A Comparative Quality Metric for Untargeted Fuzzing with Logic State Coverage

Gwangmu Lee
iss300@gmail.com

Abstract

While fuzzing is widely accepted as an efficient program testing technique, it is still unclear how to measure the *comparative quality* of different fuzzers. The current de facto quality metrics are edge coverage and the number of discovered bugs, but they are frequently discredited by inconclusive, exaggerated, or even counter-intuitive results.

To establish a more reliable quality metric, we first note that fuzzing aims to reduce the number of *unknown abnormal* behaviors by observing more *interesting* (i.e., relating to unknown abnormal) behaviors. The more interesting behaviors a fuzzer has observed, the stronger guarantee it can provide about the absence of unknown abnormal behaviors. This suggests that the number of observed interesting behaviors must directly indicate the fuzzing quality.

In this work, we propose *logic state coverage* as a proxy metric to count observed interesting behaviors. A logic state is a set of satisfied branches during one execution, where its coverage is the count of *individual* observed logic states during a fuzzing campaign. A logic state distinguishes less repetitive (i.e., more interesting) behaviors in a finer granularity, making the amount of logic state coverage reliably proportional to the number of observed interesting behaviors. We implemented logic state coverage using a bloom filter and performed a preliminary evaluation with AFL++ and XMLLint.

1 Introduction

Since AFL [50] gained popularity, fuzzing has been widely regarded as one of the most effective software testing methods. The most basic form is *untargeted* fuzzing, which investigates a whole program by executing randomly mutated inputs indefinitely without any target code. It has shown promising effectiveness on simple software despite its simplicity, motivating researchers to further improve it by incorporating various techniques [7, 12, 14, 30, 35, 46, 52] and applying it to complex software, such as hypervisors [29, 32, 36, 37] and language interpreters [11, 20, 40, 44, 47].[1]

Meanwhile, there have been ever-existing debates on measuring the *comparative quality* of fuzzing as more research has been done [9, 25, 28, 43, 45]. While some researchers proposed alternative ways to measure quality [8,9,28], the current de facto standard quality metrics are *edge coverage* (i.e., how many control-flow edges were triggered) and *the number of discovered bugs* (i.e., how many bugs were discovered). Generally, either larger edge coverage or more discovered bugs is deemed as a positive indicator of the fuzzing quality.

However, both metrics have been consistently refuted as unreliable. Edge coverage has long been discredited as a *weak* proxy metric as it often leads to indecisive measurements; it is common to see the edge coverage comparisons offering little difference (e.g., equally saturated edge coverage [32, 36]) or influenced by initial inputs (e.g., different requirements for initial inputs [11]). Furthermore, some experiments yield counter-intuitive results where a code coverage increment does not translate to more discovered bugs so well [5].

The number of discovered bugs, on the other hand, is a much more direct metric because it counts the very objects (i.e., bugs) that fuzzing aims to reveal. However, it is highly sensitive to side factors such as bug deduplicatation method, initial inputs, or the implementation details of both the fuzzer and the software under test [26]. Even worse, the number of total bugs is usually only a handful, making unclear factors easily introduce a measurement bias that significantly exaggerates the result in certain software.

In pursuit of a more reliable quality metric for fuzzing, we first note that the goal of fuzzing, from the software testing perspective, is *minimizing unknown abnormal program behaviors* (i.e., bugs). Fuzzing approaches this goal by observing as many *interesting* program behaviors as possible that likely reveal abnormal behaviors; the more such behaviors it checks, the fewer unknown abnormal behaviors are expected to be left. This rationale suggests that *comparing the number of observed interesting program behaviors between fuzzers* must

[1] In this paper, we call *untargeted* fuzzing simply fuzzing if not specified.

indicate its *comparative quality*, as a larger number offers a greater guarantee of fewer unknown abnormal behaviors.

As it is challenging to directly count interesting program behaviors when fuzzing produces hundreds of them every second, we propose *logic state coverage* as a feasible proxy metric of interesting behaviors. A logic state is a set of all satisfied branches during one execution, encoding the exhibited program behavior without branch repetition information. As a consequence, less repetitive (i.e., more *interesting*) program behaviors are reflected as more *distinct* logic states. Logic state coverage counts all distinct observed logic states during a fuzzing campaign, which means that a larger coverage suggests more observed interesting behaviors.

