A Practical, Principled Measure of Fuzzer Appeal: A Preliminary Study

Miroslav Gavrilov
gavrilov.miroslav@gmail.com

Kyle Dewey
California State University, Northridge
Northridge, CA, USA
kyle.dewey@csun.edu

Alex Groce
Northern Arizona University
Flagstaff, AZ, USA
Alex.Groce@nau.edu

Davina Zamanzadeh
University of California, Los Angeles
Los Angeles, CA, USA
davina@cs.ucla.edu

Ben Hardekopf
University of California, Santa Barbara
Santa Barbara, CA, USA
benh@cs.ucsb.edu

***Abstract*—Fuzzers are important bug-finding tools in both academia and industry. To ensure scientific progress, we need a metric for fuzzer comparison. Bug-based metrics are impractical because (1) the definition of “bug” is vague, and (2) mapping bug-revealing inputs to bugs requires extensive domain knowledge.**

In this paper, we propose an automated method for comparing fuzzers that alleviates these problems. We replace the question “What bugs can this fuzzer find?” with “What changes in program behavior over time can this fuzzer detect?”. Intuitively, fuzzers which find more behavioral changes are likely to find more bugs. However, unlike bugs, behavioral changes are well-defined and readily detectable. Our evaluation, executed on three targets with several fuzzers, shows that our method is consistent with bug-based metrics, but without associated difficulties. While further evaluation is needed to establish superiority, our results show that our method warrants further investigation.

I. Introduction

Fuzzing is an important automated testing technique that has enabled the discovery of tens of thousands of bugs across a variety of software projects [1], [2]. Fuzzing is an active area of research with a long history (e.g., [2]–[5]). To ensure scientific progress, we must compare new fuzzers to competing fuzzers to determine if new fuzzers advance the field.

In this paper, we address the problem of comparing fuzzers in an efficient, automatable, precise, and reproducible way. Fuzzers are meant to find bugs, so the first metric that comes to mind is comparing the sets of bugs that each fuzzer finds on an arbitrary system under test (SUT). However, in a study of 32 fuzzer evaluations in the literature, only 7 used bug-based comparisons [5]. Of these 7, all used fixed SUTs containing known bugs to find; none compared on arbitrary SUTs with unknown bugs. Our own independent study (Section V) corroborates the rarity of bug-based evaluations. In practice, we rarely evaluate fuzzers in the same way we use fuzzers.

We argue that realistic bug-based evaluations are rare because they are prohibitively labor-intensive to perform. Specifically, bug-based evaluations require us to (1) classify inputs into *triggers* (i.e., inputs that cause a detectable fault in the SUT), and (2) group triggers into equivalence classes.

Classifying inputs into triggers and non-triggers is known as the testing-oracle problem [6], which requires extensive SUT domain knowledge. While triggers causing the SUT to crash (e.g., with a segmentation fault) are easily detectable, detecting more subtle faults like normal termination with incorrect results is far more difficult. Detection of such *correctness* bugs in addition to more commonly studied *crash* bugs, without requiring full oracles, is a key goal of our approach.

Grouping triggers into equivalence classes by the bugs they manifest is similarly difficult. For example, 1,000 triggers may all expose the same bug, meaning only one bug is found; heavily skewed situations like these are seen in practice [7]. Conversely, 2 triggers may expose two separate bugs, meaning two bugs are found. Heuristics exist to group triggers (e.g., Chen et al. [7], mutants [8], coverage profiles [9], fuzzy stack hashes [10]), but Kees et al. [5] show these often fail, leading to inaccurate results. While targeted bug fixes [11] have shown promise in crash-focused fuzzing (e.g. typical AFL [9]), this is limited to a small set of predefined bug types; for example, while this has been done for buffer overflows, it is still an open problem for use-after-frees [12]. Targeted bug fixes are extremely expensive and unlikely to ever be applicable to complex SUTs like compilers [12]. Overall, automatic grouping even for simple crashes is currently limited, and may *never* be effective enough for fuzzer comparison. Manually grouping bugs is impractical, given that fuzzers often find hundreds of thousands of triggers [2], [7], [13], [14].

Our goal with this work is to provide a fuzzer evaluation methodology that, like bugs, is rooted in SUT behavior on inputs from fuzzers. However, we want a methodology that is automatable and reproducible. Towards solving this problem, we make several key (though seemingly disjoint) observations:

- While bugs are subjective and non-reproducible, they nonetheless capture an interesting trait of the SUT’s behavior on an input. Despite their imperfections, bugs are on the right track for fuzzer comparison.
- Bugs are viewable as developer-made code changes which cause a SUT to exhibit unintended behavior.

