Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University
[†]{jwang131,wsong008}@ucr.edu, {heng,csong}@cs.ucr.edu [‡]yd375@cornell.edu

Abstract

Coverage-guided greybox fuzzing has become one of the most common techniques for finding software bugs. Coverage metric, which decides how a fuzzer selects new seeds, is an essential parameter of fuzzing and can significantly affect the results. While there are many existing works on the effectiveness of different coverage metrics on software testing, little is known about how different coverage metrics could actually affect the fuzzing results in practice. More importantly, it is unclear whether there exists one coverage metric that is superior to all the other metrics. In this paper, we report the first systematic study on the impact of different coverage metrics in fuzzing. To this end, we formally define and discuss the concept of *sensitivity*, which can be used to theoretically compare different coverage metrics. We then present several coverage metrics with their variants. We conduct a study on these metrics with the DARPA CGC dataset, the LAVA-M dataset, and a set of real-world applications (a total of 221 binaries). We find that because each fuzzing instance has limited resources (time and computation power), (1) each metric has its unique merit in terms of flipping certain types of branches (thus vulnerability finding) and (2) there is no grand slam coverage metric that defeats all the others. We also explore combining different coverage metrics through cross-seeding, and the result is very encouraging: this pure fuzzing based approach can crash at least the same numbers of binaries in the CGC dataset as a previous approach (Driller) that combines fuzzing and concolic execution. At the same time, our approach uses fewer computing resources.

1 Introduction

Greybox fuzzing is a state-of-the-art program testing technique that has been widely adopted by both mainstream companies such as Google [45] and Adobe [47], and small startups (e.g., Trail of Bits [48]). In the DARPA Cyber Grand Challenge (CGC), greybox fuzzing has been demonstrated to be more effective compared to other alternatives such as symbolic execution and static analysis [8, 15, 34, 37, 39].

Greybox fuzzing generally contains three major stages: seed scheduling, seed mutation, and seed selection. From a set of seed inputs, the seed scheduler picks the next seed for testing. Then, more test cases are generated based on the scheduled seeds through mutation and crossover in the seed mutation stage. Finally, test cases of good quality are selected as new seeds to generate more test cases in the future rounds of fuzzing. Among these stages, seed selection is the most important one as it differentiates greybox fuzzing from blackbox fuzzing and determines the goal of the fuzzer. For example, when the goal is to improve coverage, we use a coverage metric to evaluate the quality of a test case, and when the goal is to reach a particular code point, we can use distance to evaluate the quality of a test case [2]. Note that although previous studies [14, 17] have shown that better coverage of test suite is not directly related to a better quality of the tested software, the observation that under-tested code is more likely to have bugs still holds. For this reason, coverage-guided greybox fuzzing still works very well in practice.

Although various techniques have been proposed to improve greybox fuzzing at the seed scheduling stage [2, 3, 27, 29] and the seed mutation stage [21, 28, 29, 37, 54], very few efforts focus on improving seed selection. HonggFuzz [40] only counts the number of basic blocks visited. AFL [38] utilizes an improved branch coverage that also counts how many times a branch is visited. Angora [7] further extends the branch coverage to be context-sensitive. More importantly, many critical questions about coverage metrics remain unanswered.

First, *how do we uniformly define the differences among different coverage metrics?* Coverage metrics can be categorized into two major categories: code coverage and data coverage. Code coverage metrics evaluate the uniqueness among test cases at the code level, such as line coverage, basic block coverage, branch/edge coverage, and path coverage. Data coverage metrics, on the other hand, try to distinguish test cases from a data accessing perspective, such as memory addresses, access type (read or write), and access sequences. While many new metrics have been proposed individually in recent works,

there is no systematic and uniform way to characterize the differences among them. Apparently, different coverage metrics have very distinct capability of differentiating test cases, which we refer to as *sensitivity*. For example, block coverage could not tell the difference between visits to the same basic block from different preceding blocks, while branch coverage can. Therefore, branch coverage is more sensitive than block coverage. A systematic and formal definition of *sensitivity* is essential as it can not only tell the differences among current metrics but also guide future research to propose more metrics.

Second, *is there an optimal coverage metric that outperforms all the others in coverage-guided fuzzing?* Although sensitivity provides us a way to compare the capability of two coverage metrics in discovering interesting inputs, a more sensitive coverage metric does not always lead to better fuzzing performance. More specifically, fuzzing can be modeled as a multi-armed bandit (MAB) problem [51] where each stage (seed selection, scheduling, and mutation) has multiple choices, and the ultimate goal is to *find more bugs* with a limited time budget. A more sensitive coverage metric may select more inputs as seeds, but the fuzzer may not have enough time budget to schedule all the seeds or mutate them sufficiently. Implementation details such as how coverage is actually measured can further complicate this problem. For instance, a previous study [12] has shown that hash collisions could reduce the actual sensitivity of a coverage metric. A systematic evaluation is essential to understand the relationship between sensitivity and fuzzing performance better.

Third, *is it a good idea to combine different metrics during fuzzing?* Hypothetically, if different coverage metrics have their own merits during fuzzing, then it would make sense to combine them so that different metrics could contribute differently. This question is also crucial as it motivates different thinking and may lead to strategies for improving fuzzing.

To answer the questions mentioned above, we conduct the first systematic study on the impact of coverage metrics on the performance of coverage-guided fuzzing. In particular, we formally define and discuss the concept of *sensitivity* to distinguish different coverage metrics. Based on the different levels of sensitivity, we then present several representative coverage metrics, namely "basic branch coverage," "context-sensitive branch coverage," "n-gram branch coverage," and "memory-access-aware branch coverage," as well as their variants. Finally, we implement six coverage metrics in a widely-used greybox fuzzing tool, AFL [38], and evaluate them with large datasets, including the DARPA CGC dataset [4], the LAVA-M dataset [42], and a set of real-world binaries. The highlighted findings are:

- Many of these more sensitive coverage metrics indeed lead to finding more bugs as well as finding them significantly faster.
- Different coverage metrics often result in finding different sets of bugs. Moreover, at different times of the whole fuzzing process, the best performer may vary. As a result, there is no grand slam coverage metric that can beat others.
- A combination of these different metrics can help find more bugs and find them faster. Notably, using less computing resources, a combination of fuzzers with different coverage metrics is able to find at least the same amount of bugs in the CGC dataset as Driller, a hybrid fuzzer augmented AFL with concolic execution did [35].

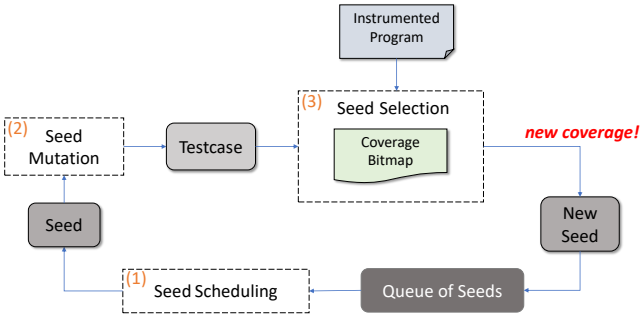

Figure 1: The workflow of coverage-guided greybox fuzzing.

To facilitate further research on this topic, we have made the source code and dataset available at https://github.com/bitsecurerlab/afl-sensitive.

2 Background

In this section, we provide the background information about coverage-guided greybox fuzzing, with a focus on the seed selection.