2 Formalizing Fuzzing

In this section, we formalize the fuzzing procedure to derive a strong quality metric. We first start with defining a program behavior (Section 2.1) and, based on the definition, formalize the procedure of untargeted fuzzing (Section 2.2).

Notation and assumption. We denote a set and its element in bold and lowercase, respectively. For example, $\mathbb{I}$ and $\mathbb{O}$ denote a set of all inputs and outputs respectively, where i and o are the element of each set (i.e., an input and an output). If not specified, all arguments assume single-thread programs.

2.1 Program Behavior

A program p is a map that cast an input i to an observable output o and a *program behavior* h, which can be defined as a stream of executed instructions. Formally, if $\mathbb{H}$ is a *program behavior space*, a set of all possible program behaviors,

$$p(i) = (o,h) \iff p : \mathbb{I} \to \mathbb{O} \times \mathbb{H}. \tag{1}$$

Program conditions. Program conditions are the instructions that affect the control flow and transform the program behavior accordingly. They are further broken down to two kinds: branch and exceptional. Branch conditions (or *branches*) decide the regular control flow between basic blocks. Exceptional conditions (or *exceptions*), on the other hand, interrupt the control flow and abnormally terminate the program.[2]

Normality of program behaviors. Program behaviors can be divided into two classes depending on the satisfied program conditions: normal and abnormal. Normal program behaviors only satisfy regular branches and terminate at program exit points. Abnormal program behaviors, on the other hand, satisfy at least one exception and terminate at non-exit points. Sanitization [13, 17–19, 38, 41, 49] can be seen as bringing *unintended* behaviors (e.g., integer-overflow or use-after-free) to the *abnormal* territory by adding exceptions.

[2] C++-style exceptions, despite its name, can be regarded as a form of branch conditions as they transfer the control flow to another basic block.

Uniqueness of program behaviors. For a single-thread program, program behaviors are exclusively determined by the history of program conditions. This is because a single-thread program always executes the instructions sequentially until the next program condition, making every portion of the instruction stream *implied* by terminating program conditions.

Formally, let $\mathcal{T}_b$ and $\mathcal{T}_e$ be the history of branches and exceptions. Then, for a single-thread program, there exists a function $f(\mathcal{T}_b, \mathcal{T}_e) = h$ that uniquely maps a pair $(\mathcal{T}_b, \mathcal{T}_e)$ to a certain program behavior h.

2.2 Untargeted Fuzzing

Fuzzing, as a form of software testing, aims at minimizing the number of unknown abnormal program behaviors. *Untargeted* fuzzing addresses this goal by: (i) investigating as many *new* program behaviors as possible and (ii) keeping the mutation base input *interesting* (i.e., more likely revealing unknown abnormal behaviors after mutation).

New vs. unknown behaviors. We first make a clear distinction between *new* and *unknown* behaviors. *New* behaviors are, as the name suggests, simply the behaviors that have not been observed until now. On the other hand, *unknown* behaviors are a subset of the *new* behaviors that are *not equivalent* to any observed behaviors. In particular, an abnormal behavior is *unknown* if it is not equivalent to any observed abnormal behaviors (e.g., a unique root cause). Formally, if U and N are the sets of **u**nknown and **n**ew behaviors, $\mathrm{U} \subset \mathrm{N}$.

Formal justification. The approach of untargeted fuzzing can be formally described as follows. Upon executing a mutated input, let $\Pr(\mathrm{N})$ be the probability of observing a **n**ew behavior as a result of execution. Then, the probability of discovering **u**nknown **a**bnormal behavior $\Pr(\mathrm{UA}) := \Pr(\mathrm{U} \cap \mathrm{A})$ is

$$\begin{aligned} \Pr(\mathrm{UA}) &= \Pr(\mathrm{N} \cap \mathrm{UA}) && \text{(by } \mathrm{U} \subset \mathrm{N}\text{)} \\ &= \Pr(\mathrm{N}) \cdot \Pr(\mathrm{UA} \mid \mathrm{N}). && \text{(by chain rule)} \end{aligned}$$

Here, the second term $\Pr(\mathrm{UA} \mid \mathrm{N})$ is the expectation of revealing an unknown abnormal behavior given a new behavior, which can be interpreted as the *interestingness* of the mutation base. In other words, untargeted fuzzing attempts to (i) increase $\Pr(\mathrm{N})$ to investigate more program behaviors while (ii) keeping the interestingness $\Pr(\mathrm{UA} \mid \mathrm{N})$ high.