- Software has versions, where each version is the result of code changes to a prior version. Developers make these code changes, which may fix or introduce bugs.
- Given a test input from a fuzzer, we can derive some SUT output for each version under the same input. By comparing these outputs, we can detect when a SUT version change leads to an output change.

Combining these observations, given test inputs from a fuzzer, we can automatically detect when a developer-initiated SUT change impacted the SUT's behavior. We use this detection as a novel basis of fuzzer evaluation known as *fuzzer appeal*.

We evaluate fuzzer appeal on three case studies (Section III), intentionally using examples from the compiler-fuzzing domain, where custom fuzzers are often required and simple bug-counting methods are ineffective. These case studies show that fuzzer appeal produces results consistent with those from bug-counting methods, and can even deliver meaningful results in the absence of bugs. We present this as a *preliminary* study, as many more case studies are needed to fully establish fuzzer appeal's validity. The lack of sufficient case studies is a reflection of the problems with bug-based evaluations: performing a single bug-based evaluation is a daunting task, requiring massive SUT developer cooperation and likely forcing us to use recent SUT versions which were not previously evaluated.

The contributions of this paper are as follows:

- The introduction of *fuzzer appeal*: our automated method for evaluating fuzzer effectiveness, based on finding developer-initiated changes in SUT behavior. (Section II)
- Preliminary experiments demonstrating the validity of fuzzer appeal in three case studies. (Section III)
- A qualitative comparison of fuzzer comparison methods, particularly those used in practice. (Section V)

II. Fuzzer Appeal

This section defines fuzzer appeal, our proposed metric for effective fuzzer comparisons. We begin with fuzzer appeal's intuition, followed by a formal definition.

A. Intuition Behind Fuzzer Appeal

Rather than trying to classify inputs into triggers and group triggers by the bugs they exhibit, we rely on two key notions:

- **History**. Rather than contacting developers to pinpoint interesting parts of the code, we rely on SUT version history as a proxy telling us which parts of the code the developers found interesting enough to modify. Ideally, this version history is composed of all internal development commits, not just major and minor SUT releases.
- **Behaviors**. Instead of basing our comparison metric on bugs, which are hard to define and require developer support to compute, we base comparisons on observed *behaviors*. A behavior is a test input (produced by a fuzzer), combined with the SUT's output on said input.

From a high level, fuzzer appeal is based on observing changes in SUT behavior on the same test inputs across the SUT's version history. Fuzzers which find more unique behavior changes are more effective at finding the consequences of developer-initiated code changes. Fuzzer appeal overapproximates bugs, as bugs form a subset of developer-initiated code changes. However, unlike bugs, fuzzer appeal requires no understanding of intended vs. actual SUT semantics.

When computing fuzzer appeal, we only care about changes in behavior, not the actual behaviors themselves. To this end, we construct a separate *behavior pattern* for each test input. A behavior pattern is a bitvector, where each bit's index corresponds to a pair of consecutive SUT versions (e.g., (v_0, v_1), (v_1, v_2), (v_2, v_3), and so on). The bit is a 0 if the two SUT outputs for the given input are identical, else a 1.

From these behavior patterns, we build a *behavior map*: a map where the keys are unique behavior patterns, and the values record how many times the given behavior pattern was observed. We consider all behavior patterns with this map, irrespective of the particular test input. This map tells us how many unique behavior patterns a fuzzer found (the keys), and how redundant a fuzzer's output (test inputs) were (the values).

Example. Say our SUT is a primitive calculator with seven versions. One fuzzer we want to compare creates three test inputs: "2+2", "3+3" and "3*3". We run these inputs on each of the seven versions, yielding the following outputs:

$$2+2 \mapsto 4,4,4,4,5,4,4$$
$$3+3 \mapsto 5,5,5,5,8,6,6$$
$$3*3 \mapsto 9,9,9,9,9,9,9$$

Without determining if the output is semantically correct, we can see that the first and second input expose differences when the output changes from 4 to 5 (first) and from 5 to 8 (second) between versions 4 and 5, and then again when 5 changes to 4 (first) and 8 changes to 6 (second) between versions 5 and 6. We construct behavior patterns from these behaviors:

$$2+2 \mapsto 4,4,4,4,5,4,4 \mapsto \mathbf{000110}$$
$$3+3 \mapsto 5,5,5,5,8,6,6 \mapsto \mathbf{000110}$$
$$3*3 \mapsto 9,9,9,9,9,9,9 \mapsto \mathbf{000000}$$

From these behavior patterns, we build a behavior map:

$$\{000110 \mapsto 2, 000000 \mapsto 1\}$$

This behavior map shows two distinct behavior patterns, where one comes from one input, and the other from two inputs.