2.1 Coverage-guided Greybox Fuzzing

Coverage-guided greybox fuzzing generates inputs (or test cases) incrementally via a feedback loop. Specifically, there are three main stages, as illustrated in Figure 1. (1) *Seed scheduling*: a seed is picked from a set of seeds according to the scheduling criteria. (2) *Seed mutation*: within a limited time budget, new test cases are generated by performing various mutations on the scheduled seed. (3) *Seed selection*: each generated test case is fed to the program under test and evaluated based on the coverage metric; if the testcase leads to new coverage, it will be selected as a new seed. As this feedback loop continues, more coverage will be reached, and hopefully, a test case will trigger a bug.

2.2 Seed Selection

A seed selection strategy determines the trend and speed of the evolution of the fuzzing process. Essentially, a good seed selection strategy needs to solve two essential problems: (1)

how to collect coverage information and (2) how to measure the quality of test cases.

Coverage Information Collection. AFL instruments the program under test to collect and compute the coverage. There are two instrumentation approaches. When the source code of the program under test is available, a modified `Clang` compiler is used to insert the coverage calculation logic into the compiled executable at assembly level (normal mode) or intermediate representation level (fast mode). When the source code is not available, a modified user-mode QEMU is used to run the binary code of the tested program directly, and the coverage calculation logic is inserted during the binary translation phase. VUzzer [29] uses PIN [41] to perform binary instrumentation to collect the information. HonggFuzz [40] and kAFL [31] use hardware branch tracers like Intel Process Tracing (PT) to collect coverage information and DigTool [25] uses a hypervisor to collect coverage information from OS kernels.

Test Case Measurement. The quality of test cases is measured by leveraging coverage metrics. HonggFuzz [40] and Vuzzer [29] use basic block coverage metric that tracks visits of basic blocks. AFL [38] uses an improved branch coverage metric that could differentiate the visits to the same block from different preceding blocks. LibFuzzer [43] can use either block coverage or branch coverage. A more recent work Angora [7] extends the branch coverage metric with a calling context. Another important aspect is how the metric is really measured. Since coverage is measured during the execution of each test case, fuzzers usually prefer simpler implementations to improve the fuzzing throughput. For example, AFL identifies a branch using a simple hash function (Equation 1). Unfortunately, this approximation could reduce the effective sensitivity of a coverage metric due to hash collisions [12].

3 Sensitivity and Coverage Metrics

In this section, we formally define and discuss the concept of the sensitivity of a coverage metric. Accordingly, we present several coverage metrics that have different sensitivities.

3.1 Formal Definition of Sensitivity

When comparing different coverage metrics, a central question is "is metric A better than metric B?" To answer this question, we need to take a look at how a mutation-based greybox fuzzer finds a bug. In mutation-based greybox fuzzing, a bug triggering test case is reached via a chain of mutated test cases. In this process, if an intermediate test case is deemed "uninteresting" by a coverage metric, the chain will break and the bug triggering input may not be reached. Based on this observation, we decide to define *sensitivity* as a coverage metric's ability to preserve such mutation chains.

To formally describe this concept, we first need to define a coverage metric as a function $\mathcal{C} : (\mathcal{P} \times \mathcal{I}) \to \mathcal{M}$, which produces a measurement $M \in \mathcal{M}$ when running a program $P \in \mathcal{P}$ with an input $I \in \mathcal{I}$. Given two coverage metrics C_i and C_j, C_i is "more sensitive" than C_j, denoted as $C_i \succ C_j$, if

(1) $\forall P \in \mathcal{P},\ \forall I_1, I_2 \in \mathcal{I},\ C_i(P,I_1) = C_i(P,I_2) \to C_j(P,I_1) = C_j(P,I_2)$, and

(2) $\exists P \in \mathcal{P},\ \exists I_1, I_2 \in \mathcal{I},\ C_j(P,I_1) = C_j(P,I_2) \wedge C_i(P,I_1) \neq C_i(P,I_2)$

The first condition means, for any program P, if any two inputs I_1 and I_2 produce the same coverage measurement using C_i; then they must produce the same measurement using C_j, i.e., C_j is always not more discriminative than C_i. The second condition means, there exists at least a program P such that two inputs I_1 and I_2 would produce the same measurement using C_j but different measurements using C_i, i.e., C_i can be more discriminative than C_j.

3.2 Coverage Metrics

In this subsection, we introduce several coverage metrics and their approximated measurement. Then we compare their sensitivity.

Branch Coverage Branch coverage is a straightforward yet effective enhancement over block coverage, which is the most basic one that can only tell which code block is visited. By involving the code block preceding the currently visited one, branch coverage can differentiate the visits of the same code block from different predecessors. Branch here means an edge from one code block to another one.

Ideally, branch coverage should be measured as a tuple (*prev_block*, *cur_block*), where *prev_block* and *cur_block* stand for the previous block ID and the current block ID, respectively. In practice, branch coverage is usually measured by hashing this tuple (as key) into a hash table (e.g., a `hit_count` map). For example, the state-of-the-art fuzzing tool AFL identifies a branch as:

$$block\_trans = (prev\_block << 1) \oplus cur\_block \qquad (1)$$

where branch ID is calculated as its runtime address. The *block_trans* is then used as the key to index into a hash map to access the `hit_count` of the branch, which records how many times the branch has been taken. After a test case finishes its execution, its coverage information is compared with the global coverage information (i.e., a global `hit_count` map). If the current test case has new coverage, it will be selected as a new seed.

Although branch coverage is widely used in mainstream fuzzers, its sensitivity is low. For instance, considering a

branch within a function that is frequently called by the program (e.g., `strcmp`). When the branch is visited under different calling contexts, branch coverage will not be able to distinguish them.

N-Gram Branch Coverage After incorporating one preceding block in branch coverage, it is intuitive to incorporate more preceding basic blocks as history into the current basic block. We refer to this coverage metric as *n-gram branch coverage*, where n is a configurable parameter that indicates how many continuous branches are considered as one unit, and any changes of them will be distinguished. When $n = 0$, n-gram branch coverage is reduced to block coverage. On the opposite extreme, when $n \to \infty$, n-gram branch coverage is equivalent to path coverage because it incorporates all preceding branches into the context and any change in the execution path will be treated differently.

Ideally, n-gram branch coverage should be measured as a tuple $(block_1, \ldots, block_{n+1})$. For efficiency, we propose to hash the tuple as a key into the `hit_count` map as $(prev\_block\_trans << 1) \oplus curr\_block\_trans$, where

$$prev\_block\_trans = (block\_trans_1 \oplus \cdots \oplus block\_trans_{n-1}) \tag{2}$$

In other words, we record the previous $n-1$ block transitions (calculated as in Equation 1) and XOR them together, left shift 1 bit, and then XOR with the current block transition.

Now an interesting question is: *what is the best value for n?* If n is too small, it might be almost the same as branch coverage. If n is too large, it may cause seed explosion (a similar phenomenon as path explosion). Fuzzing progress would be even slower due to the enormous amount of seeds.

To answer this question empirically, we adapt AFLFast to n-gram branch coverage where n is set to 2, 4, and 8. We will evaluate these settings in §4.

Context-Sensitive Branch Coverage A function lies between a basic block and a path with respect to the granularity of code. Therefore, calling context is another important piece of information that can be incorporated as part of the coverage metric, which allows a fuzzer to distinguish the same code executed with different data. We refer to this coverage metrics as "context-sensitive coverage metric."

Ideally, context-sensitive branch coverage metric should be measured as a tuple (*call_stack*, *prev_block*, *curr_block*). For efficiency, we define a calling context *call_ctx* as a sequence of program locations where function calls are made in order:

$$call\_ctx = \begin{cases} 0 & \text{initial value} \\ call\_ctx \oplus call\_next\_insn & \text{if call} \\ call\_ctx \oplus ret\_to\_insn & \text{if ret} \end{cases} \tag{3}$$

Then the key-value pair stored in the bitmap will be now calculated as $call\_ctx \oplus block\_trans$.