Interestingness in practice. As the interestingness $\Pr(\mathrm{UA} \mid \mathrm{N})$ is the expectation of an *unobserved* behavior, there is no direct way to measure interestingness. However, the common usage of corpus minimization [1, 3, 23, 23] and branch hit count buckets [3, 50] in untargeted fuzzing suggests that a program behavior is less likely to be *unknown abnormal* (i.e., less interesting) if its mutation base input is highly repetitive. The rationales are: (i) if a fewer repetition was normal, more repetition is also likely normal and (ii) even if it was abnormal, it is not unknown if it corresponds to pre-observed less-repetition counterparts.

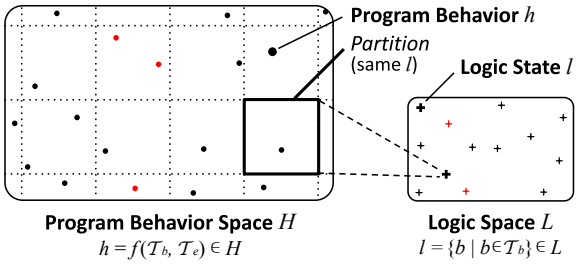

Figure 1: The program behavior space and the logic space. Red indicates abnormal program behaviors or logic states

.

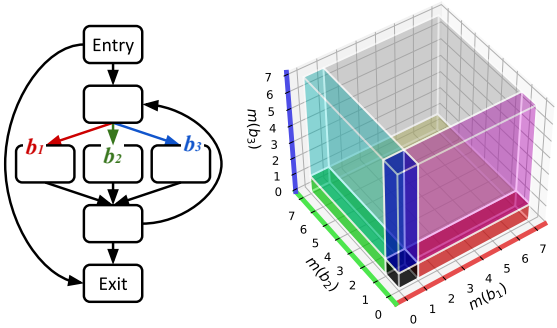

Figure 2: Example control flow graph and its logic states (colored box) that the program behaviors with different branch hit counts belong to. $m(b)$ denotes the hit count of branch b.

To generalize this practice from mutation base inputs to all inputs (i.e., all their program behaviors), a program behavior can be deemed less interesting if it include more repetitions.

Quality of untargeted fuzzing. As the procedure of untargeted fuzzing can be seen as maximizing observed interesting behaviors in the program behavior space, the portion of such behaviors in the space directly indicates the *quality* of the procedure: the more it checks such behaviors, the less the unknown abnormal behaviors are expected to be left.

Based on this, we can define the *comparative quality* between different untargeted fuzzing techniques as follows: given the same amount of time and computing resource, **how many more interesting program behaviors can it observe than others?** We argue that this is a *strong quality metric* for untargeted fuzzing as it directly compares the reduced expectation of unknown abnormal behaviors.

3 Logic State Coverage

In theory, interesting behaviors can be naively counted by investigating every program behavior. However, this naive approach is infeasible as fuzzing produces hundreds of arbitrarily long instruction streams *every second*. In this section, we propose *logic state coverage* as an indirean indirectt proxy of counting interesting behaviors.

Logic state and logic space. A *logic state* l is a set of all satisfied branches during execution, which is essentially the order-ignored version of a branch history. Formally, given a branch history $\mathcal{T}_b$ of an execution,

$$l := \{b \mid b \in \mathcal{T}_b\}. \tag{2}$$

A *logic space* $\mathbb{L}$, then, is a set of all possible logic states of a given program. It is possible to see a logic space as the *subdivision* of a program behavior space along logic states. Figure 1 shows the relationship between a program behavior space and a logic space, where each *partition* of a subdivided program behavior space corresponds to one logic state.

Logic state coverage. *Logic state coverage* is *the count of observed distinctive logic states during fuzzing*, where a logic state is called *covered* when fuzzing observed a program behavior that belongs to such a logic state (i.e., triggering all and only the branches in the logic state). Analogically, logic state coverage represents the *covered area* of a logic space.

Logic state coverage is crucially distinguished from edge coverage in that, while edge coverage merges all observed edge traces into one, logic state coverage distinguishes every logic state and counts them individually. This enables logic state coverage to count underlying program behaviors independent of what has been observed before, while edge coverage unpredictably ignores some of them as it is heavily affected by the previous observation.

As shown in Figure 1, a logic state (i.e., a partition in the program behavior space) may contain various program behaviors, suggesting a single covered logic state may encompass several different behaviors. However, in Section 4, we explain that a logic state is representative of interesting program behaviors as it (i) reflects the interestingness of underlying program behaviors and (ii) only contains the program behaviors of the same normality.