Fuzzers with big behavior maps seem better for finding bugs, as such fuzzers better explore SUT semantics. However, the behavior map's size only tells part of the story. In addition to exposing many unique behavior patterns, an ideal fuzzer should expose a *diverse* set of behavior patterns. While behavior pattern counts and diversity may sound similar, they are distinct. Unlike pattern counts, measuring diversity requires more sophisticated techniques, and we use *Shannon's entropy* [15]–[17] for this purpose. In the context of fuzzer appeal, entropy is viewable as a measure of how many different ways there are to describe the SUT's history when looking through behavior patterns. The higher the entropy, the more diverse the behavior patterns observed.

B. Formal Definition of Fuzzer Appeal

DEFINITION: *Behavior Vector.* Let i be a test input, and I be a set of test inputs, where $I = \{ i_1, \ldots, i_n \}$. Given an input $i_k \in I$ and an ordered list of SUT versions $\overrightarrow{\text{SUT}} = [\,\text{SUT}_1, \ldots, \text{SUT}_s\,]$, a behavior vector $\overrightarrow{O_k}$ is defined as:

$$\overrightarrow{O_k} = \overrightarrow{SUT_j(i_k)} \text{ for } 1 \leq j \leq s \tag{1}$$

In other words, each input is mapped to the sequence of outputs for that input over the vector of SUT versions. The form of the inputs and outputs is irrelevant, as are how they are derived; they can be primitive datatypes, complex data structures, images, stack traces, or otherwise. The only requirement is to have some notion of equality among outputs.

DEFINITION: *Behavior Pattern.* Assume $\texttt{eq}$ is a function which takes two SUT outputs and returns 1 if they are equal, else 0. A behavior pattern $\overrightarrow{\Delta_k}$ for an input i_k is defined as:

$$\overrightarrow{\Delta_k} = \overrightarrow{\texttt{eq}(O_k(j), O_k(j+1))} \text{ for } 1 \leq j < s \tag{2}$$

In other words, each output in the behavior vector is tested for equality with the next output in sequence. Every element of $\overrightarrow{\Delta_k}$ can be encoded as a bitvector of length $s - 1$.

EXAMPLE. Formalizing the example given previously, where i_1 = "2 + 2", i_2 = "3 + 3", and i_3 = "3 ∗ 3", the observed behaviors $\overrightarrow{O_k}$ associated with these inputs are:

$$\begin{aligned}\overrightarrow{O_1} &= [\,\overbrace{4,4,4,4,5,4,4}^{|\overrightarrow{SUT}|}\,]\\ \overrightarrow{O_2} &= [\,5,5,5,5,8,6,6\,]\\ \overrightarrow{O_3} &= [\,9,9,9,9,9,9,9\,]\end{aligned}$$

We then compute the behavior patterns $\overrightarrow{\Delta_k}$ as:

$$\begin{aligned}\overrightarrow{\Delta_1} &= \ [\,0,0,0,1,1,0\,]\\ \overrightarrow{\Delta_2} &= \ [\,0,0,0,1,1,0\,]\\ \overrightarrow{\Delta_3} &= \ [\,\underbrace{0,0,0,0,0,0}_{|\overrightarrow{SUT}|-1}\,]\end{aligned}$$

DEFINITION: *Behavior Map.* A behavior map P_F for a fuzzer F partitions the input set by behavior pattern $\overrightarrow{\Delta}$ values and maps each partition to its size.

EXAMPLE. Continuing the prior example, we have two unique patterns, and thus two equivalence classes of patterns:

$$\{\overrightarrow{\Delta_1}, \overrightarrow{\Delta_2}\} \qquad \{\overrightarrow{\Delta_3}\}$$

This partitioning leads to the following behavior map:

$$P_F = \{000110 \mapsto 2, 000000 \mapsto 1\}$$

DEFINITION: *Information Entropy.* Given a random variable X that takes a value x with probability $p(X = x)$, the *information entropy* H of X is:

$$H(X) = -\sum_{x \in X} p(x)\; log_2\, p(x). \tag{3}$$

This comes directly from Shannon's entropy [15]–[17]. Entropy is lower when a data set can be explained in a simple way (i.e., the set can be significantly compressed without losing information), and higher when a data set cannot be explained in a simple way (i.e., the set cannot be compressed without losing information). For fuzzer appeal, we calculate the entropy of the behavior map P_F. Our discrete probability function $p(x)$ is the ratio of the number of times behavior pattern x was seen, over the total number of test inputs:

$$p(x) = \frac{P_F(x)}{\sum\limits_{y \in \text{dom}(P_F)} P_F(y)} \tag{4}$$

The more unique behavior patterns there are, and the closer the counts of each behavior pattern are to a uniform distribution, the higher the entropy.

