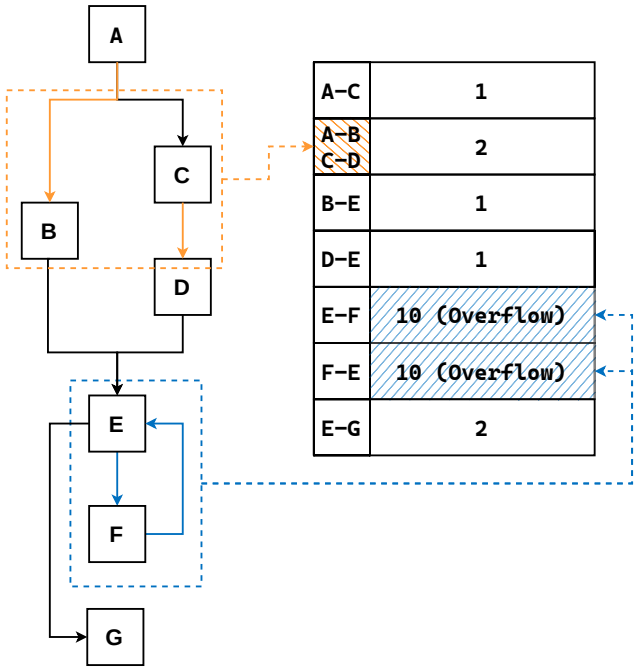

Figure 1: Overview of *hash-collisions* (`A-B` and `C-D` in orange) and an *arithmetic inaccuracy* stemming from counter overflows (`E-F` and `F-E` in blue) in coverage recording techniques.

2.1 Coverage-Guided Fuzzing

Fuzzing refers to the idea of subjecting programs to randomly generated inputs [13]. The program is commonly referred to as a Program Under Test (PUT). In principle, a fuzzer can follow a brute-force approach, i.e., just randomly generate an input and feed it to the PUT. An existing body of research, however, shows the effectiveness of using an adaptive, feedback-driven approach. Among several alternatives, *coverage* has shown to be a useful feedback-channel. By instrumenting a PUT to record the execution of its basic blocks, fuzzers use coverage to (1) guide the testing process, and (2) explore the PUT's state space. For example, the recorded information allows the fuzzer to correlate individual inputs with the PUT's state space, and thus control the mutation process.

Fuzzers instrument a PUT through inserting coverage recording tasks into a PUT at compile-time (cf. Section 3). Alternatively, AFL++ [6] provides a way to use FRIDA to instrument basic blocks in a binary-only mode[1], or LTO mode to instrument at link-time.

2.2 Link-Time Optimization

Real-world software today comprises multiple compilation units, i.e., object files that are compiled separately and then linked together. Separate compilation and subsequent linking pose an obstacle to both optimizing compilers and fuzzers.

Optimizing compilers cannot perform so-called whole-program optimizations, as the program in its entirety is only known *after* linking. Modern compilers, such as LLVM, address this issue by

[1] https://github.com/AFLplusplus/AFLplusplus/blob/stable/frida_mode/README.md

providing a so-called LTO infrastructure[2] [8, 11]. Using LTO, LLVM can, for example, detect unused functions and perform dead-code optimization across all compilation units.

Fuzzers face a related problem: For efficient indexing into the coverage array, each basic block would need a strictly monotonic, unique identifier that can be used as index. Such identifiers are only possible, however, by *enumerating* all basic blocks of a PUT. But how can the compiler enumerate the basic blocks of a PUT, if separate compilation is used?

One could, instead, generate a non-monotonic but unique identifier for all basic blocks during compilation (e.g., a pseudo-random identifier). The downside of this approach is that the hash function that computes the index into the coverage hash-table becomes expensive, thereby needlessly slowing down PUT execution. In addition, the exact size of the coverage hash-table is only known at *link-time*.

By compiling with LTO, however, the compiler is able to determine the number of basic blocks. This number allows the compiler to skip the hashing part and instrument with more efficient array indexing instead. In addition, the exact size of the coverage hash-table is known, enabling the fuzzing system to adjust its space requirements.

3 Problem Statement and Motivating Example

3.1 Original Approach Taken by AFL

AFL's technical description whitepaper contains the following snippet to illustrate coverage recording [20]:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

With the additional detail given that `shared_mem` corresponds to a 64 KB memory chunk. The motivation for such a small chunk is that it fits perfectly inside the L2 cache of a CPU. The purpose of the right-shift operation in the last line is to preserve the direction of control-flow transfers. Because the XOR operation (^) is symmetric, the direction between two basic blocks A and B is identical, i.e., $A \oplus B \equiv B \oplus A$. By shifting the current location by one bit to the right, however, the direction $A \rightarrow B$ is preserved: $A \oplus B^{\twoheadrightarrow} \not\equiv B \oplus A^{\twoheadrightarrow}$.

Recognizing collisions. By providing a list of hash collisions, Michał Zalewski already notes that they can and do occur. Specifically, Zalewski notes that the collisions increase with increasing numbers of branch points. At 50,000 branch points, the number of collisions rises to about 30%.

Frequency counter imprecision. Overflows are also possible in the original AFL implementation and depend on the elementary datatype of the `shared_mem` array. Interestingly, Zalewski already points this out:

> The absence of simple saturating arithmetic opcodes on Intel CPUs means that the hit counters can sometimes wrap around to zero. Since this is a fairly unlikely and localized event, it's seen as an acceptable performance trade-off.