4 Representativeness of Logic States

In this section, we discuss the desirable properties of logic states to represent interesting program behaviors (Section 4.1). Then, we continue to discuss how logic states have such properties (Section 4.2 and 4.3).

4.1 Desirable Properties

Interestingness. Logic states should be sufficiently representative of the *interesting* program behaviors. Specifically, it should distinguish most program behaviors deemed interesting while not over-representing less interesting ones.

Normality. The *normality* of all program behaviors should be uniform (i.e., the same) within a logic state. Otherwise, the normality of one observed program behavior cannot represent that of other behaviors in the same logic state.

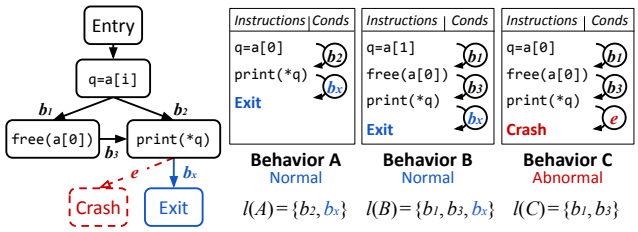

Figure 3: Example control flow graph and program behaviors. The exceptional condition e is satisfied when the memory pointed by q is a freed memory. $l(X)$ denotes the logic state of a program behavior X.

4.2 Interestingness

Logic states coalesce less interesting program behaviors that exhibit more branch repetitions in a fewer logic states, resulting in smaller (larger) logic state coverage as it observes more uninteresting (interesting) program behaviors.

Figure 2 shows an example control-flow graph and its logic space, gridded up by the different hit counts $m(b)$ of the branch b_1, b_2, and b_3. Each colored box represents one logic state, which includes all program behaviors that fall into the box. For example, a program behavior that does not hit any of b_1, b_2, and b_3 (i.e., $m(b_1) = m(b_2) = m(b_3) = 0$) belongs to the ■ logic state, while all program behaviors that hit only b_1 (i.e., $m(b_1) > 1$ and $m(b_2) = m(b_3) = 0$) fall into ■.

The colored boxes in Figure 2 suggest the increasing size of logic states as their constituent program behaviors involve more repetitive branches. To be specific, a logic state covers a progressively larger volume in the logic space (i.e., ■ → ■/■/■ → ■/■/■ → ■) as it contains more repetitive branches (i.e., $0 \to 1 \to 2 \to 3$ repetitive branches). This results in repetitive (i.e., uninteresting) program states packed to fewer logic states, suppressing (boosting) logic state coverage as it observes more uninteresting (interesting) behaviors.

4.3 Normality

Given that a program does not exhibit an exception exactly at the program exit point, all program behaviors in one logic state have the same normality, namely either normal or abnormal altogether. This is because normal behaviors trigger the *exit branches* that lead to the exit point, while abnormal behaviors do not. Since the logic states of two behaviors are always distinguished by exit branches, they cannot have the same set of satisfied branches (i.e., the same logic state).

Figure 3 shows an example program that contains a use-after-free exception e, which is satisfied when the memory pointed by q belongs to freed memories. Three example program behaviors and their logic states are shown on the side. Since normal behaviors (Behaviors A and B) terminate at the program exit, their logic states contain the exit branch b_x that directly leads to the program exit. On the other hand, since the abnormal behavior (Behavior C) triggers the exception e that crashes the execution before reaching the program exit, its logic state does not contain any exit branch unlike normal behaviors.

5 Measuring Logic State Coverage

In this section, we first summarize the high-level procedure of logic space coverage measurement and its subsequent requirements (Section 5.1). Then, we introduce a *bloom filter* as an effective way to achieve the requirements (Section 5.2).

5.1 High-level Procedure

At a high level, the measurement can be described as follows:

1) *Record* a logic state per execution.
2) *Add* the logic state to a set of observed logic states.
3) *Count* the size of the set, yielding logic state coverage.

In this procedure, recording a logic state (1) is technically identical to edge tracing in coverage-guided fuzzers [3,14,50]. Managing logic state coverage (2 and 3), on the other hand, poses an extra challenge as fuzzers are expected to observe a myriad of distinct logic states during a campaign.