DEFINITION: *Fuzzer Appeal.* For a fuzzer F with a computed behavior map P_F, we define *fuzzer appeal A_F* as:

$$A_F = |\text{dom}(P_F)| \cdot H(P_F) \tag{5}$$

where H is information entropy from Definitions 3 and 4.

Unlike bug counts, fuzzer appeal has minimum and maximum possible values, determined by plugging in the smallest and largest possible P_F sizes in Definition 5. This yields a minimum of 0 and a maximum of $2^{(s-1)} \cdot s$. In practice, s need only be large enough to allow different fuzzers to yield significantly different fuzzer appeal values, and Section III shows that s values as small as 7 can be effective.

III. EXPERIMENTS

For our evaluation, we chose three industry-grade targets with rich communities and publicly-available version histories. Bugs found in these applications have proven hard to track and solve, mostly because they are non-obvious correctness bugs. All three targets have complex input languages, so generic mutation-based fuzzers like AFL [9] need example corpuses or time to give productive results. Our goal with this section is **not** to find new bugs, but rather show how fuzzer appeal compares to other fuzzer evaluation metrics like bugs. We show that fuzzer appeal agrees with bug-based metrics, and can give meaningful results even in the absence of bugs. We phrase this entire work as a preliminary study because three case studies is insufficient to conclusively demonstrate fuzzer appeal's superiority; this is only enough data to show that fuzzer appeal is promising and worth further investigation.

A. Evaluation Setup

All experiments were run on a machine with 12-cores, 32 GB RAM, and Ubuntu 16.04. For each SUT we used two ways to limit fuzzing, corresponding to whether fuzzer effectiveness or efficiency is the main concern:

- Time-limited: stop fuzzers after k hours
- Size-limited: stop fuzzers after they generate k inputs

We use existing, well-known comparison metrics (Section V) and fuzzer appeal (Section II) to:

- **Compare similar fuzzers targeting the Z3 SMT solver [18]**. One fuzzer generates inputs targeting Z3's semantics, whereas the other merely generates valid Z3 inputs. This evaluation shows how *fuzzer appeal compares to the most precise bug-based metric with much less manual effort*; the actual fuzzers are irrelevant.
- **Compare two previously examined CSMITH operation modes** (default [2] and swarm testing [4]) showing that *fuzzer appeal can be used to show differences in test set variability with less work.*
- **Compare fuzzers targeting the Solidity smart contract compiler [19]**, showing that *fuzzer appeal can compare fuzzers even when trigger- and bug-based metrics cannot.*

B. Description of the SUTs and Fuzzers

1) **THE Z3 SMT SOLVER:** SMT solvers are vitally important tools used in many domains, including automated testing [20], synthesis [21], and verification [22]. Given the sensitive nature of these applications (e.g., proofs in verification), solver correctness bugs are a critical issue. We choose Z3 [18] as a target since it is actively developed, has an interested and responsive developer team, and its inputs (SMT-LIB2 [23]) and outputs (satisfiability results and models) are well-defined and understood. We choose 23 versions from Z3's git repository[1] previously found to contain 13 correctness bugs[2]. The two fuzzers used for evaluating Z3 were inspired by Dewey et al. [24], and generate Z3 inputs in different ways:

- The *syntactic fuzzer* generates arbitrary valid inputs
- The *semantic fuzzer* understands the meaning behind Z3's operations, and uses this knowledge to generate valid inputs which should evaluate to specific values

2) **THE GCC COMPILER:** The CSMITH C compiler fuzzer covers a large subset of the C language, and avoids the pitfalls of undefined or unspecified behavior. We compare two CSMITH [2] modes: **vanilla**, the default with all toggleable features on, and **swarm** [4], where a series of feature subsets (called *configuations*) are enabled and run for proportionally shorter amounts of time. Groce et al. [4] show that swarm testing reveals 42% more distinct ways to crash compilers with 30% fewer crash instances, and that it overall finds more bugs. Our evaluation explores whether this input diversity translates into greater fuzzer appeal. We use the same GCC versions as the swarm paper [4]. We chose a swarm of a 1,000 different configurations, generated from a binomial distribution.