[2] https://llvm.org/docs/LinkTimeOptimization.html

Note that the issue of overflows is related to the issue of collisions. If two edges use the same coverage map slot due to a collision, they also increase the same frequency counter. As a result, the exact frequency of each edge is blurred.

3.2 Advancements by AFL++

3.2.1 Context-Sensitive Compile-Time Instrumentation. AFL++ adds the option to add context to the coverage information. For example, call-context sensitivity adds information about the functions on the stack to the existing edge coverage.

```
map[
  current_location_ID
  ^ previous_location_ID >> 1
  ^ hash_callstack_IDs
]
+= 1;
```

The corresponding documentation gives the following explanation:

> Basically every function gets its own ID and, every time when an edge is logged, all the IDs in the callstack are hashed and combined with the edge transition hash to augment the classic edge coverage with the information about the calling context.
>
> So if both function A and function B call a function C, the coverage collected in C will be different.
>
> The callstack hash is produced XOR-ing the function IDs to avoid explosion with recursive functions.

Besides call-context sensitivity, AFL++ can also approximate path-sensitivity via its `n-gram` instrumentation [6]. For example, in a `4-gram` setting AFL++ will not only track a single basic block, but also its three predecessors.

On one hand context sensitivity allows AFL++ to explore new PUT states. On the other hand, however, it increases the pressure on the coverage map, and thus increases the likelihood of more collisions. Consider, for example, a function f in a given PUT. If f is called from twenty different call sites, it will require twenty times more slots in the coverage map.

Note that other instrumentations, not necessarily implying context sensitivity, also increase the pressure on the coverage map. For example, the `LAF-INTEL` transformation[3] breaks down composite multi-byte comparisons to simple single-byte comparisons. Thus it increases the number of edges in the control-flow graph, and the number of used slots of the coverage hashmap.

3.2.2 Larger Coverage Map to Reduce Collisions. The default parameter of shared memory allocation to hold the coverage map in AFL++ is set to eight megabytes[4]. This means that eight million bytes can be stored in the coverage map and that 23 bits are required to address a single byte in the coverage map.

Due to the pigeonhole principle, if a program has more than eight million edges, collisions *must* occur. Because AFL++ uses context-sensitive optimizations (see above), a program with fewer edges could already lead to collisions.

[3] https://lafintel.wordpress.com/

[4] See, for example, the `get_map_size` procedure in `afl-common.c` and `DEFAULT_SHMEM_SIZE` in `config.h`.

Specifically, the AFL++ developers state the following (N.B. emphasis copied from original source):

> *This issue is underestimated in the fuzzing community.* With a 2^{16} = 64kb standard map at already 256 instrumented blocks, there is on average one collision. On average, a target has 10.000 to 50.000 instrumented blocks, hence the real collisions are between 750-18.000!

AFL++ contains additional comments regarding coverage map size:

> Most targets just need a coverage map between 20-250kb. [...] Hence afl-fuzz deploys a larger default map. The largest map seen so far is the `xlsx` fuzzer for LibreOffice which is 5MB.

AFL++'s git repository dates this commit[5] to March 17th, 2021. Before this commit the amount was set to 1 MB, instead of the now standard 8 MB.

However, a too large coverage map reduces the fuzzer's throughput (cf. Section 6). The reason is that the map can no longer fit the CPU cache, which leads to frequent cache misses while tracking coverage. Thus the fuzzer's bookkeeping runtime overhead increases, and its throughput declines.

3.2.3 Preventing Overflowing Counters Wrapping to Zero. If a counter in the coverage map overflows, it will automatically wrap to zero. As a result, if there is no follow-up incrementation before the PUT terminates, the fuzzer will fail to register the exercized state(s). To prevent this from happening, AFL++ does not only increment the counter by one, but also adds the carry bit[6]. As a consequence overflowing counters in AFL++ wrap to one.

Nevertheless, even non-zero, overflowing counters still lead to missed states due to high execution frequencies (e.g., in loops, cf. below). Interestingly, AFL++'s authors attempted to implement saturated counters that freeze at their maximum value before overflowing. However, because of the additional branching, their performance overhead was found to be too high [6, Section 3.3].

3.2.4 Buckets vs. Overflows. So far we have only considered simple counters that are monotonically incremented by one. AFL and AFL++ implement an alternative approach, so-called *buckets*. Buckets group frequencies into ranges and store only whether the execution frequency lies within a certain interval (e.g., logarithmic counters). The assumption behind buckets is that, especially for larger frequencies, small differences in frequency should be irrelevant.

Buckets are an improvement over a naive counter as they reduce the number of overflows. However, they still have issues. Buckets lose precision with larger execution frequencies. For example, AFL's default implementation does not differentiate execution counts of 200 and 2000.

3.2.5 Link-Time Optimization. As mentioned in the background, the precise number of basic blocks is not known until link-time. AFL++, thus, offers to use LLVM's LTO interface to enumerate all basic blocks at link-time. By using this information, for a given PUT we can create a perfect hashmap, free of static collisions.