Initially, the calling context value *call_ctx* is set to 0. Then during the program execution, when encountering a `call` instruction, we XOR the current *call_ctx* with the instruction's position immediately next to the call instruction and store the result in *call_ctx*. Similarly, when encountering a `ret` instruction, we XOR the current `call_ctx` with the return address. In this way, a small value *call_ctx* efficiently accumulates function calls made in sequence and eliminates function calls that have returned.

Memory-Access-Aware Branch Coverage In addition to leveraging extra control flow information as stated above, data flow information also deserves to be considered. Based on the intuition that a primary focus of fuzzing is to detect memory-corruption vulnerabilities, memory access information can be of great help in measuring coverage. Fundamentally, memory corruption exhibits an erroneous memory access behavior. Therefore, it makes sense to select seeds that exhibit distinct memory access patterns.

In general, this memory-access aware coverage metric is more sensitive than branch coverage. Because if a new test case reaches a branch that has been covered by prior test cases, but at least one new memory location is accessed, this test case will still be considered as "interesting" in memory-access aware coverage metric and kept as a seed.

There can be many ways to characterize memory access patterns. In this paper, we investigate one design option. We instrument memory access operations of the program under test, and define each memory access as a tuple (*type*, *addr*, *block_trans*), where *type* represents access type (read or write), *addr* is the accessed memory location, and *block_trans* means after which branch this memory access is performed.

For efficiency, we propose to calculate the hash key as $(block\_trans \oplus mem\_ac\_ptn)$, where

$$mem\_ac\_ptn = \begin{cases} mem\_addr & \text{if read} \\ mem\_addr + half\_map\_size & \text{if write} \end{cases} \tag{4}$$

Note that reads are distinguished from writes by allocating their keys to different half regions of the map.

Since memory corruption is mainly caused by memory writes, it is meaningful to investigate a variant of memory access coverage: "memory-write-aware branch coverage." That is, we only instrument and record memory writes, but not reads, making it less sensitive.

3.3 Sensitivity Lattice

Obviously, $\succ$ is a strict partial order, because it is asymmetric (if $C_1 \succ C_2$, by no means $C_2 \succ C_1$), transitive (if $C_1 \succ C_2$ and $C_2 \succ C_3$, then $C_1 \succ C_3$), and irreflexive ($C_i \succ C_i$ is not possible). However, it is not a total order, because it is possible that two metrics are not comparable.

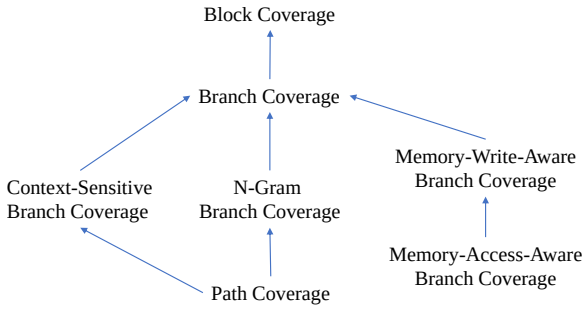

Figure 2: Sensitivity Lattice for Coverage Metrics

As a result, we can draw a sensitivity lattice for the coverage metrics discussed above. Figure 2 shows this lattice. Block coverage is the least sensitive metric, compared to the rest, so it appears on the top. Immediately below is branch coverage. It is more sensitive than block coverage. Then below branch coverage are the three coverage metrics that incorporate different extra information on top of branches.

However, there is no direct comparison among these three coverage metrics, because each of them extends branch coverage in different dimensions: context-sensitive branch coverage incorporates calling context, n-gram branch coverage integrates n-1 preceding block transitions, and memory-access-aware branch coverage includes memory accesses. We can always construct a program and two inputs, such that the same coverage measurement is produced for one metric, but two different coverage measurements are produced for another.

For different values of n in n-gram branch coverage, i-gram is more sensitive than j-gram if $i > j$. Ultimately, path coverage is more sensitive than n-gram branch coverage and context-sensitive branch coverage.

Interestingly enough, we cannot compare path coverage with either memory-access-aware branch coverage or memory-write-aware branch coverage. Path coverage is not necessarily more sensitive because two inputs may follow the same path but exhibit different memory access patterns.

It is noteworthy that the coverage metrics presented here are a few representative ones but are by no means complete. We hope this work can stimulate research on developing more coverage metrics and obtaining a deeper understanding of their impact.

4 Evaluation

To answer the research questions raised in §1, we implemented all the coverage metrics mentioned in §3 except the basic branch coverage, which is already implemented in AFL. We then conducted comprehensive experiments to evaluate the performance of different coverage metrics. Moreover, to better understand how different coverage metrics working together could affect fuzzing; we also evaluate the combination of them.

Table 1: real-world applications used in evaluation.

Applications	Version	Applications	Version
objdump+binutils	2.29	readelf+binutils	2.29
strings+binutils	2.29	nm+binutils	2.29
size+binutils	2.29	file	5.32
gzip	1.8	tiffset+tiff	4.0.9
tiff2pdf+tiff	4.0.9	gif2png	2.5.11
info2cap+ncurses	6.0	jhead	3.0

4.1 Implementation

In this study, since our primary goal is to fuzz binaries without source code, we choose to add our instrumentation based on user-mode QEMU. For instance, for context-sensitive branch coverage, we instrument `call` and `ret` instructions to calculate calling context, and for memory-access-aware branch coverage, we instrument memory reads and writes. For n-gram branch coverage, we use a circular buffer to store the last n-block transitions, for efficient n-gram calculation.

For convenience, in the remainder of this paper, we use the following abbreviations to represent different metrics: `bc` represents the existing branch coverage in AFL, `ct` represents context-sensitive branch coverage, `mw` is short for memory-write-aware branch coverage, and `ma` represents memory-access-aware branch coverage. For n-gram branch coverage, we choose to implement three versions: 2-gram, 4-gram and 8-gram, and use `n2`, `n4`, and `n8` for their abbreviations.

Furthermore, we adopted the seed scheduling of AFLFast [3] in our implementation. Since AFLFast inclines to allocate more fuzzing time on newly generated seeds, different coverage metrics will make a greater impact on fuzzing performance.

4.2 Dataset

We collect binaries from DARPA Cyber Grand Challenge (CGC) [4]. There are 131 binaries from CGC Qualifying Event (CQE) and 74 binaries from CGC Final Event (CFE), and thus 205 ones in total. These binaries are carefully crafted by security experts to utilize different kinds of techniques (e.g., complex I/O protocols and input checksums) and embed vulnerabilities in various ways (e.g., buffer overflow, integer overflow, and use-after-free) to comprehensively evaluate various vulnerability discovery techniques.

We also choose the LAVA-M Dataset [11, 42], which consists of four GNU coreutils programs (`base64`, `md5sum`, `uniq`, and `who`) for evaluation. Each of these binaries is injected with a large number of specific vulnerabilities. As a result, we treat these injected vulnerabilities as ground truth and use them to evaluate different coverage metrics.

In addition to the two datasets above, we also manage to collect 12 real-world applications with their latest versions (Table 1) and assess the performance of different coverage metrics in practice with them.

4.3 Experiment Setup

Our experiments are conducted on a private cluster consisting of a pool of virtual machines. Each virtual machine has a Ubuntu 14.04.1 operating system equipped with 2.3 GHz Intel Xeon processor (24 cores) and 30GB of RAM. As fuzzing is a random process, we followed the recommendations from [20] and performed each evaluation several times for a sufficiently long period.