3) **THE SOLC SOLIDITY COMPILER:** SOLC is a language for implementing smart contracts for the Ethereum Virtual Machine (EVM), and is a popular fuzzing target due to the current interest in blockchain. We fuzz SOLC versions $0.4.\{16 - 22\}$. Bhargavan et al. [25] formally verified the correctness of EVM bytecode, but neither SOLC itself nor its translation to EVM bytecode has been verified.

We fuzz SOLC with two fuzzers: a black-box grammar-based fuzzer written in TSTL [26], and AFL [9], a gray-box mutational fuzzer. AFL guides fuzzing via information gathered from instrumented SUT source code. We construct a seed corpus for AFL by supplying it example SOLC contracts from the SOLC distribution. We run AFL both with and without the corpus to evaluate these options against each other.

As a black-box fuzzer, the TSTL-based fuzzer (in swarm [4] mode) generates inputs without any runtime SUT knowledge. The TSTL-based fuzzer for SOLC was built after the Solidity development team observed that AFL stopped finding bugs after long periods of time. We expect the TSTL-based fuzzer to be more effective than AFL. A second version of the TSTL-based fuzzer was subsequently develeoped which adds more constructs to the grammar used to generate SOLC contracts; we expect the second version to perform better than the first.

C. Results and Discussion

Our results are shown in Tables I and II. Fuzzer appeal values are across 10 separate runs, with the mean and standard deviation given. We split our discussion by case study below.

1) **Z3:** Depending on how we limit the experiments, the two fuzzers perform very differently. The semantic fuzzer is significantly slower than the syntactic one. With this in mind, if the generation is time-limited, the semantic fuzzer generates only around 1,000 inputs in 24 hours. In contrast, the syntactic fuzzer creates an input set with 39,630 inputs in the same amount of time. Each fuzzer found a single distinct bug; there was no overlap between the fuzzers. While counting bugs and triggers is uninformative for comparison here, fuzzer appeal reveals that there are meaningful differences between the semantic and syntactic fuzzers, with the semantic fuzzer revealing more behaviors.

When the generation is size-limited (with a limit of 6,000 inputs), cutting the fuzzers off when they reach the limit is equivalent to running the semantic fuzzer for 10 days, and the syntactic for 8h. This scenario is interesting for users running fuzzers indefinitely, who prioritize fuzzers which yield better overall results. We see that even with a third of the input set removed, the syntactic fuzzer still generates a similar fuzzer appeal (27.98) as before (34.02). The semantic fuzzer has a much higher appeal (220.9 vs. 43.0), however. This increase corresponds to the behavior map having more patterns (8 when time-limited, but 66 when size-limited). This increase in fuzzer appeal reveals that the semantic fuzzer is able to explore a much larger space of behaviors given more time, whereas the syntactic fuzzer's performance plateaus. We know this despite the fact that only two bugs are in play.

2) **GCC:** Fuzzer appeal of vanilla CSMITH is ~0.15 for both time- and size-limited cases. From this, we conclude that vanilla produces only a narrow range of behavior patterns, confirmed via manual inspection. Only one bug was found.

The swarm result differs between the two runs, but combining the time- and size-limited runs, the fuzzer found 3

[1] commits `424f34d`, `09980a4`, `4e7c7f2`, `44105b7`, `25f75b6`, `52df222`, `2115ea8`, `d4b6653`, `0b15fc9`, `04266fc`, `57af3a4`, `98b3a5b`, `40c5152`, `9dfc2bc`, `ec5a4ba`, `ed1a579`, `7a317a4`, `5d0db6d`, `cb6d008`, `ade2dbe`, `bf3a5ef`, `9b91e6f`, `bd187e0` in https://github.com/Z3Prover/z3

[2] https://github.com/Z3Prover/z3/issues/{68, 190, 212, 215, 220, 222, 480, 551, 567, 615, 616, 642, 643}.

TABLE I: Evaluation results with 24h time limit.

Target	Versions	Runs	Fuzzer	Appeal
Z3	23	10	Syntactic	34.02 ± 1.95
			Semantic	43.00 ± 1.02
CSmith	10	10	Vanilla	0.13 ± 0.02
			Swarm	20.20 ± 4.76
SolC	7	10	AFL	1.88 ± 0.02
			TSTL	6.22 ± 4.16

TABLE II: Evaluation results with 6,000 input limit.