Even in the face of LTO-based coverage map optimization, three problems remain. First, even though a minor issue, not all software is readily amenable to use an LTO-build process. `Libxml2`, for example, does not currently build *with* LTO enabled. Second, imprecisions due to arithmetic inaccuracies (e.g., overflowing counters) remain unaffected by coverage map size optimization. Third, collision-free only refers to edge coverage based on basic block identifiers. Potential collisions due to path-sensitivity approximation, such as call-context sensitivity, remain an open issue. For example, AFL++'s LTO collision-free instrumentation cannot handle indirect procedure calls, thus completely missing PUT states resulting from such calls[7].

3.3 Problem and Impact

In summary, coverage-guided fuzzers, by an example of AFL++, lose precision in coverage tracking due to (1) collisions in the coverage map, and (2) arithmetic inaccuracies in how the hit counters are managed.

First, a coverage map large enough to hold every control-flow edge must not necessarily eliminate all collisions (cf. Section 4). For example, the hash function could capture additional context, e.g., procedure calls. Furthermore, large coverage maps could negatively impact the fuzzer's throughput as they no longer fit the CPU cache. Additionally, AFL++'s link-time instrumentation is collision-free only w.r.t. basic-block edges and direct procedure calls. Specifically, AFL++'s LTO is incomplete, as it fails to register call-context sensitivity for indirect calls (this information is fully available only at runtime).

Second, arithmetic inaccuracies, specifically imprecise range buckets and overflowing counters (even if not wrapping to zero), lead to the fuzzer missing PUT states. For example, hot code, such as loops, becomes blurred to the coverage tracking. How significant such PUT state misses are is an open research question (see also Section 5). We can expect that the impact depends on the fuzzer's scheduling strategy. However, even a single bit missed from the coverage map can mean a potentially wrongfully discarded test case[8].

Accordingly, the problem becomes tracking coverage optimally—reducing imprecisions, and maintaining high fuzzing throughput.

4 Preliminary Results

Even though not up to date, related work already provides data about the problems collisions in a small coverage map can cause (cf. Section 3.1 and Section 6). Here we present the results of a preliminary study we conducted about the presence of (1) arithmetic inaccuracies, and (2) collisions purely because of the hashing function, given a pratically infinite coverage map.

[5] AFL++ commit 5e2a5f1110e29c36f1c41fb4677ab698c5d571c0

[6] See `skip_nozero` in `afl-llvm-pass.so.cc` for the implementation, and `NeverZero` in the documentation [6, Section 3.3].

[7] AFL++ uses a fork of LLVM's `SanitizerCoverage` for its LTO implementation (cf. `SanitizerCoverageLTO.so.cc`).

[8] See `save_if_interesting`, `has_new_bits`, and `has_new_bits_unclassified` in `afl-fuzz-bitmap.c`. Also see `update_bitmap_score` and `cull_queue` in `afl-fuzz-queue.c`.

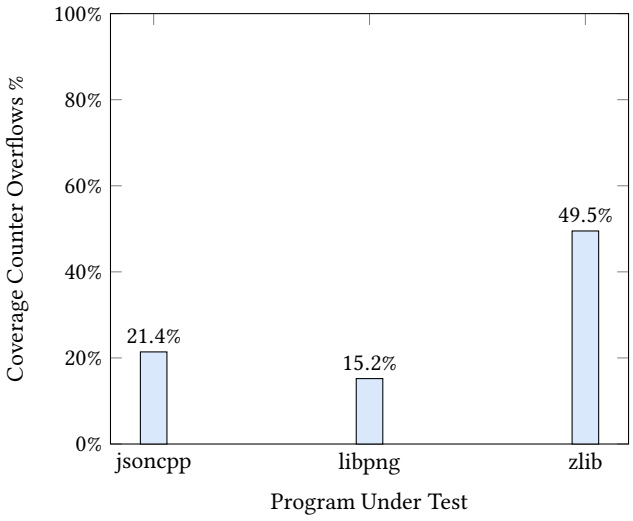

Figure 2: Percentage of counter overflows in AFL++'s coverage map, using OSS-Fuzz's corpora for the given Program Under Test (PUT).

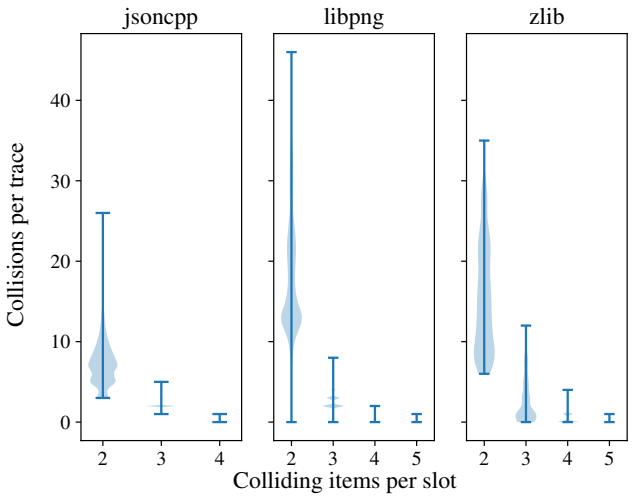

Figure 3: A violin plot summarizing the relationship between number of colliding items per slot and their order in a coverage map of maximum size for each examined PUT. For example, a single fuzzing execution for jsoncpp can lead to up to 28 slots where 2 different edges collide.

