Poster: YFuzz: Data-Driven Fuzzing

Yuan Chang
National Taiwan University
Taipei, Taiwan
r08922028@csie.ntu.edu.tw

Chun-Chia Huang
National Taiwan University
Taipei, Taiwan
r12922103@csie.ntu.edu.tw

Tatsuya Mori
Waseda University
Shinjuku, Tokyo, Japan
mori@nsl.cs.waseda.ac.jp

Hsu-Chun Hsiao[*][†]
National Taiwan University
Taipei, Taiwan
hchsiao@csie.ntu.edu.tw

Abstract

Code coverage is an effective objective for guiding fuzzers to explore code and identify bugs, and it has been a key factor in the success of greybox fuzzing. However, code coverage has a critical limitation: coverage-guided fuzzers can miss bugs even when the associated code is covered. This limitation arises because merely executing the associated code is often insufficient to trigger a bug; specific conditions are usually also required. These conditions are not fully captured by code coverage, which focuses only on whether the code was executed.

To address this problem, we propose a new objective: value state coverage, an additional dimension in coverage metrics that is orthogonal to code coverage. Value state is a combination of the values assigned to program variables and the order of their assignment, and by measuring the coverage of value states, we can guide a fuzzer to explore the triggering conditions of bugs. We also introduce *Data-Driven Fuzzing*, a novel fuzzing technique that focuses on value state coverage, and utilizes security-related variables, mutation strategies, and extreme values captured at runtime to effectively discover bugs. We implemented our approach in a prototype fuzzer named *YFuzz*. *YFuzz* has found 12 bugs in programs included in the OSS-Fuzz project, including 4 assigned CVEs, indicating that our approach is effective in finding bugs.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Fuzzing, Value State Coverage, Code Coverage, Vulnerability

ACM Reference Format:
Yuan Chang, Chun-Chia Huang, Tatsuya Mori, and Hsu-Chun Hsiao. 2024. Poster: YFuzz: Data-Driven Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3658644.3691420

[*]Also with Academia Sinica.
[†]This research was supported by grants NSTC-112-2223-E-002-010-MY4, 113-2634-F-002-001-MBK, and NTU-113L7871, 113L900901/113L900902/113L900903.

1 Introduction

Code coverage is one of the most popular and effective techniques in software fuzzing. It provides a measure of progress in exploring the codebase by monitoring which parts of a program's code are executed during testing. By increasing code coverage, fuzzers can systematically explore new execution paths, thereby increasing the potential for discovering bugs. Many state-of-the-art fuzzing techniques and fuzzers have been developed based on the concept of code coverage and use it as guidance to optimize their strategies.

However, a recent study by Böhme et al. [1] pointed out a critical limitation of code coverage: "simply reaching a given branch or statement is often insufficient to trigger a bug." To trigger a bug, two requirements must be met: first, the associated code needs to be executed; second, the triggering conditions of the bug, such as certain variable values being within a particular range, must be satisfied. Code coverage addresses the first requirement by measuring which code has been executed, but the second requirement is not addressed. As a result, coverage-guided fuzzers may overlook bugs even when the associated code is covered, since the triggering conditions might not be satisfied.

Some existing research has tried to address the second requirement using various approaches. These approaches fall roughly into two categories. The first focuses on extracting security-related or vulnerability-related information from the program and optimizing fuzzing in these areas [7]. The second involves defining specific program states, such as call stack, memory values, data dependencies, or variable values, and encouraging the fuzzer to explore these states more thoroughly [2], [5]. The first category prioritizes fuzzing code that is commonly associated with bugs, hoping to satisfy the conditions for triggering them. However, it does not explicitly target the states that satisfy those triggering conditions. As a result, it may still miss bugs due to the lack of awareness of their triggering conditions. The second category often defines program states too broadly, which can make it challenging to focus on the information that truly describes the triggering conditions of bugs, and can also suffer from the state explosion problem.

To address the problem that code coverage cannot fully capture the triggering conditions of bugs, we propose a new dimension to measure the exploration of a program: value state coverage. Value state coverage measures the value space of program variables. By guiding fuzzers to explore this space, they can more directly aim to

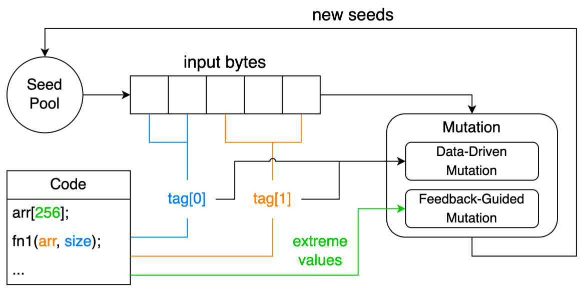


Figure 1: Workflow of *Data-Driven Fuzzing*

satisfy the triggering conditions of bugs. We believe this dimension is critical for finding bugs and has been overlooked by coverage-guided fuzzing in the past. We also introduce *Data-Driven Fuzzing*, a novel fuzzing technique guided by value state coverage. It tracks the value states of two types of program variables—those typically involved in vulnerability—and explores the space of value states of one execution path at a time. By doing so, *Data-Driven Fuzzing* can narrow down the search space, thus avoiding state explosion and focusing on describing the triggering conditions of bugs.