The tests are mainly focused on the CGC dataset. Specifically, each coverage metric is tested with every binary of the CGC dataset in the dataset using two fuzzing instances for 6 hours (i.e., similar to one instance running 12 hours). We chose this fuzzing time because almost all of the bugs found by fuzzer in CQE and CFE were reported within the first six hours. Moreover, in order to take the randomness of fuzzing into account, each test is performed ten times. The total evaluation time is around 60 days. For binaries with initial sample inputs, we utilized them as initial seeds; otherwise, we used an empty seed.

For the LAVA-M dataset, we tested each coverage metric separately for 24 hours and three times. We used the seed inputs provided by this benchmark and dictionaries of constants extracted from the binary as suggested in [44]. For the real-world dataset, we tested each coverage metric for 48 hours, with two fuzzing instances, and for six times. We used the example inputs from AFL as seeds whenever possible; otherwise with an empty seed.

4.4 Evaluation Metrics

To answer the question of whether there is an optimal coverage metric, we propose three metrics to quantify the experimental results and evaluate the performance of the presented coverage metrics:

- **Unique crashes**. A unique crash during fuzzing implies that a potential bug of the binary has been found. For the CGC dataset, each binary is designed to have a single vulnerability, so we did not perform any crash deduplication. For the LAVA-M dataset, each bug is assigned with a unique ID which is used for crash deduplication. For the real-world dataset, we utilize the hash of each crash's backtrace for deduplication.

- **Time to crash**. This metric indicates how fast a given binary can be crashed by a fuzzer and is mainly for the CGC dataset. Because a CGC binary only has one vulnerability, this metric can be used to measure the efficiency of fuzzing with different coverage metrics.

- **Seed count**. A more sensitive coverage metric is more likely to convert a testcase into a seed, and thus the number of unique seeds may be larger. Therefore, this metric quantifies the sensitivity of each coverage metric in a practical sense.

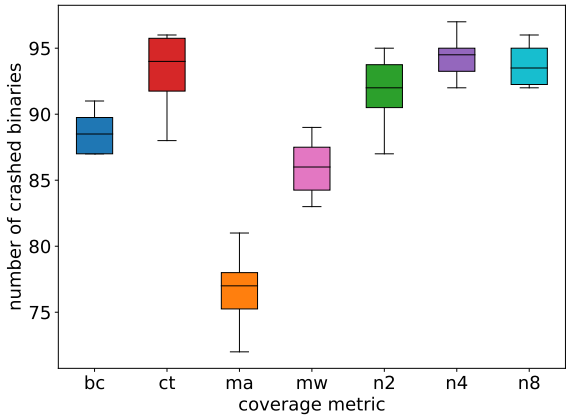

Figure 3: Number of crashed CGC binaries. Because each binary only has on vulnerability, this number is equivalent to the total number of unique crashes.

4.5 Comparison of Unique Crashes

CGC dataset Figure 3 summarizes the number of crashed CGC binaries for each coverage metric across ten rounds of trials. Overall, the baseline metric `bc` crashed about 89 binaries on average and 91 binaries at most. Except for `ma` and `mw`, all other more sensitive coverage metrics (`ct, n2, n4, n8`) outperform `bc`. This result is encouraging: sensitivity does play an important role in finding crashes. However, as demonstrated by `mw` and `ma`, too much sensitivity could also have a negative impact on fuzzing performance. The reason is, more sensitive metrics will select more test cases as seeds (§4.7); when the time budget is limited, each seed will get less time to mutate or not get scheduled at all.

Next, we investigated each coverage metric's ability to trigger individual bug/crash – is there any bug that is only triggered by one or a subset of the evaluated metrics but not the rest? To answer this question, we conducted a pairwise comparison on crashed binaries (Table 2). For each pair of coverage metrics i (in the row) and j (in the column), we first count the number of binaries that were only crashed by i but not by j, denoted as the number after the "/". Since such

Table 2: Pairwise comparisons (row vs. column) of uniquely crashed CGC binaries.

	bc	ct	ma	mw	n2	n4	n8	others
bc	0/0	**0/6**	**0/15**	**0/11**	**0/6**	**0/6**	**0/5**	**0/2**
ct	**9/13**	0/0	9/23	10/15	6/12	3/6	4/8	1/3
ma	**2/3**	3/4	0/0	2/3	4/6	4/5	2/3	1/1
mw	**6/8**	2/5	0/12	0/0	3/8	2/7	3/5	0/2
n2	**4/4**	0/3	7/16	4/9	0/0	0/2	0/2	0/0
n4	**9/12**	3/5	12/23	8/16	8/10	0/0	0/5	0/1
n8	**9/10**	6/6	13/20	10/13	7/9	2/4	0/0	0/0
all	**19/21**	10/14	20/33	19/24	18/23	11/15	9/16	110

Table 3: Number of unique bugs found by different coverage metrics on the LAVA-M dataset

	bc	ct	ma	mw	n2	n4	n8	Listed
base64	45	45	44	45	45	45	45	44
md5sum	54	58	35	43	59	58	51	57
uniq	29	29	29	20	29	29	29	28
who	261	255	301	231	166	159	299	2136

differences could be caused by randomness, we conducted a second experiment focusing on the impact of sensitivity. Specifically, during fuzzing, we recorded the chain of seeds that led to each crashing test case. Each chain starts with the initial seed and ends with the crashing test case. Afterward, for each pair of coverage metrics (i, j), we checked whether each seed along the chain selected by i would also be selected by j as seed, without any additional mutation (i.e., fuzzing). In this process, we also discarded additional sensitivity (non-binary `hit_count`) and insensitivity (key collision) introduced in implementation. The result is denoted as the number before the "/" in each cell of Table 2. For example, entry (`ct`, `bc`) indicates that there were 13 binaries crashed by `ct` but not by `bc`, within which 9 crashes have at least one seed along the crashing chains that will be dropped by `bc`. Similarly, entry (`bc`, `ct`) indicates that 6 binaries crashed by `bc` are not crashed by `ct`, of which however none of the seeds along the crashing chain will be dropped by `ct`. Besides, for a metric k, entry (`all`, k) indicates the number of binaries crashed by at least one of the other coverage metrics but not by k and entry (k, `others`) indicates the number of binaries only crashed by k but not by any other coverage metrics. Finally, entry (`all`, `others`) indicates the number of binaries crashed by at least one of all the seven coverage metrics.

We can see that the difference between any two coverage metrics is considerable. More importantly, there is no single winner that beats everyone else. Even for `ma`, although it crashes the smallest amount of binaries in total, it contributes 2 unique crashed binaries beyond `bc`, and 3, 2, 4, 4, and 2 unique crashed binaries beyond `ct`, `mw`, `n2`, `n4`, and `n8` respectively, of which the crashes have at least one seed along the crashing chains that will be dropped by the other metric. In other words, every coverage metric can make its own and unique contribution. This observation further motivates us to study the combination of different coverage metrics. We will discuss more in §4.8.