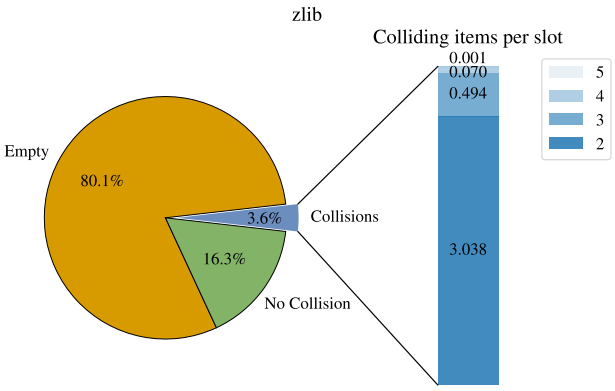

Figure 4: Detailed breakdown of slot collisions in the coverage map of zlib.

4.1 Synopsis

Specifically, we measure the number of overflows in the jsoncpp, libpng, and zlib PUTs accumulated by oss-fuzz's saturated corpora, whereby we maximize each coverage map's size to correct for collisions. We find that 21.4% of jsoncpp's slots overflow, for libpng 15.2%, and for zlib 49.5% respectively (cf. Figure 2).

Furthermore, 2.8% of jsoncpp's slots wrap back to zero (or one) because of the overflow. For libpng the wraps are 1.2%, and for zlib 1.4% respectively.

In addition, we measure the number of collisions and their order (cf. Figure 3 and Figure 4). We find that in 100% of the cases, jsoncpp and zlib have at least one collision. For libpng, this is 98%. Additionally, we observe, for example, that libpng has up to 5 colliding items per slot. However, most of the collisions (≈ 50) are of order 2.

Below we discuss in greater detail the experiment and what the results imply for the next phase of our research.

4.2 Methodology and Implications

First, we download the latest saturated corpora for all three PUTs from oss-fuzz. For that we run a local oss-fuzz experiment and set `oss-fuzz-corpus: true` in the configuration. Note this allows us to take advantage of a dataset generated by different fuzzers in hundreds of CPU hours[9][10][11].

Second, we build an instrumented version for each project, together with its corresponding fuzz driver. For the instrumentation we use SanitizerCoverage[12] to record each edge and the total number of basic blocks.

Third, we execute each instrumented PUT with all of its input files from the corpus, and record the corresponding execution trace. We compress large traces with lz4 to prevent running out of storage space.

Finally, we analyze each execution trace, and emulate AFL++'s coverage tracking with a coverage map big enough to store each basic block pair in its own entry. During the analysis we keep a list of each hit in a map's entry. At the end, lists longer than 255 indicate counter overflows, as the coverage map is a byte array. Figure 2 depicts the final result.

We count the coverage map collisions using a chaining method to keep track of the collisions. At the end we calculate every chain's length.

[9] https://introspector.oss-fuzz.com/project-profile?project=jsoncpp
[10] https://introspector.oss-fuzz.com/project-profile?project=zlib
[11] https://introspector.oss-fuzz.com/project-profile?project=libpng
[12] https://clang.llvm.org/docs/SanitizerCoverage.html

Interestingly only 15.2% of `libpng`'s slots and almost half of `zlib`'s overflow. Due to time constraints we have not analyzed the exact reason for such a distribution, but plan to do in the next phase (cf. Section 5). However, a cursory examination reveals that `zlib` has a longer critical path that utilizes loops, whereas `libpng`'s critical path is largely linear.

Our experiment does not contain enough PUTs for its results to be representative. However, it suggests that there could be a PUT-specific adaptation for the coverage map, such that the overflows are reduced (e.g., larger slots for hot code such as loops).

Furthermore, it is not yet clear what the exact impact of overflows is w.r.t. coverage precision and fuzzer performance. In addition, in this experiment we have eliminated overflows due to collisions, as we maximized the coverage map's size. However, this assumption is not realistic, as this would penalize performance in real-world fuzzing campaigns. In the next section we design a set of new experiments to shed light on these questions and better our understanding about coverage tracking.

5 Methodology and Research Questions

In this section we (1) describe the questions we plan to investigate in the second stage of our research, (2) outline a workplan, and (3) design the required experiments.

At the core of our study are two research questions:

- How much precision does a coverage-guided fuzzer lose because of inaccuracies in its coverage tracking (i.e., collisions, arithmetic inconsistencies)?
- How much performance does a fuzzer lose if we improve its coverage tracking precision?

We note both questions contain multiple degrees of freedom, and thus require several experiments. For that reason, we divide each question in multiple, smaller, more easily to address subproblems.

Furthermore, we organize our experiments in two groups—one for studying the *imprecision* (cf. Section 5.4), and one for addressing the *performance* (cf. Section 5.5).

Before we detail each experiment, we describe the common components in our setup—the target hardware systems, software, and dataset.

5.1 Evaluation: Target Hardware Systems

The coverage map is for performance reasons a cache-sensitive data structure. Furthermore, CPU caches are property of the microarchitecture. For example, cache size and whether a cache bank is shared among CPU cores varies with the microarchitecture. Thus, for our experiments we plan to use 12 different systems from 3 instruction set architectures (`RISC-V`, `ARM`, and `x86_64`) and diverse cache topologies (cf. Table 1).