To verify our ideas, we implemented a prototype fuzzer of *Data-Driven Fuzzing*, called *YFuzz*. *YFuzz* first uses static analysis to identify the program variables that may lead to security vulnerabilities, which we refer to as security-related variables, and instruments them to collect their values at runtime. We then design the mutation strategies and seed scheduling to explore a variety of values of these security-related variables. To effectively explore the values of these variables, we utilize lightweight data flow analysis to locate the input bytes that can affect the values of the security-related variables, thereby reducing the input space for mutation. Additionally, *YFuzz* infers the extreme values of the security-related variables that might cause out-of-bound memory access during runtime and provides these values as feedback to the mutator. This step helps the mutator to trigger bugs more efficiently.

In our preliminary experiment, *YFuzz* discovered 12 vulnerabilities in real-world programs that had been extensively fuzzed, with 4 of these vulnerabilities assigned CVE numbers. These initial results suggest that our approach are promising in finding bugs.

The contributions of this poster are:

- We propose a new dimension in coverage to guide fuzzers to satisfy the triggering conditions of bugs by exploring the value state of program variables.
- We introduce novel seed scheduling and mutation strategies for exploring the value state space.
- The preliminary results of our prototype, *YFuzz*, demonstrate the potential of value state coverage for finding bugs.

2 Value State Coverage

We define a *value state* as the sequence of values assigned to program variables in the order of their assignment along an execution path. For example, consider an execution path where variable A is assigned the value 1, then variable B is assigned the value 2, followed by variable A being assigned the value 3. The value state of this execution path can be described as [A=1, B=2, A=3].

Value state coverage can describe the state of a program that code coverage cannot capture regarding the triggering conditions of bugs. This concept is illustrated by the example code shown below. In this example, there is a buffer overflow bug due to an incorrect size check, and the bug can only be triggered when the variable `size` is in the range of 91 to 99. Code coverage alone is insufficient to capture the triggering conditions of this bug because the program state with `size` in the range of 0 to 90 and `size` in the range of 91 to 99 yield the same code coverage. As a result, code coverage cannot effectively guide fuzzers to satisfy the triggering conditions, and a scheduling strategy optimized for code coverage may encourage fuzzers to explore other parts of the code once the `read` statement is reached. This could cause the fuzzers to miss the bug. However, value state coverage can differentiate these two states because it considers the value of the variable `size`. It can thus guide the fuzzer to explore the value space of `size` before moving on to other parts of the code.

```
unsigned size;
char buf[90];
if (size < 100) {
  read(0, buf, size);  // buffer overflow
}
```

Value state coverage is an orthogonal dimension to code coverage, and both can be used by a fuzzer to explore code and satisfy the triggering conditions of bugs. In this poster, we focus exclusively on value state coverage; increasing code coverage is beyond the scope of this work.

3 Data-Driven Fuzzing

Inspired by Fioraldi et al.'s work [2], the intuition behind *Data-Driven Fuzzing* is that the space of value states for an execution path can be viewed as multiple subspaces divided by the triggering conditions of bugs. Covering one value state within a subspace is sufficient to trigger the bug associated with that subspace. Thus, *Data-Driven Fuzzing* is designed to increase the diversity of the value states along a given path, thereby increasing the chance of exploring different subspaces and triggering bugs.

Data-Driven Fuzzing can be broken down into four components: *Identify Security-related Variables*, *Locate the Corresponding Input Bytes*, *Data-Driven Mutation*, and *Feedback-Guided Mutation*. The workflow of *Data-Driven Fuzzing* is depicted in Figure 1.

3.1 Identify Security-related Variables

The space of value state can become infeasible to explore when too many program variables are considered. To avoid state explosion and focus on triggering bugs, our approach considers only two types of variables that commonly lead to vulnerabilities when calculating value state coverage. These types of variables are selected based on our real-world experience in software exploitation.

The first type of security-related variable is size-related variables. These include the size arguments of memory and I/O operations, such as those used in input, output, memory access, and memory allocation. Incorrect values in these variables can lead to common overflow-based bugs and memory allocation errors. The second type is pointer-related variables. These include pointers and variables that control arrays, such as array indices. Incorrect values in

Table 1: Average # of unique value states explored in 48 hours.

Target	YFuzz	AFL++
exiv2	17,365,987 (+64%)	10,586,739
sndfile-convert	129,274,865 (+912.3%)	12,770,007
xmllint	31,892,762 (+36.9%)	23,303,361

these variables can result in out-of-bound memory access, causing unexpected behavior or crashing the program.