LAVA-M dataset Table 3 summarizes the bugs found on LAVA-M dataset by different coverage metrics, while the last column represents the number of bugs listed by LAVA authors. Compare to the CGC dataset, the LAVA-M dataset is not very suitable for our goal. In particular, most injected bugs are protected by a magic number, which is very hard to be solved by random mutation and cannot reflect unique abilities of different coverage metrics. Although we have followed the suggestions from [44] and used dictionaries of constant

Table 4: Number of unique crashes found by different coverage metrics in the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
gif2png	4	4	3	4	5	4	4
info2cap	1446	1063	481	99	568	933	943
objdump	–	–	–	–	1	1	–
size	–	1	–	–	1	1	1
nm	–	1	–	1	–	–	1

(magic) numbers extracted from the binary, we still cannot rule out the differences caused by not being able to solve the magic number. For binary `base64`, `md5sum`, and `uniq`, the difference between different coverage metrics is small, except for `ma` in `md5sum` and `mw` in `uniq`. For binary `who`, it is surprising that in addition to `n8`, `ma` also finds much more unique bugs than `bc` and other three metrics, despite its poor performance on the CGC dataset.

Real-world dataset There are many crashes found for binaries in the real-world dataset. We use the open-source tool `afl-collect` [49] to de-duplicate these crashes and identify unique crashes. Overall, we have successfully found unique bugs in 5 real-world binaries as listed in Table 4. It is worth noting that for binary `objdump`, `size`, and `nm`, only our newly proposed coverage metrics find unique bugs.

4.6 Comparison of Time to Crash

CGC dataset Since most CGC binaries only contain one bug, we then measure the time to first crash (TFC) for different coverage metrics across the ten rounds of trials. The accumulated number within a 95% confidence of binaries crashed over time is shown in Figure 4. The x-axis presents time in seconds while the y-axis shows the accumulated number of binaries crashed. For example, we can see that `n4` almost manages to crash more binaries than other coverage metrics in the first hour (3600 seconds) and `ma` performs the worst among them. We also see that all of the proposed coverage metrics other than `ma` and `mw` can help find crashes in binaries more quickly than the original AFL (`bc`). Moreover, although `n4` does not find the most crashes, it is the best one during the early stage (30 to 90 minutes). After 90 minutes, `ct` surpasses it and becomes one of the best performers. For the time each coverage metric spends on crashing individual binaries, please refer to Figure 11 in Appendix.

LAVA-M dataset Figure 5 presents the number of unique bugs found over time by different coverage metrics on the four binaries. We can see that the newly proposed coverage metrics outperform `bc` on all four binaries. Although `ma` is slower than others, it finally finds the same number of unique bugs on binary `base64` and `unique`. On binary `who`, `ma` even finds quite more unique bugs. Moreover, `ct` and `n8` perform stably well across four binaries, and the latter one performs

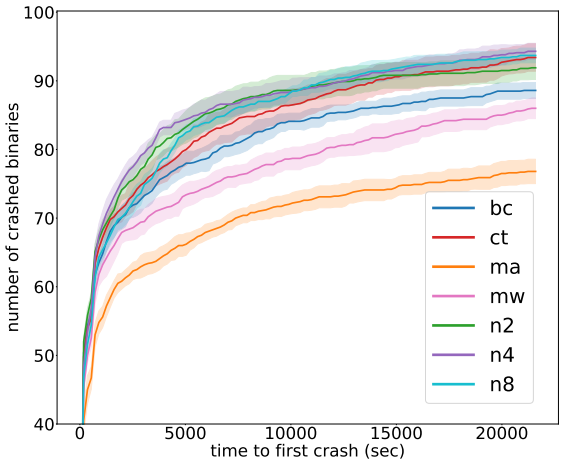

Figure 4: Number of binaries crashed over time during fuzzing on the CGC dataset. The x-axis presents in seconds and the y-axis shows the number of binaries whose TFC (time-to-first-crash) were within that time.

extremely well on binary `who`: it finds the largest number of unique bugs and much faster than the rest.

Real-world dataset Similarly, Figure 6 shows the number of unique bugs over time found by different coverage metrics on the five crashed binaries in the real-world dataset. We can see that except for `info2cap`, `bc` either finds unique bugs much more slowly than others or does not find any bugs at all. In addition, there is no global trend about which coverage metric is the fastest one to find bugs across the five binaries.

4.7 Comparison of Seed Count

CGC dataset We collect the number of seeds selected for each binary using different coverage metrics and report the mean number within a 95% confidence among the ten runs. Figure 7 displays the cumulative distribution of the numbers of generated seeds. A curve closer to the top left in the figure implies that in general fewer seeds are generated for binaries with the corresponding coverage metric.

We had several observations from the result. First, `ma` was significantly more sensitive than the rest coverage metrics. It selects several orders of magnitude more seeds than the others. While most of these seeds are stepping stones for more meaningful mutations that lead to final crashes, too many of them would hurt the fuzzing performance because the differences among most of the seeds are so tiny that they are unlikely to result in any new bug. Second, for n-gram branch coverage, as n increases from 1 (`bc`) to 8, the number of seeds increases correspondingly, although the lines for `bc` and `n2` are too close to each other. This phenomenon meets our expectation, as $n8 \succ n4 \succ n2 \succ bc$. Third, while in theory, we cannot compare `ct` with n-gram regarding their sensitivities, we observe that the seed count distribution for `ct` is between `n4` and `n8`, at least for the CGC dataset. Fourth, in theory, $ma \succ mw \succ bc$. We indeed observe these relations in the form of seed counts for `ma`, `mw`, and `bc`.

Table 5: The numbers of seeds generated by different coverage metrics on the LAVA-M dataset.

	bc	ct	ma	mw	n2	n4	n8
base64	208	170	16372	200	196	273	425
md5sum	706	497	75323	71131	474	719	4958
uniq	104	52	43928	50178	77	92	153
who	223	144	14183	16511	190	271	470

LAVA-M dataset Table 5 lists seed counts generated by each coverage metric on the four binaries in the LAVA-M dataset. We can see that the observations for the CGC dataset still hold in general, with some outliers. For instance, the seed counts of `ct` on all four binaries are smaller than those of `bc`. These numbers are not statistically significant, given such a small-scale dataset.

Table 6: The numbers of seeds generated by different coverage metrics on the real-world dataset.

	bc	ct	ma	mw	n2	n4	n8
file	38	38	38	19462	38	38	38
gif2png	1039	2037	151008	29606	804	1665	3840
gzip	1305	1340	124253	65035	1002	1875	5446
info2cap	4966	12555	76048	30136	4802	8831	17104
objdump	6015	42625	49401	126578	4978	8756	22914
readelf	8461	15317	91982	63009	8758	15425	35429
strings	61	62	1619	59	69	68	131
tiff2pdf	834	883	143902	2841	724	1108	2395
tiffset	2	2	2	2	2	2	2
size	2117	4860	111978	143693	1605	3003	10278
nm	12566	49307	133460	73386	5947	10174	23322
jhead	384	284	75328	29229	362	576	1376

Real-world dataset Table 6 lists seed counts generated by each coverage metric on the 12 real-world binaries. We can draw similar observations as on the CGC and LAVA-M datasets with some exceptions: the seed count distribution for `ct` is no longer between `n4` and `n8` in general.

4.8 Combination of Coverage Metrics

From the evaluation results above, we observe that each coverage metric has its unique characteristics in terms of crashes found and crashing times. This observation leads us to wonder whether combining fuzzers with different coverage metrics together would find more crashes and find them faster. To answer this question, we consider two options for combination: (1) fuzzers with different coverage metrics are running in parallel and synchronizing seeds across all metrics periodically (i.e., cross-seeding); and (2) fuzzers with different coverage metrics are running in parallel but independently, as the baseline to show whether cross-seeding really helps.