The systems range from more constrained (e.g., middle-class 32-bit embedded systems) to desktop machines with high single-core performance (e.g., `i7`, and `i9` CPUs), and server systems with multi-core packages and larger cache sizes. Our motivation for choosing a diverse set of evaluation hardware is twofold. First, slower devices with less cache should be more sensitive to smaller changes in the coverage tracking mechanism, thus making the analysis of results easier. Similarly, CPU architectures with distributed topologies amplify effects when fuzzing in parallel (e.g., separate fuzzer instances compete for shared caches). Second, results that are consistent across multiple different devices are more likely to be representative.

5.2 Evaluation: Software Setup

To reduce variability in our experiments because of software differences, we equip all machines (see Table 1) with standard `Debian 12`, without virtualization and without patching the kernel.

Furthermore, we will use exclusively `AFL++`[13] and its `LLVM`[14] instrumentation backend. However, depending on the experiment (see below), we propose to examine `AFL++`'s behavior under different configurations. When required, we patch `AFL++` accordingly.

As part of our final, stage-two publication, we will make all source code, data, and documentation available to the public to facilitate open and reproducible research.

5.3 Evaluation: Dataset

We select 90 PUTs from `oss-fuzz`, representing 15 groups of user-space programs with different demands and programming patterns (cf. Table 2). For example, numerical code and media codecs are expected to have tight CPU-bound parts. On the other hand, I/O should dominate in PUTs from the network and database categories. Furthermore, parsers and programming language systems require fuzzing inputs to satisfy complex syntax and semantic constraints.

We note all PUTs have `AFL` fuzzing drivers, and are written in either `C` or `C++`. Additionally, our dataset is a superset of `FuzzBench`'s [14].

Besides the abovementioned dataset, we also take advantage of the `SPEC CPU 2017` benchmark to study the performance impact of different coverage strategies (cf. Section 5.5).

5.4 Research Questions and Experiments: Imprecision

Here we propose an evaluation aimed at understanding how collisions and arithmetic inaccuracies impact coverage precision. To this end, we first need to identify *sources* for both types of imprecision. Once identified and understood, we will be in a position to analyze under which circumstances collisions and arithmetic inaccuracies occur, and also measure the extent to which these errors propagate throughout the fuzzing process.

With these goals in mind, we propose the following research questions.

Q.1 Where and how many collisions occur?
Q.2 Where how many, and what arithmetic inaccuracies occur?
Q.3 To which extent do collisions affect precision?
Q.4 To which extent do arithmetic inaccuracies affect precision?

5.4.1 Calibrating `AFL++`. To investigate the abovementioned problems, we undertake the following changes on `AFL++`. At the beginning, we set `AFL++`'s coverage map's size to 2 GB to correct for collisions[15]. Next, we change the coverage counters to monotonically increasing unsigned 64-bit slots. Then, as for our preliminary

[13] Commit `db23931e7c1727ddac8691a6241c97b2203ec6fc`, dated July 24, 2024.
[14] with `LLVM` version 18
[15] Our preliminary analysis shows that 2 GB is enough for our PUTs.

Table 1: Hardware systems planned for conducting the experiments. Abbreviations: p.c.—per core, sh.—shared. The `Apple M3 Max` machine has 12 performance and 4 low-powered CPU cores, and its shared caches are grouped in clusters.

Machine	Architecture	CPU Model	Cores	Threads	L1	L2	L3	RAM	Storage
apu4d4	x86, 32 bit	AMD Embedded BX-412TC, 1.00 GHz	4	4	32 KB p.c.	2 MB sh.	-	4 GB	128 GB SSD
visionfive2	RISC-V, 64 bit	StarFive JH7110, 1.50 GHz	4	4	32 KB p.c.	2 MB sh.	-	8 GB	1 TB NVMe
quartz64	ARM, 64 bit	ARM Cortex-A55, 2.00 GHz	4	4	32 KB p.c.	-	512 KB sh.	4 GB	1 TB NVMe
celeron	x86_64, 64 bit	Intel Celeron N2820, 2.41 GHz	2	2	56 KB p.c.	512 KB p.c.	-	4 GB	240 GB SSD
nuc8i7	x86_64, 64 bit	Intel i7-8559U, 4.50 GHz	4	8	64 KB p.c.	256 KB p.c.	8 MB sh.	64 GB	1 TB SSD
i9	x86_64, 64 bit	Intel i9-9900K, 5.00 GHz	8	16	64 KB p.c.	256 KB p.c.	16 MB sh.	64 GB	1 TB SSD
xeon	x86_64, 64 bit	Intel Xeon 8358, 3.40 GHz	32	64	64 KB p.c.	1 MB p.c.	48 MB sh.	256 GB	3.2 TB NVMe
ryzen	x86_64, 64 bit	AMD Ryzen 7 3700X, 4.40 GHz	8	16	256 KB p.c.	4 MB p.c.	32 MB sh.	64 GB	1 TB NVMe
epyc	x86_64, 64 bit	AMD EPYC 7H12, 3.30 GHz	64	128	96 KB p.c.	512 KB p.c.	256 MB sh.	1 TB	3.2 TB NVMe
threadripper	x86_64, 64 bit	AMD Threadripper 3970X, 4.50 GHz	32	64	64 KB p.c.	512 KB p.c.	128 MB sh.	128 GB	1 TB NVMe
ampere	ARM, 64 bit	Ampere Altra, 3.00 GHz	80	80	64 KB p.c.	1 MB p.c.	32 MB sh.	512 GB	3.2 TB NVMe
m3max	ARM, 64 bit	Apple M3 Max, 2.75 - 4.06 GHz	12/4	16	192/128 KB p.c	16/16/4 MB sh.	-	128 GB	1 TB NVMe

Table 2: Dataset for the (precision) experiments. All programs are part of `oss-fuzz` and have `AFL` fuzzing drivers.