We identify the security-related variables through static analysis and instrument them to gather information at runtime.

3.2 Locate the Corresponding Input Bytes

To increase the value diversity of the security-related variables and thereby explore the space of value states for an execution path, we locate the bytes in the input that affect the values of these variables without changing the execution path. This ensures that the seeds generated by mutation mostly follow the same execution path.

To locate these bytes without introducing excessive overhead, we use a lightweight taint analysis technique similar to Taint Inference in GERYONE [4]. First, the target program is executed with an input, and both code coverage and value state coverage are recorded. Next, we mutate each byte of the input one by one, comparing the new code coverage and value state coverage with the original ones. If only the value state coverage changes, indicating that the byte affects the value of some security-related variables without changing the execution path, then the byte is labeled. Each byte is mutated multiple times before labeling to verify its effect on code coverage and value state coverage. Finally, the labeled bytes are grouped into tags, with each tag representing the bytes that affect the value of a security-related variable on the execution path.

3.3 Data-Driven Mutation

To mutate an input, we randomly select a tag each time before reaching the energy limit, then mutate only the labeled bytes associated with that tag. This approach ensures that the mutated seeds follow the same execution path, allowing us to explore different value states along the path and cover more subspaces.

3.4 Feedback-Guided Mutation

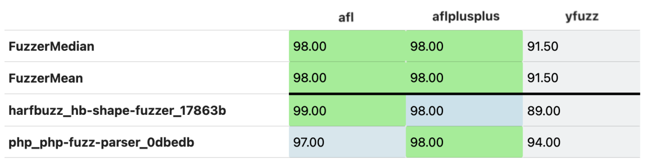
The value range of variables that trigger bugs can be narrow. In our real-world observations, overflow-based bugs often have a limited range of values that can trigger them, typically near the boundaries of the memory they access. For example, off-by-one byte overflow bugs are only triggered when the variable takes on a specific value, usually just one step beyond the buffer size.

To aid the fuzzers in reaching these value ranges, we capture the boundaries of the security-related variables and use them as a guide for mutation. This helps the fuzzers focus their exploration on the value states near the triggering conditions, which are the boundaries of subspaces, making it more likely to trigger bugs.

4 Preliminary Results

YFuzz discovered 12 bugs in 4 programs: Exiv2, libxml2, libsndfile, and openjpeg, which have been extensively fuzzed in the OSS-Fuzz project. 4 of these bugs have been assigned CVE numbers: CVE-2021-3482, CVE-2021-29623, CVE-2021-3537, and CVE-2021-3575. While further evaluation is still in progress, the ability to find new bugs in programs already included in the OSS-Fuzz project demonstrates the capability of *YFuzz* in identifying new vulnerabilities.

As shown in Table 1, the value state coverage of *YFuzz* is consistently higher when compared to AFL++ [3]. Although increasing code coverage is not our goal, our early evaluation using FuzzBench [6] indicates that *YFuzz* tends to achieve lower code coverage compared to AFL [8] and AFL++, as shown in Fig. 2.



	afl	aflplusplus	yfuzz
FuzzerMedian	98.00	98.00	91.50
FuzzerMean	98.00	98.00	91.50
harfbuzz_hb-shape-fuzzer_17863b	99.00	98.00	89.00
php_php-fuzz-parser_0dbedb	97.00	98.00	94.00

Figure 2: Results of code coverage using FuzzBench

5 Conclusion

This poster proposed value state coverage, a new dimension for fuzzing that addresses the limitations of code coverage. Our approach focuses on exploring the value states of security-related program variables, guiding fuzzers to meet the triggering conditions of bugs. This method enhances the ability to uncover vulnerabilities that may be missed by relying solely on code coverage. The preliminary results demonstrate the potential of our approach, although some challenges remain. For instance, the code coverage achieved by our approach tends to decrease, and an execution path with too many security-related variables may still lead to a state explosion.

Value state coverage is promising for finding bugs previously missed by coverage-guided fuzzing. Many fuzzing concepts and techniques developed based on code coverage, such as seed scheduling, mutation strategies, and hybrid fuzzing, could be adapted and applied to value state coverage in the future. Additionally, comparing, integrating, and balancing value state coverage with code coverage offers promising opportunities for future research. We encourage researchers to study this new dimension of coverage and explore the triggering conditions of various bugs.

References

[1] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *ICSE*.

[2] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *USENIX Security*.

[3] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *WOOT*.

[4] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: data flow sensitive fuzzing. In *USENIX Security*.

[5] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with Data Dependency Information. In *EuroS&P*.

[6] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *ESEC/FSE*.

[7] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *NDSS*.

[8] Michał Zalewski. 2016. *American Fuzzy Lop*. Retrieved July 31, 2024 from https://lcamtuf.coredump.cx/afl/