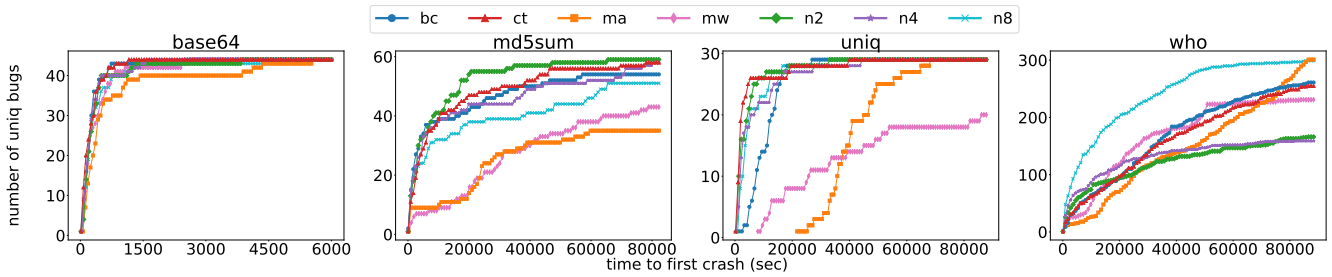

Figure 5: Number of unique bugs found over time during fuzzing on the LAVA-M dataset.

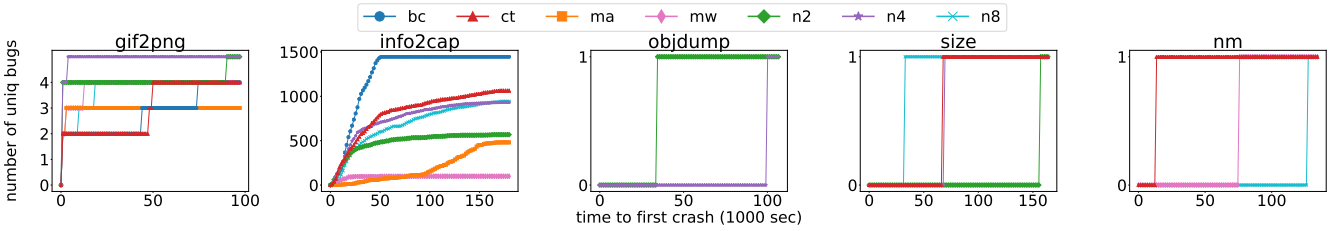

Figure 6: Number of unique crashes found over time on real-world dataset. The x-axis presents TFC in 1000 seconds.

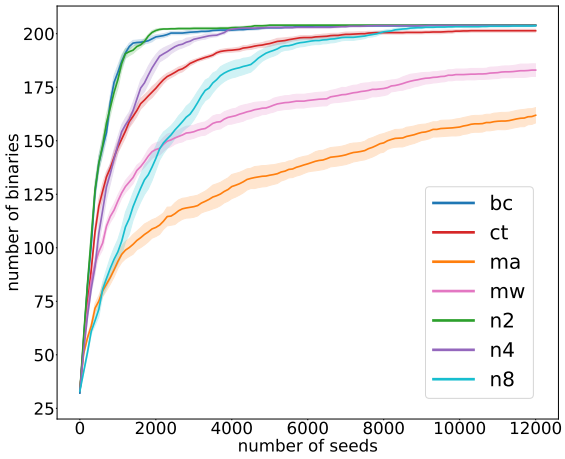

Figure 7: Partial CDFs of seeds generated by different coverage metrics on the CGC dataset. A curve closer to the top left indicates fewer generated seeds.

To study these two options, we create three configurations of 14 fuzzing instances: (a) all 14 fuzzing instances with `bc` and seed synchronization; (b) 2 fuzzing instances for each of the 7 different coverage metrics with seed synchronization only within the same metric; and (c) 2 fuzzers for each of the 7 different coverage metrics with seed synchronization across all metrics (i.e., cross-seeding).

CGC dataset We run the three configurations each for six hours, and for three times to get median results on the CGC dataset. Figure 8 illustrates the number of binaries crashed over time for the three configurations. We can make the following observations. First, both combination options outperform the baseline by large margins, with respect to both the number of crashed binaries and crash times. The combination without cross-seeding (configuration b) crashes 78 CQE binaries, 31 CFE binaries, and 109 binaries in total. The one with cross-seeding (configuration c) crashes 77 CQE binaries, 33 CFE binaries, and 110 in total. Meanwhile, the baseline only crashes 64 CQE binaries, 30 CFE binaries, and 94 in total. It is a notable achievement: the hybrid fuzzer Driller [35] was able to crash 77 CQE binaries after 24 hours with the help of concolic execution, where each binary is assigned to four fuzzing instances and all binaries share a pool of 64 CPU cores for concolic execution, using totally 12,640 CPU hours ($131\ \textit{binaries} \times 4\ \textit{cores} \times 24\ \textit{hours} + 60\ \textit{cores} \times 24\ \textit{hours}$). Compared with Driller, *we can achieve the same or even better results by pure fuzzing with less computing resources* ($131\ \textit{binaries} \times 14\ \textit{cores} \times 6\ \textit{hours} = 11{,}004\ \textit{CPU hours totally}$)*!*

Second, the blue line and the red line cross at around 3 hours. At this cross point, 105 binaries have been crashed for both configurations. It implies that the combination with cross-seeding is able to crash 105 binaries much earlier than the one without cross-seeding.

LAVA-M and real-world datasets We also run the three configurations each for 24 hours on LAVA-M dataset, and each for 48 hours on the real-world dataset. Figure 9 and Figure 10 present the results. We observed that the combination without cross-seeding always outperforms the baseline (14 fuzzers with `bc` only) by large margins. On the other hand, the combination with cross-seeding has inconsistent performance across these nine binaries. In some cases, it is even worse than the baseline. Unlike the result for the CGC dataset, this result is not statistically significant. However, it does indicate that sometimes, the overhead of cross-seeding may outweigh its benefits. Xu et al. [52] have shown that cross-seeding overhead is significant in parallel fuzzing and propose OS-level modifications for improving fuzzing performance. It would be interesting to re-evaluate the performance of the combi-

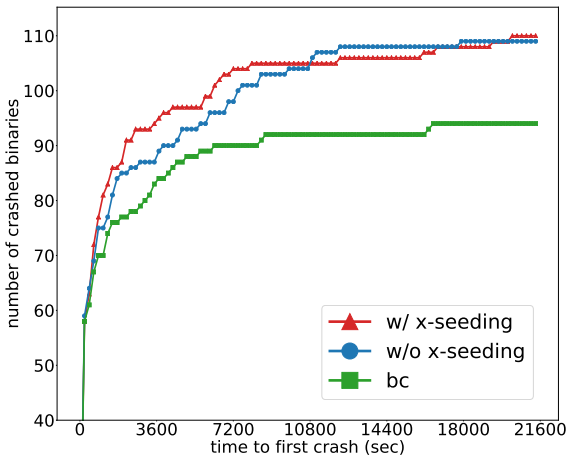

Figure 8: Number of binaries crashed during fuzzing tests by combining different coverage metrics on the CGC dataset.

nation with cross-seeding with these OS-level modifications. We leave it as future work.

In summary, *it is better to combine different coverage metrics with or without cross-seeding, which can help find more bugs and find them faster*.

5 Discussion and Future Work

In this section, we discuss several areas that can be potentially improved and explored in future work.

Precision and collision of coverage calculation For our presented coverage metrics, we adopt straightforward formulas for computing key-value pairs in the `hit_count` map, for the sake of efficiency, but at the cost of precision. For instance, Equation 3 uses a simple XOR for computing the calling context. As a result, it cannot differentiate a function being called twice from a function just returned. Similarly, Equation 2 XOR's previous $n-1$ block transitions together to compute n-gram branch coverage. This computation omits the exact order among these $n-1$ block transitions, and thus loses precision. A related problem is hash collisions [12]. Simple formulas presented in this paper may end up computing the same key from two sets of different input values. Better formulas that improve precision and reduce collisions deserve more investigation. Note that although [12] has proposed a greedy algorithm to reduce collision, the proposed method only works for branch coverage and cannot be easily applied to other coverage metrics.