Category	Program Samples (PUTs)
Programming Language Systems	ruby, wasmedge, wasm3, quickjs, php, mruby, lua, cpython3, janet
Database Systems	duckdb, sqlite3, postgresql
Cryptographic Libraries	wolfssl, openssl, mbedtls, bearssl, boringssl, libsodium, libressl
Network Libraries	curl, libpcap, openthread, libssh, libssh2, libtorrent
Web/HTTP Servers	apache-httpd, nginx, lighttpd, mongoose, h2o, uWebSockets
Network Daemons	dovecot, openssh, dropbear
JSON	jq, cjson, jsoncpp, json-c, boost-json
XML	tinyxml2, pugixml, libxslt, expat, libxml2, xerces-c
Compression Libraries	lz4, lzo, xz, zlib, zstd, zopfli, unrar, miniz, minizip
Video & Codecs	ffmpeg, wavpack, vorbis, vlc, openh264, mpg123, libmpeg2
Images	libpng, libspng, libtiff, imagemagick, libjpeg-turbo
Numerical Algorithms	xnnpack, llamacpp
Regular Expressions & Lexing	re2, flex, wuffs
Fonts	woff2, freetype2, harfbuzz
Others	lcms, proj4, systemd, git, libgit2, utf8proc, libyaml, xpdf, mupdf, md4c, msgpack-c, nanopb, file, capstone, keystone, bloaty

results, we keep track of collisions in the coverage map using chain lists (see Section 4).

5.4.2 Fuzzing Campaign. Subsequently, we execute two fuzzing campaigns.

First, we fuzz each PUT from our dataset (cf. Table 2) for 24 hours starting from `oss-fuzz`'s saturated corpora. Note that although saving time from a fuzzing campaign, using this corpus leads to more conservative results. In particular, since `oss-fuzz` has obtained the corpus from configurations that potentially suffered collisions and arithmetic inaccuracies, the finally reached PUT states are undercounted. Thus we try to correct for undercounting and fuzz for additional 24 hours, under `AFL++`'s default setup.

Second, we fuzz each PUT from the dataset for 72 hours without any initial corpus. This allows us to additionally correct for starting states that `AFL++` missed because of `oss-fuzz`'s corpus.

Finally, for both fuzzing campaigns, we store each seed and its corresponding coverage map in the order `AFL++` discovered them.

We note that for studying *imprecision* we can ignore variables with performance impact w.r.t. execution speed and hardware effects, such as the cache size. Here we safely ignore the machine properties as a degree of freedom. However, we correct for the reduced throughput because of the 2 GB coverage map with longer fuzzing duration (24 and 72 hours respectively). Additionally, we use a cluster of epyc machines to speed up the experiment (cf. Table 1).

5.4.3 Analysis. The collected seeds form our *evaluation base* w.r.t. which we can measure what collisions and arithmetic inaccuracies `AFL++` would suffer under different conditions, and what their impact is. More specifically, we identify three degrees of freedom for our analysis, which we explain below: (1) the coverage map's size, (2) the hit counter's datatype, (3) and additional `AFL++`'s optimizations.

To vary the first dimension — the map size — we start from 8 KB and continue in steps of 8 KB until 128 MB. The rationale for such progression is twofold. First, we want to cover different, but legitimate L1 through L3 cache sizes. Second, if the jumps between sizes are too big, we can miss local effects.

Furthermore, different `AFL++` configurations can increase the pressure on the coverage map. Thus, to address the second and third dimensions — counter's datatype and `AFL++` optimizations — we identify the following settings:

(1) default `AFL++` setup (with `NeverZero`),
(2) without `NeverZero`,
(3) 4-gram path sensitivity, and
(4) `CmpLog`[16], i.e., `REDQUEEN` [2].

Finally, for each possible coverage map size and `AFL++` setting, we compile every PUT, execute it with each seed from its collected corpus set, and measure:

- The number of collisions and their order (**Q.1**).
- The number and type of counter overflows, and missed counters because of bucketing (**Q.2**). We intend to report the distribution of hit counters.
- Which and how many seeds would be missed because of imprecise coverage map either due to collisions, or arithmetic inconsistencies (**Q.3** and **Q.4**). This metric offers us a

[16] https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md

conservative approximation of the PUT states AFL++ would miss in the respective setting.

Besides the quantitative metrics, we also propose a qualitative macro analysis. Specifically, we intend to trace high frequency counters to their respective source (e.g., particular program constructs, such as loops or repeatedly called subroutines).

5.5 Research Questions and Experiments: Performance and Hardware Dependence

Gaining precision w.r.t. tracking more PUT states costs performance. Specifically, a larger coverage map and more accurate hit counters increase the execution overhead. As a result, the fuzzer throughput suffers:

Q.5 At what performance cost can we track coverage more accurately?