Requirements. As the purpose of logic state coverage is comparing (i.e., evaluating) different fuzzers, the measurement should not induce any bias to certain fuzzers. To this end, it should satisfy the following requirements.

- **R1. Light computation overheads.** The computation overheads should not be heavy. Otherwise, it would overshadow the benefit of fast fuzzing executions, penalizing high-speed fuzzer designs [36,37,39].
- **R2. Constant computation overheads.** The measurement overheads should not increase as logic state coverage expands. Otherwise, it would penalize the fuzzers with larger logic space coverage.
- **R3. Feasible memory overheads.** The memory requirement for the measurement should be feasible for commodity evaluation settings for practical reasons.

5.2 Leveraging a Bloom Filter

For logic state coverage management, we employ a *bloom filter* that offers an efficient set implementation at the cost of manageable inaccuracy (i.e., false positives). Figure 4 illustrates the basic operations of a bloom filter. Starting from an array of 0s, it *adds* an element by setting the hash indices to 1 and *counts* the number of elements by tallying 1s in the filter and applying it to the counting formula (Equation 4).

Usage and benefit. In the measurement, a bloom filter represents *a set of observed logic states*, which accepts a logic

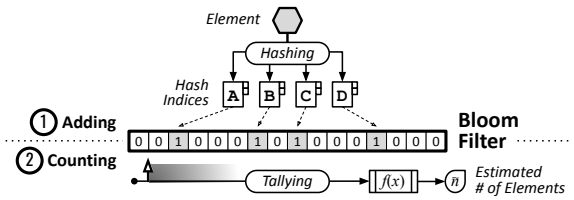

Figure 4: Operation illustration of a bloom filter.

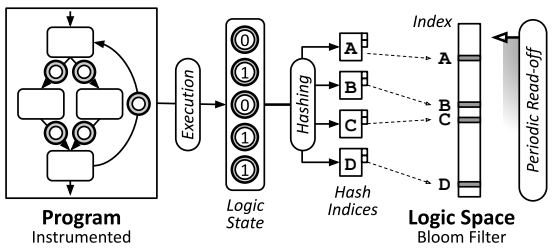

Figure 5: Logic state coverage measurement overview.

state every execution and yields logic state coverage via the number of elements in the set. It only requires a few computations for hashes and the counting formula (**R1**), both of which are nearly $O(1)$ as the number of hashes and the filter size are fixed (**R2**). Furthermore, the filter size is entirely agnostic to the size of a logic state thanks to hashing (**R3**).

Deciding parameters. A bloom filter may introduce some inaccuracy by construction. However, it can be reliably suppressed by the choice of data structure parameters, namely the filter size (i.e., the number of bits) n_b and the number of hashes n_h. They can be optimally determined by the maximum expected number of elements n_e and the desired false positive probability ε by the following equations [42]:

$$n_b = -n_e \ln \varepsilon / (\ln 2)^2, \quad n_h = -\log_2 \varepsilon, \tag{3}$$

In our setting, n_e corresponds to the upper bound of *observable distinct logic states* during a fuzzing campaign. To conservatively estimate n_e, we adopt three premises in ideal evaluation : (i) fuzzing continues for the maximum 24 hours, (ii) a fuzzer executes the maximum 1,000 inputs per second, and (iii) *every* execution yields a distinctive logic state. Then,

$$\begin{aligned} n_e &= 24\,\mathrm{h} \times 3600\,\mathrm{s/h} \times 1000\,\mathrm{exec/s} \times 1\,\mathrm{state/exec} \\ &= 86.4 \times 10^6\,\mathrm{state} \approx 84 \times 2^{20}\,\mathrm{state}. \end{aligned}$$

Assigning the upper bound n_e and a reasonable false positive probability $\varepsilon = 0.05$ to Equation 3 yields:

$$n_b = 538 \times 10^6\,\mathrm{bit} \approx 512\,\mathrm{Mbit} = 64\,\mathrm{MB}, \quad n_h = 4.32 \approx 4.$$

Notice that adding a logic state only requires calculating four hashes per execution (**R1** and **R2**) and the memory overhead (64 MB) is also marginal to a commodity setup (**R3**).