Target	Versions	Runs	Fuzzer	Appeal
Z3	23	10	Syntactic	27.98 ± 4.65
			Semantic	220.9 ± 12.5
CSmith	10	10	Vanilla	0.15 ± 0.01
			Swarm	14.56 ± 6.5
SolC	7	10	AFL	3.93 ± 0.02
			TSTL	6.28 ± 2.71

distinct bugs, two of which involved crashes. The fuzzer appeal is about two orders of magnitude higher than that of vanilla CSmith, 14.56 when size-limited and 20.20 when time-limited. The increased time-based advantage shows that swarm testing is faster (428,880 generated inputs in 24h), and that numerous new input patterns are generated in this time, rather than repeating old patterns. In contrast, vanilla CSmith fuzzer appeal remains largely unchanged with more inputs.

Our results are best understood in the context of the original swarm testing paper results [4]. There, results for any one compiler and version were sometimes inconclusive, and the difference in coverage, which required a large number (14) of 24 hour runs to measure, was less than 2% for both GCC and Clang. Single target bug-counting or coverage results suggested that swarming offered only a marginal improvement. Only the collection of results from counting bugs over a large set of compilers and versions allowed the authors to conclude that swarm testing was *much* more effective, finding an additional 31 bugs with a 42% improvement. Fuzzer appeal avoids the need to classify behaviors as buggy or non-buggy, and (using 7 fewer compiler versions) even more conclusively shows the improvement of swarm over vanilla, in terms not just of *input diversity* but of *input-output behavioral diversity*. Only the large number of crash bugs present in the compiler versions allowed the original swarm paper to reach this conclusion; in a setting with fewer crash bugs and more semantic bugs requiring developer confirmation to distinguish, or simply with fewer bugs, there would be no easy way to accurately measure swarming's impact.

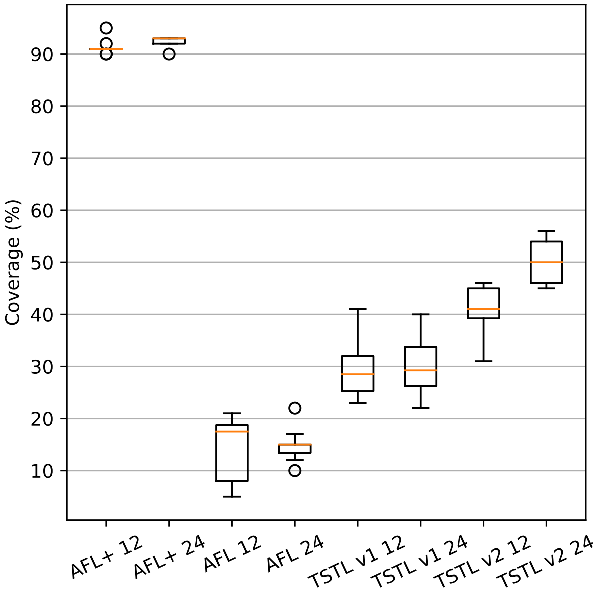

Fig. 1: Coverage for SolC testing

Unlike coverage, fuzzer appeal effectively captures input diversity. For example, a compiler expert examining vanilla CSmith tests would conclude that most tests are "very similar" programs; the narrow difference in coverage compared to swarming CSmith ($< 2\%$) shows that coverage does not reflect this similarity. The fact that CSmith detected many bugs means that there are subtle differences, but at a high level the inputs have low entropy and are simple to "explain". The same expert would note that swarm-produced C programs are often distinctive (e.g., "this program lacks pointers, and that one has many `goto` statements..."). This information cannot be represented by coverage percentage, or even mutation scores requiring many CPU-months to produce. Coverage, mutants, and small sets of bugs only scratch the surface of behavioral variety, hence the limitations of the correlations reported between these and fault detection [27]–[30].

3) **SolC:** Tables I and II show the AFL run that uses a seed corpus and the second (empirically better) version of the TSTL-based fuzzer. No bugs or triggers were found, representing the most difficult case for fuzzer comparison.