To that end we propose to use `SPEC CPU 2017` as a compute-intensive benchmark to examine the impact of different coverage maps w.r.t. performance on different CPU architectures. Specifically, for each of the systems listed in Table 1, we propose to measure the overhead added to each PUT for coverage map sizes of 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, and 8 MB under AFL++'s default settings. Using such sizes allows us to cover approximately all quartiles of the available L1 and L2 caches at our disposal. We note, however, that during the precision experiments (see Section 5.4) we might become aware of a better resolution for the coverage map size, such that it allows us to expose the relationship between performance and precision more accurately.

The above experiment does not consider overhead to the overall fuzzing process. For that reason, we also measure the end-to-end overhead added to a fuzzing campaign using 6 PUTs from our dataset (see Table 2). More concretely, we measure the number of iterations AFL++ explores in 24 hours when fuzzing three compute-intensive (`zlib`, `llamacpp`, `lua`), and three I/O-bound programs (`sqlite3`, `curl`, `libpcap`). Our hypothesis is that for I/O-bound programs we should be able to increase precision with less performance penalty.

Furthermore, we plan to study the caching-behavior of the coverage map (specifically, L2 cache utilization). Consider, for example, whether the bitmap resides at all times in L2 cache or spills to L3 and further:

Q.6 What is the coverage map's L2 cache utilization?

Finally, we are interested in conflicts stemming from concurrent fuzzing. For example, L3 cache is usually shared between multiple cores or symmetric multithreads (e.g., hyperthreads on `Intel`). When fuzzing concurrently (i.e., we run multiple fuzzing processes and thus occupy multiple cores), the different fuzzer instances will run into cache conflicts, thus penalizing each other when collecting coverage:

Q.7 What is the coverage map's L3 cache utilization?

To address **Q.6** and **Q.7**, we plan on using `perf` to inspect during the above fuzzing campaign where the coverage map resides.

5.6 Hardware and PUT-specific Adaptive Optimizations

Finally, we propose to leverage the gathered data and implement *adaptive* optimization strategies. The previous research questions enable us to quantify both sources of imprecision (i.e., collisions and arithmetic errors), and understand hardware-specific effects (e.g., cache utilization). Here we want to automatically derive optimal fuzzing parameters for a given PUT and hardware. To that end, we hypothesize about the following optimization opportunities:

(1) The fuzzer can automatically select the optimal size for the coverage map. An optimal choice depends on the hardware's cache capacity and the PUT.
(2) Hot code parts, such as loops, might quickly reach the highest execution frequency bucket. For these cases, choosing slots better suited for large frequencies might be profitable (e.g., 64-bit counters). Likewise, we can save capacity for the coverage map with bit counters for simple, linear, fall-through paths.

An AFL++ fuzzing campaign may, therefore, use fine-tuned parameter settings derived from the PUT, for a specific hardware. Potential improvement could include increased precision and fuzzer throughput.

6 Related Work

In 2019 Wang et al. survey the different types of coverage metrics for fuzzing [17]. The authors recognize the need to study the impact of map collisions on exploring new PUT states.

The work closest to ours is CollAFL [7]. CollAFL is one of the first papers that systematically analyzes the interplay of coverage granularity, hash collisions, and fuzzing throughput. The authors acknowledge the possibility of increasing the coverage map size, but also show the resulting drop in fuzzing throughput (60% when increasing the coverage map from 64 KB to 4 MB). As an alternative, the authors suggest combining different hashing algorithms to increase the number of guaranteed or at least likely unique coverage map slots. From a conceptual point of view, CollAFL's hashing algorithms approximate the collision-free coverage afforded by AFL++'s LTO instrumentation (see Section 2.2).

CollAFL's investigation is an important step in addressing the issue of coverage imprecision. CollAFL's evaluation, however, had a different focus and is, thus, not suitable to fully understand the relationship between coverage and architectural details. First, CollAFL focuses primarily on hash collisions, but does not investigate the issue of imprecision due to arithmetic inaccuracies. Second, CollAFL's evaluation was performed on a single-core virtual machine with unspecified CPU architecture details, such as the L2 cache size. As thoroughly discussed in Section 3 and Section 5, the L2 cache size is an important factor in sizing the coverage map. Third, only some of the PUTs in CollAFL's evaluation are CPU bound. We think that a larger number of CPU bound PUTs needs to be investigated to understand the effect of hot code, such as tight loops, on counter overflows. Finally, CollAFL was published before the availability of fuzzing benchmark suites and coverage precision improvements such as AFL++'s LTO instrumentation. In our view a reevaluation with both a more standardized and extensive benchmark set as well as modern hardware provides new insights.

As we discussed in Section 3, Fioraldi et al. describe AFL++'s LTO mode, and how the fuzzer tries to mitigate overflowing counters by wrapping them to one instead of to zero [6].

In their work from 2023, Xu et al. study the frequency of hash collisions using AFL in a dataset of seven programs [19]. For this dataset, Xu et al. find on average more than 70% of the hash collisions are repeated across different seeds. Thus, the authors propose to eliminate hash collisions in small programs and statically generate unique identifiers for each edge. How to do this without LTO is not described. For large programs, the authors propose to rehash the identifiers to decrease the number of collisions. Xu et al. do not report data about the overhead their approach induces on AFL's throughput. Besides a small dataset, their study does not consider counter overflows and is conducted on a single machine.