Counting elements. Along with data structure parameters, a counting formula that estimates the number of elements may also affect the accuracy of logic state coverage. Papapetrou et al. [34] proposed a high-accuracy counting formula that calculates the estimated number of elements $\tilde{n}_e(X)$ when the number of 1s in the filter is X as follows:

$$\tilde{n}_e(X) = \frac{\ln(1 - X/n_b)}{n_h \ln(1 - 1/n_b)}. \tag{4}$$

Equation 4 comes with three major benefits. First, it offers one of the best accuracy among all alternatives (c.f., only 4% error even in an extreme filter density of 90%) [22, 34]. Second, it allows for a near-constant-time calculation as it only requires the number of 1s apart from the formula calculation, as well as empirically shown in [22] (**R2**). Finally, it presents a calculable error bound that the actual number of elements likely falls within given a confidence probability [34].

6 Design and Implementation

In this section, we first present the design overview (Section 6.1) and elaborate on details (Section 6.2 and 6.4). The implementation in progress can be found at `https://github.com/gwangmu/prog-space-cov`.

6.1 Overview

Figure 5 illustrates the workflow of logic state coverage measurement. Similar to edge coverage, it first records the *logic state* of each execution via instrumentation (Section 6.2). Then, it adds the recorded logic state to a *bloom filter* that represents a set of observed logic states, by setting the hash indices (Section 6.3). Finally, the measurement periodically reads off the bloom filter to count the number of observed logic states (Section 6.4).

6.2 Recording a Logic State

Recording a logic state is equivalent to tracing executed edges in conventional edge coverage. Specifically, a taken edge is calculated by combining the hashes of two consecutively executed basic blocks, which is then added to the logic state of the current execution. Notice that a logic state is *per execution* and is not merged throughout multiple executions as in edge coverage. We implemented logic state recoding based on the LLVM pass for `afl-clang-fast` [50].

6.3 Updating Logic State Coverage

The recorded logic state is updated to the bloom filter at the end of every execution. In particular, a number of hashes are calculated from the entire logic state (i.e., `A`, `B`, `C`, and `D` in Figure 5) and are used as bloom filter indices to be set. We used MurMurHash3 [4] to hash a logic state.

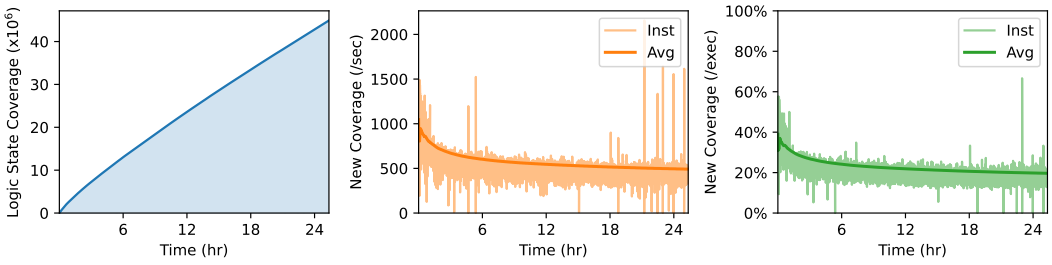

Figure 6: 24-hour fuzzing campaign of XMLLint on AFL++. (Ins: instantaneous, Avg: average)

6.4 Counting Logic State Coverage

Finally, the logic state coverage of the fuzzing campaign is measured by periodically reading off the bloom filter, where the number of 1s in the filter is fed to Equation 4 to count the distinctive number of observed logic states. We implemented logic state coverage measurement as a separate process, which a measurer should launch before starting a fuzzing campaign.

7 Evaluation

7.1 Preliminary Result

Figure 6 shows the preliminary evaluation result using XMLLint and AFL++. The evaluation was done on an Intel Core i7-8665U CPU machine with 16 GB of memory. The logic state coverage (left) shows the distinctive logic states covered over time. The new coverage per second (middle) shows the number of newly discovered logic states per time, while the new coverage per execution (right) shows the percentage of executions that discovered new logic states per execution. AVG indicates the average new coverage per second or per execution, and INS indicates the instantaneous new coverage during the last 10-second measurement frame.

The logic state coverage increases steadily over time but gradually slows down toward the 24-hour mark. This trend is also reflected in the new coverage plot, where the average new coverage slowly decreases to a steady point toward the end of the fuzzing campaign. Notice that the sporadic spikes in the instantaneous new coverage (INS) are more in the per-second plot than in the per-execution plot, suggesting that these sudden new coverage surges were primarily caused by quicker execution.

7.2 Research Questions

Some of the prime research questions that further evaluation should answer include the following.