Figures 1 and 2 show, respectively, code coverage and fuzzer appeal for the 10 runs of each approach to SolC testing we tried, for both 12 and 24 hour time-limited runs. AFL+ means AFL using a seed corpus of example contracts included in the SolC distribution. Seeded AFL produces much higher code coverage than the TSTL fuzzers, **but much lower fuzzer appeal**. AFL tests primarily the parser, whereas the TSTL fuzzers focus on fuzzing core compiler functionality, generating degenerate, deeply-nested constructs that usually parse. While code coverage alone indicates the TSTL fuzzers are ineffective, fuzzer appeal shows that they are exploring complex compiler behaviors invisible to coverage. Sure enough, when all fuzzers were ran longer, the TSTL fuzzers found two new bugs in SolC[3] (now patched).

[3] https://github.com/ethereum/solidity/issues/{3738, 3759}

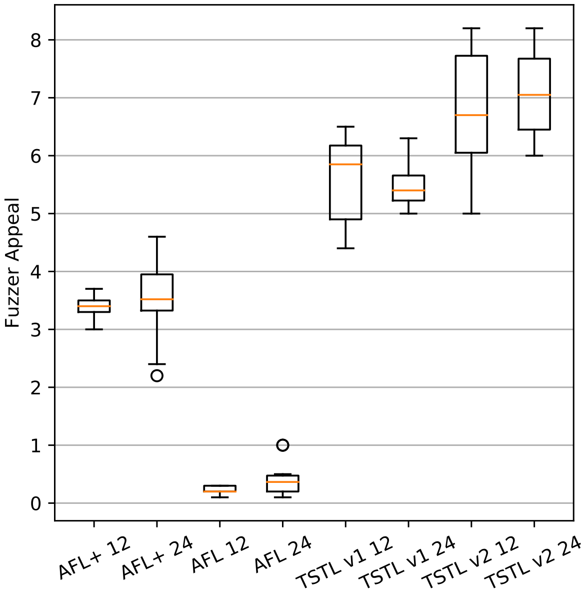

Fig. 2: Fuzzer appeal for SOLC testing

IV. Related Work

To the best of our knowledge, Klees et al. [5] is the only paper which extensively looks at the problem of fuzzer comparison. Klees et al. analyze the evaluations of 32 fuzzing papers, and find flaws in them all. Only 7 evaluations had meaningful bug-based comparisons, and these were all based on fuzzer test suites (Section V-B4) instead of arbitrary bugs. Klees et al. note a number of assumptions authors make with their evaluations, experimentally demonstrate these assumptions to be false, and show how this can lead to faulty conclusions. For example, they show that heuristic techniques for discarding duplicate triggers are highly inaccurate, leading unpredictably to bug overcounting and undercounting. Klees et al. underscores the importance of our work: it establishes that bug-based evaluations are preferred, but surprisingly rare.

V. Comparing Fuzzer Comparison Methods

In this section, we qualitiatively compare fuzzer appeal with other methods commonly used in the literature. Our comparison is based on three properties:

1) **Practicality**: can the method reasonably be used in practice? Methods that require prohibitive amounts of manual labor or developer interaction are impractical.
2) **Accuracy**: how accurate are the results from the metric? Inaccurate metrics may lead to incorrect conclusions.
3) **Generality**: can the metric be applied to arbitrary SUTs and fuzzers? E.g., requiring source code limits generality.

Table III summarizes our comparison results. Fuzzer appeal is the only method which simultaneously offers high practicality, accuracy, and generality. The rest of this section explains how we arrived at these results. We divide our discussion into *bug-agnostic* and *bug-aware* approaches.

Comparison Metric	Practicality	Accuracy	Generality
Code Coverage	High	Very Low	Low*
Mutation Testing	Medium	Low-Medium	Low*
Fuzzer Appeal	High	High	High
Sets of Bugs	Low	High	High
Bug Count	Low	High	High
Crash Bug Count	Med-High	Medium	High
Preselected Bug Count	High	Med-High	Low
Trigger Count	High	Low	High
Trigger Percentage	High	Very Low	High

TABLE III: Qualitative comparison of fuzzer comparison metrics. *Assumes coverage/mutants are based on source code.

A. Bug-Agnostic Methods

Bug-agnostic approaches are highly practical, as they do not identify bugs, do not involve developers, and operate without SUT specifications. Specific methods are discussed below.

1) Code Coverage: For fuzzer evaluation purposes, more code coverage is better. Tools measuring code coverage are widely available, it is simple and inexpensive to gather. However, code coverage has been found to be a poor metric even in simple settings [29], [31], and is only weakly correlated with bug-finding power [5], making it inaccurate. Additionally, code coverage is best performed on possibly unavailable SUT source code, harming its generality.