In 2024, Borrello et al. highlight the issue of hash collisions and execution slowdown caused by context-sensitive coverage techniques [4]. Instead of improving the efficiency of coverage encoding or enlarging the bitmap, however, the authors suggest a predictive context-sensitive coverage approach. Their approach implements context-sensitivity through cloning functions which are predicted to profit from context sensitivity. The prediction is based on a dataflow analysis of the function call sites. Our optimizations and predictive context-sensitivity are mutually beneficial, as a more collision-resistant coverage map allows for more aggressive context sensitivity.

CollAFL highlights that naively increasing the coverage bitmap size is infeasible due to the decrease in fuzzing throughput. To allow for a larger coverage bitmap without hurting fuzzing performance, the authors of BigMap propose an indexed, two-level coverage data structure [1]. The authors show that with increasing bitmap sizes, fuzzers spend a considerable amount of time with analyzing the bitmap after each test run. BigMap improves over AFL's bitmap by allowing the analysis to focus only on parts that have changed in last test run. We think that BigMap and our envisioned architecture and PUT-specific optimizations are mutually beneficial.

In a work from 2019, Wang et al. analyze different coverage metrics, such as variants of branch coverage or memory-access aware coverage [18]. The paper introduces a new metric called *sensitivity* that allows to compare the ability of coverage metrics to differentiate program states. The authors find that more sensitive coverage metrics or even a combination of them indeed leads to an improved bug-finding performance. As more sensitive coverage metrics typically produce more coverage data, they profit from faster access and fewer collisions.

FuzzTastic is a tool for obtaining more accurate coverage reports [12]. The combination of collisions and fuzzers keeping only seeds that lead to new coverage can lead to imprecise coverage reports. In their paper, the FuzzTastic authors quantify this effect and propose an alternative, LLVM-based coverage instrumentation that guarantees more accurate reports.

UnTracer and HeXcite, a generalization to more powerful coverage metrics, show the power of coverage-guided tracing [15, 16]. Coverage-guided tracing *adaptively* enables coverage recording only at the frontier of the currently known coverage, leading to a several orders of magnitude higher fuzzing throughput. In a similar way, we propose adaptive coverage hash functions to reduce the number of hash collisions.

In their work from 2018, Hsu et al. propose to instrument only certain basic blocks for tracking coverage, primarily to reduce the performance impact on the fuzzing process [10]. A side effect of this optimization is that collisions are also reduced. Similar to our last research question (cf. Section 5), this approach is also PUT-adaptive.

7 Conclusions

This preregistered report proposes two mutually dependent lines of work.

First, we propose to experimentally analyze how coverage-based fuzzers, exemplified by AFL++, lose precision while tracking PUT states. Specifically, we plan on analyzing the effect of coverage map collisions and arithmetic inaccuracies (e.g., overflows and range counters). For this reason, we identify the degrees of freedom, and pose four research questions. As a result, we design a set of experiments to run on 12 different systems across 3 instruction set architectures (x86_64, ARM64, and RISC-V), and different benchmark programs.

Second, we outline a plan to construct an *adaptive optimization*. More concretely, we propose to leverage the experimental data and derive optimal fuzzing parameters for a given PUT and hardware configuration. Ideally, we can improve AFL++'s accuracy without losing throughput, thus increasing its utilization across fuzzing campaigns.

Finally, our experimental study may help us identify new problems in coverage tracking, thus leading to qualitative improvements.

Acknowledgments

The authors are greatful for the helpful feedback by the anonymous reviewers.

We also thank the authors and contributors of AFL++ for making it open-source, and continuing to improve it. We recognize Michał Zalewski's seminal practical and theoretical contributions.

The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) [(FFG grant no. 888338)] within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

[1] Alif Ahmed, Jason D. Hiser, Anh Nguyen-Tuong, Jack W. Davidson, and Kevin Skadron. 2021. BigMap: Future-proofing Fuzzers with Efficient Large Maps. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 531–542. https://doi.org/10.1109/DSN48987.2021.00062

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[3] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[4] Pietro Borrello, Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti, Leonardo Querzoni, and Cristiano Giuffrida. 2024. Predictive Context-sensitive Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2024.24113

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

[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[7] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[8] Taras Glek and Jan Hubicka. 2010. Optimizing real world applications with GCC link time optimization. *arXiv preprint arXiv:1010.2196* (2010).

[9] Adrian Herrera, Mathias Payer, and Antony L Hosking. 2023. DatAFLow: Toward a data-flow-guided fuzzer. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–31.

[10] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, Vol. 40.

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[12] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. 2022. FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 75–79. https://doi.org/10.1145/3510454.3516847

[13] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[14] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. 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*. 1393–1403.

[15] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[16] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 351–365. https://doi.org/10.1145/3460120.3484787

[17] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 1–15.

[18] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[19] Hang Xu, Zhi Yang, Xingyuan Chen, Bing Han, and Xuehui Du. 2023. BitAFL: Provide More Accurate Coverage Information for Coverage-guided Fuzzing. In *3rd International Conference on Management Science and Software Engineering (ICMSSE 2023)*. Atlantis Press, 521–530.

[20] Michal Zalewski. 2014. Technical whitepaper for AFL-fuzz. Retrieved June 23, 2024 from https://raw.githubusercontent.com/google/AFL/master/docs/technical_details.txt

Received 2024-06-21; accepted 2024-07-22