- **RQ1** Can logic state coverage be implemented with all requirements in Section 5.1 satisfied?
- **RQ2** How does logic state coverage measure the comparative quality of popular untargeted fuzzers?
- **RQ3** Can logic state coverage distinguish the fuzzers indistinguishable from conventional edge coverage?

For **RQ1**, the evaluation may evaluate the computation and memory overhead added by the logic state coverage measurement. It should not perturb the execution speed of measured fuzzers to minimize the measurement bias. For **RQ2**, the evaluation may compare the logic state coverage of popular untargeted fuzzers [2, 3, 14, 50] with a set of standard fuzzing targets (e.g., FuzzBench [31]). For **RQ3**, the evaluation may compare the logic state coverage of specialized fuzzers (e.g., hypervisor fuzzers [29, 32, 36, 37] and grammar fuzzers [11, 20, 40, 44, 47]), whose quality has not been well-distinguished by a conventional edge coverage metric.

8 Discussion

Completeness of untargeted fuzzing. Unlike comparative quality that only requires to compare the number of observed distinct program behaviors, it is difficult to measure the absolute *completeness* of untargeted fuzzing that requires to measure how much a program has been completely verified. This is because the total number of program behaviors is unknown: if it can be known, then we would already know which abnormal behaviors are possible, precluding the necessity of fuzzing in the first place.

One possible compromise is performing preliminary static analysis to *estimate* the total number of program behaviors, including abnormal ones. However, it would pose a potential problem that the estimated 100% completeness does not necessarily mean the actual 100% completeness, defeating the very purpose of measuring *completeness*.

Generalization to multi-thread programs. Unlike single-thread programs, *multi-thread* programs may concurrently execute multiple basic blocks. In this case, a program behavior (i.e., a stream of executed instructions) also depends on the thread interleaving in addition to the branch and exception histories, which breaks the proxy properties of a logic space (i.e., interestingness and normality). An alternative proxy of

comparative quality may be proposed by either extending logic state coverage or devising yet another coverage metric.

Normality of a partial logic state. It is crucial to notice that an abnormal behavior *still follows the regular control flow up until any exception happens*. In Figure 3, the abnormal behavior (Behavior C) follows the same control flow to a normal behavior (Behavior B) until use-after-free, which makes the behavior eventually abnormal unlike its normal counterpart. What it implies is that the executions that do not contribute to the growth of logic state coverage are not entirely wasteful executions because they are still possible to reveal abnormal behaviors. However, they are unlikely to do so, let alone revealing *unknown* ones (Section 2.2).

Quality metric for general fuzzing. Logic state coverage evaluates the quality of *untargeted* fuzzing, where the program behaviors with less repetition are generally deemed more interesting. However, *targeted* fuzzing [10, 21, 33, 53] attempts to capitalize the recent findings [6, 51, 53] that some code such as recent patches are potentially *more interesting* (i.e., likelier to cause abnormal behaviors), which may render logic state coverage to under-estimate targeted fuzzing by overlooking potentially interesting behaviors.

To devise a quality metric that also encompasses targeted fuzing, future research may need to address two issues revolving around interestingness. First, it should accurately describe the interestingness in an impartial way (i.e., not favoring a certain fuzzer or setting). To do so, it should assess which factor affects interestingness and how much more than one another. Second, it should incorporate such a description into the quality metric. Should it follow the philosophy of logic state coverage, grouping uninteresting behaviors may be a possible approach to make them less significant.

Alternative ways to manage logic state coverage. There has been much research work regarding how to estimate the cardinality (i.e., the number of elements) in a set in an efficient way [22, 34], where using a bloom filter is just one of the approaches. Harmouch et al. [22] evaluated different cardinality approximation approaches and compared their accuracy with various benchmarks. Although using a bloom filter is the most recent approach compared in the paper, other alternative approaches can be incorporated to manage logic state coverage as it is not always the optimum solution.

Semantically abnormal behaviors. While the definition of program behaviors is separated from the output, it can still incorporates *semantically* abnormal behaviors (i.e., semantic bugs) by means of exceptions. To be specific, semantic error detection such as semantic sanitizers [24, 49] may capture semantic bugs on the fly and make the behavior abnormal by terminating it. The definition allows this because the output is just an observable subset of the data context during the execution, which exceptions can reference at runtime.