Application-aware coverage metric selection and resource allocation In Figure 2, we can see the presented coverage metrics are not in a total order in terms of sensitivity. This means different coverage metrics have either unique strength in breaking through a specific pattern of code like loops. From the evaluation results presented in §4, we also observe that (1) there is no "grand slam" metric that beats all other metrics; and (2) even for metrics whose sensitivities are in total order (e.g., `bc`, `n2`, `n4`, `n8`), the most sensitive on is not always better. In this paper, we explored a simple combination of them and allocated computing resources equally among them. Because fuzzing can be modeled as a multi-armed bandit (MAB) problem [51] that aims to find more bugs with a limited time budget, previous work has shown how to improve the performance of fuzzing through adaptive mutation ratio [6]. Similarly, it might be possible to conduct static or dynamic analysis on each tested program to determine which coverage metric is more suitable. This decision may also change over time, so a resource allocation scheme might be useful to allocate computing resources among different coverage metrics dynamically.

6 Related Work

In §2 we have highlighted some related work on greybox fuzzing. In this section, we briefly discuss some additional work related to fuzzing.

Fuzzing was first introduced to test the reliability of UNIX utilities [22] in a blackbox way. Since then blackbox fuzzing has been widely used and developed that results in several mature tools such as Peach [46] and Zzuf [50]. There are many research works on improving it. For instance, Woo et al. [51] evaluate more than 20 seed scheduling algorithms using a mathematical model to find the one leading to the greatest number of found bugs within the given time budget. SYMFUZZ [6] optimizes the mutation ratio to maximize the number of found bugs given a pair of program and seed via detecting dependencies among the bit positions. Rebert et al. [30] propose an optimal algorithm of selecting a subset from a given set of input files as initial seed files to maximize the number of bugs found in a fuzzing campaign. MoonShine [23] develops a framework that automatically generates seed programs for fuzzing OS kernels via collecting and distilling system call traces.

Whitebox fuzzing aims to direct the fuzz testing via reasoning about various properties of the programs. Mayhem [5] involves multiple program analysis techniques, including concolic execution, to indicate the execution behavior for an input to find exploitable bugs. Taintscope [37] leverages dynamic taint analysis to identify checksum fields in input and locate checksum handling code in programs to direct fuzzing bypass checksum checks. BuzzFuzz [13] uses taint analysis to infer input fields affecting sensitive points in the code, which most often are parameters of system and library calls, and then make the fuzzing focus on these fields. MutaGen [18] aims to generate high-coverage test inputs via performing mutations on an input generator's machine code and using dynamic slicing to determine which instructions to mutate. Redqueen [1] presents another approach to solve magic bytes and checksum tests via inferring input-to-state correspondence based on lightweight branch tracing. ProFuzz [53] tries to infer the

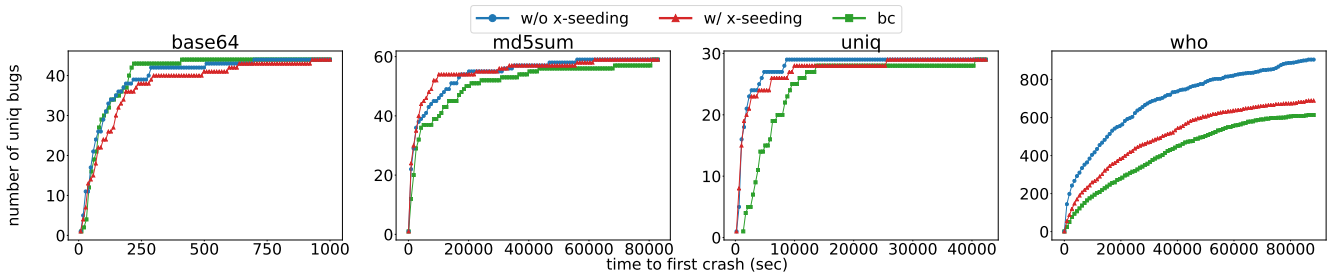

Figure 9: Number of unique bugs found over time by combining different coverage metrics on the LAVA-M dataset.

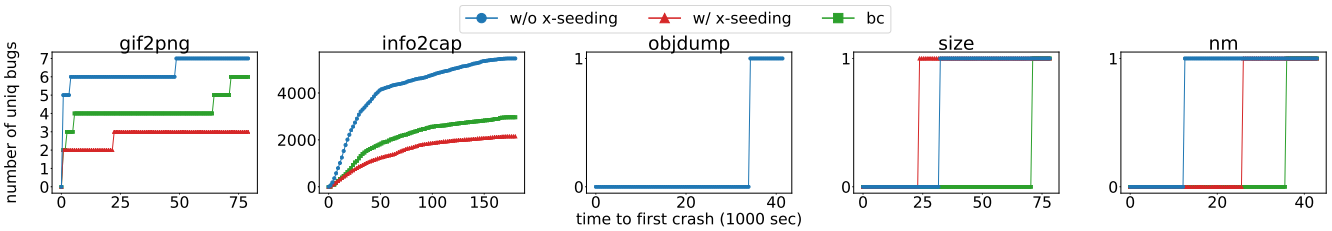

Figure 10: Number of unique bugs found over time by combining different coverage metrics for crashed real-world binaries.

semantic type of input bytes through the coverage information and apply different mutation strategies according to the type. Neuzz [32] approximates taint analysis by learning the input-to-branch-coverage mapping using a neural network, which can then predict what inputs bytes can lead to more coverage. Eclipser [9] identifies input-dependent branch predicates by checking which branches are affected when mutating an input byte; then uses binary search to flip the branch.

It is worth mentioning that recently whitebox fuzzing has been extensively explored in finding OS kernel and driver bugs. CAB-Fuzz [19] optimizes concolic execution for quickly exploring interesting paths to find bugs in COTS OS kernels. SemFuzz [54] uses semantic bug-related information retrieved from text reports to guide generating system call sequences that crash Linux kernels as Proof-of-Concept exploits. IMF [16] leverages dependence models between API function calls inferred from API logs to generate a program that can fuzz commodity OS kernels. DIFUZE [10] uses a specific interface recovered from statically analyzing kernel drive code to generate correctly-structured input for fuzzing kernel drivers.

The combination of whitebox fuzzing and blackbox/greybox fuzzing results in hybrid fuzzing. Pak's master thesis [24] first uses symbolic execution to discover frontier nodes representing unique paths and then launches blackbox fuzzing to explore deeper code along the paths from these nodes. Stephens et al. [35] develop Driller that launches selective symbolic execution to generate new seed inputs when the greybox fuzzing could not make any new progress due to complex constraints in program branches. Furthermore, Shoshitaishvili et al. [33] extend Driller to incorporate human knowledge. DigFuzz [56] proposes a novel Monte Carlo based probabilistic model to prioritize paths for concolic execution in hybrid fuzzing. QSYM [55] designs a fast concolic execution engine that integrates symbolic execution tightly with the native execution to support hybrid fuzzing.

In addition, Skyfire [36] proposes a novel data-driven approach to generate correct, diverse, and uncommon initial seeds for fuzzing to start with via leveraging knowledge including syntax features and semantic rules learned from a large scale of existing testcase samples. Xu et al. design new operating primitives to improve the performance of fuzzing with shortening the execution time for an input, especially when it runs on multiple cores in parallel [52]. T-Fuzz [26] develops transformational fuzzing that automatically detects and removes sanity checks making it get stuck in the target program to improve coverage and then reproduces true bugs in the original program via a symbolic execution-based approach.

7 Conclusion

In this paper, we conducted the first systematic study on the impact of coverage metrics on greybox fuzzing with the DARPA CGC dataset, the LAVA-M dataset, and real-world binaries. To this end, we formally define the concept of sensitivity when comparing two coverage metrics, and selectively discuss several metrics that have different sensitivities. Our study has revealed that each coverage metric leads to find different sets of vulnerabilities, indicating there is no grand slam that can beat others. We also showed a combination of different metrics helps find more crashes and find them faster. We hope our study would stimulate research on developing more coverage metrics for greybox fuzzing.

Acknowledgments

This work is supported, in part, by National Science Foundation under Grant No. 1664315, No, 1718997, Office of Naval Research under Award No. N00014-17-1-2893, and UCOP under Grant LFR-18-548175. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[4] DARPA CGC. 2014. DARPA Cyber Grand Challenge Binaries. https://github.com/CyberGrandChallenge. (2014).

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[6] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE)*. IEEE.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[11] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[13] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE.

[14] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2015. The Risks of Coverage-Directed Test Case Generation. *IEEE Transactions on Software Engineering* 41, 8 (2015), 803–819.

[15] Patrice Godefroid, Michael Y Levin, David A Molnar, and others. 2008. Automated whitebox fuzz testing. In *Proceedings of the 2008 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[16] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[17] Laura Inozemtseva and Reid Holmes. 2014. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM.

[18] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM.

[19] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS

Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX.

[20] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[21] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2008. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE.

[22] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[23] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium*. IEEE.

[24] Brian S Pak. 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *In Master's thesis, School of Computer Science Carnegie Mellon University* (2012).

[25] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*. USENIX.

[26] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[27] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[28] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[30] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 24th USENIX Security Symposium*. USENIX.

[31] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*. USENIX.

[32] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[33] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[34] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. *Information Systems Security* (2008), 1–25.

[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven Seed Generation for Fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[37] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[38] Website. 2018. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/. (2018). Accessed: 2018-04.

[39] Website. 2018. Angr: a framework for analyzing binaries. https://angr.io/. (2018). Accessed: 2018-04.

[40] Website. 2018. honggfuzz. http://honggfuzz.com/. (2018). Accessed: 2018-04.

[41] Website. 2018. Intel PIN Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. (2018). Accessed: 2018-04.

[42] Website. 2018. The LAVA Synthetic Bug Corpora. https://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html/. (2018). Accessed: 2018-04.

[43] Website. 2018. libFuzzer. https://llvm.org/docs/LibFuzzer.html. (2018). Accessed: 2018-04.

[44] Website. 2018. Of Bugs and Baselines. https://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html. (2018). Accessed: 2018-04.

[45] Website. 2018. OSS Fuzz. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. (2018). Accessed: 2018-04.

[46] Website. 2018. Peach Fuzzer. https://www.peach.tech/. (2018). Accessed: 2018-04.

[47] Website. 2018. Security @ Adobe. https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html. (2018). Accessed: 2018-04.

[48] Website. 2018. Trail of Bits Blog. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again. (2018). Accessed: 2018-04.

[49] Website. 2018. Utilities for automated crash sample processing/analysis. https://github.com/rc0r/afl-utils. (2018). Accessed: 2018-04.

[50] Website. 2018. Zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf. (2018). Accessed: 2018-04.

[51] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[52] Wen Xu, Sanidhya Kashyap, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[53] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland)*. IEEE.

[54] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*. USENIX.

[56] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

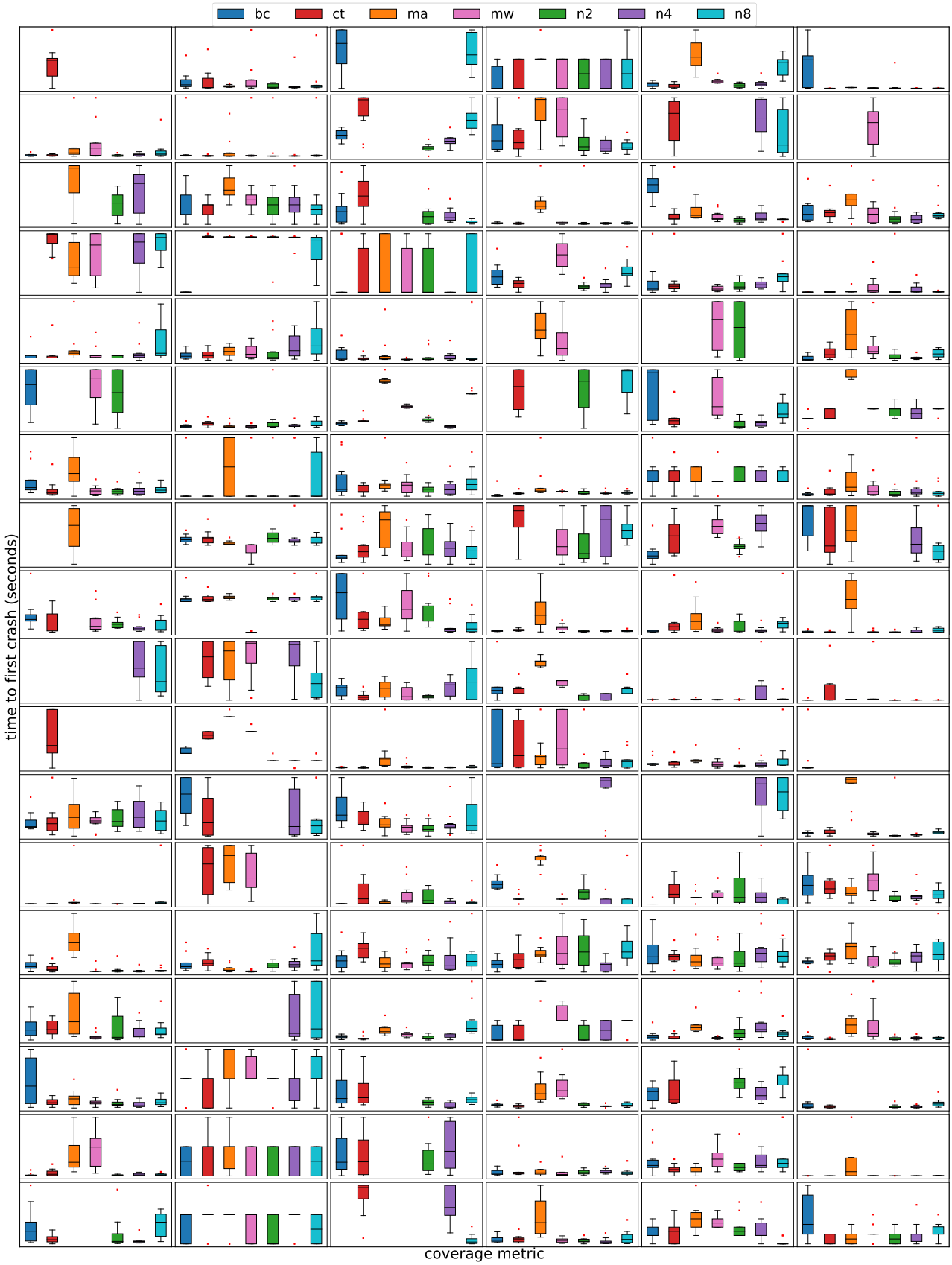

Figure 11: Time to first crash per binary of CGC dataset.