9 Related Work

Reliability of conventional metrics. Since the advent of popular fuzzing, researchers have consistently called into question a method to evaluate fuzzers. Klees et al. [26] evaluated then state-of-the-art fuzzers and raised an issue about using edge coverage and the number of discovered bugs as quality metrics. UNIFUZZ [28] also pointed out that such metrics are incomprehensible and lead to incomplete assessment. Böhme et al. [9] also noted that edge coverage does not strongly represent the comparative quality between fuzzers. While they all suggest the necessity of an alternative quality metric, it has yet to be proposed and widely accepted in practice.

Alternative metrics. Some researchers did propose alternative metrics for coverage-guided fuzzing [15, 16, 43, 45, 48]. However, they are commonly meant for *performance* (e.g., more bugs or edge coverage), not for *comparison*, which makes using such metrics for comparison likely yield an unjustified advantage toward certain fuzzers. SAND [27] proposed *execution patterns* (conceptually similar to logic states) to enable fuzzing without sanitization. However, similar to other alternative metrics, it was never meant for comparative evaluation nor correlated to the fuzzing quality.

Evaluation methods and platforms. Böhme et al. [8] proposed a technique to estimate the probability of discovering more bugs, but the probability cannot be directly compared between fuzzers as it highly depends on the capability of individual fuzzers. UNIFUZZ [28] and FUZZBENCH [31] presents a platform to comparatively evaluate fuzzers in multiple metrics. Logic state coverage can be readily incorporated into the platform as a quality metric.

10 Conclusion

While fuzzing is widely accepted as an efficient program testing technique, it still lacks the metric to comparatively measure the quality of various designs and implementations. To establish a reliable quality metric, we note that the more interesting behaviors a fuzzer has observed, the stronger guarantee it can provide about the absence of unknown abnormal behaviors. Based on this, we propose *logic state coverage* as a proxy metric to count observed interesting behaviors. A logic state is a set of satisfied branches during one execution, where its coverage is the count of *individual* observed logic states during a fuzzing campaign. We implemented logic state coverage using a bloom filter and performed a preliminary evaluation with AFL++ and XMLLint.

References

[1] Efficient fuzzing guide. https://chromium.googlesource.com/chromium/src/+/

main/testing/libfuzzer/efficient_fuzzing.md. Accessed Jun 6, 2024.

[2] Honggfuzz. https://honggfuzz.dev/. Accessed Sep 15, 2024.

[3] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed Feb 16, 2024.

[4] Murmurhash3. https://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html. Accessed Sep 15, 2024.

[5] Scaling security with ai: from detection to solution. https://security.googleblog.com/2024/01/scaling-security-with-ai-from-detection.html. Accessed Feb 16, 2024.

[6] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[8] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. Estimating residual risk in greybox fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[9] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[10] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[11] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[12] Peng Deng, Zhemin Yang, Lei Zhang, Guangliang Yang, Wenzheng Hong, Yuan Zhang, and Min Yang. Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[13] LLVM Developers. Undefined behavior sanitizer, 2017. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

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

[15] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[17] Google. The kernel address sanitizer, 2016. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[18] Google. The kernel thread sanitizer, 2016. https://github.com/google/ktsan.

[19] Google. The kernel undefined behavior sanitizer, 2016. https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html.

[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: fuzzing for javascript JIT compiler vulnerabilities. In *30th Annual Network and Distributed System Security Symposium (NDSS)*, 2023.

[21] Solmaz Salimi Byoungyoung Lee Gwangmu Lee, Duo Xu and Mathias Payer. SyzRisk: A change-pattern-based continuous kernel regression fuzzer. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '24)*, 2024.

[22] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment*, 2017.

[23] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[24] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs

in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138. ACM, 2018.

[27] Ziqiao Kong, Shaohua Li, Heqing Huang, and Zhendong Su. SAND: Decoupling sanitization from fuzzing for low overhead. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024. Preprint.

[28] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[29] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. Videzzo: Dependency-aware virtual device fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[30] Yinxi Liu and Wei Meng. Dsfuzz: Detecting deep state bugs with dependent state exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[31] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 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*, 2021.

[32] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[34] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 2010.

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[36] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[37] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-dimensional hypervisor fuzzing. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–318, 2012.

[39] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[40] Prashast Srivastava and Mathias Payer. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[41] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[42] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 2012.

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 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)*, 2019.

[44] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[45] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[46] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[47] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[48] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. PathAFL: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[49] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[50] M. Zalewski. American fuzzy lop, 2017. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[51] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guorern Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul Yu. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *Proceedings of the 2020 ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[52] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. FISHFUZZ: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[53] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.