2) Mutation Testing: Mutation testing [32] is based on modifying a SUT's source code with individual modifications called *mutants*. Ideally, mutants produce different output than the SUT under specific inputs, and better fuzzers find more of these inputs. Mutation testing is generally considered more reliable than code coverage (e.g., [33]) hence we give it higher accuracy in Table III. However, like code coverage, mutation testing is best performed on SUT source code, harming generality. Optimizing mutation testing without significant trade-offs is difficult [34], reducing its practicality, due to its "blindness" regarding what is or is not important.

3) Fuzzer Appeal: Our method bears resemblance to mutation testing, where separate software versions are analogous to mutants. However, these versions are created by developers themselves, as opposed to being artificially created. As such, we know software versions are individually important: developers would not bother to make them otherwise. Whereas mutation testing is based on discovering artificially injected code changes, fuzzer appeal is based on discovering developer-created code changes, improving accuracy. Fuzzer appeal only needs SUT binaries, improving generality as well.

B. Bug-Aware Methods

In order to accurately identify bugs, bug-aware methods must solve two problems:

1) The **input classification problem**/testing-oracle problem [6], involving identifying which inputs reveal bugs.
2) The **trigger grouping problem**/duplicate trigger removal problem, involving mapping triggers to distinct bugs.

We differentiate bug-aware metrics by how they solve these two problems. We discuss specific bug-aware metrics below.

1) Sets of Bugs: This method is based on qualitatively comparing the sets of bugs exposed by multiple fuzzers, and can yield non-comparable results. For example, if fuzzer A finds bugs $\{b_1, b_2, b_3\}$, and fuzzer B finds bugs $\{b_4, b_5, b_6\}$, we cannot say which is better, as they find completely different bugs. Since sets of bugs is based on real bugs found, this technique is the most accurate. However, close developer communication is usually required to identify triggers and remove duplicates, making it impractical. If multiple SUTs exist which follow the same specification (e.g., C compilers), differential testing [3] can be used to automatically identify triggers, but trigger grouping is still a problem. Likely due to its laborious nature, we are aware of only one actual comparison based on sets of bugs, namely Holler et al. [1].

2) Bug Count: Bug count uses sets of bugs, but compares based on set cardinality. This trades accuracy for a simple numeric comparison. However, this technique is still labor-intensive, making it impractial. Like sets of bugs, bug count is rarely used in practice, likely for the same reasons. Yang et al. [2], Le et al. [35], and Dewey et al. [24] use bug count.

3) Crash Bug Count: This method is like bug count, but considers only crash bugs, which are trivially identifiable (e.g., non-zero exit status). Trigger grouping is usually made simpler by crash-specific output describing *how* the SUT crashed (e.g., stack traces), enabling a variety of heuristics (e.g., messages [7], coverage profiles [9], fuzzy stack hashes [10]). For these reasons, crash bug count is much more practical. The downside of focusing on crash bugs is that we tacitly assume that crash bugs are representative of *all* bugs. This is untrue, as shown by techniques which are more effective at finding non-crash bugs (e.g., Le et al. [35]). We observe that evaluations based on crash bugs are more common, arguably due to increased practicality. Both Yang et al. [2] and Groce et al. [4] use crash bugs extensively, and most papers evaluting AFL-like fuzzers consider only crash bugs (see Klees et al. [5]).

4) Preselected Bug Count: This is based on first selecting previously fixed SUT bugs. For each bug, two SUT versions are created: one with and without the bug. Both versions are then fuzzed, and any output difference indicates the bug is found, simplifying input classification. Classification is similarly simple: only one trigger per version pair is recorded.

As both trigger identification and classification are automatable, this method is very practical. Additionally, since version pairs are made from real bugs and all bugs are usable, this method is also accurate. However, this method is nonetheless problematic. Initial bug selection and version pair construction is a laborious process; highlighting this difficulty, such construction is the main contribution of Just et al. [36]. Each version pair must contain only a single bug to find, or else this will undercount actual bugs found. This assumes that one version in the version pair is correct, which is unlikely for most software, so undercounting is likely. Typically, authors use pre-created sets (e.g., Defects4J [36]), as opposed to creating their own. As such, we conclude that the *use* of preselected bugs is practical, but the *creation* of preselected bug suites is not.

Table III assumes that an existing suite is available, reflecting typical usage. This makes such suites highly practical. However, by limiting ourselves to preselected bugs, generality is correspondingly low: we can only test SUTs with existing suites, and even then we can only consider bugs in the suites. As most SUTs lack existing suites, this approach is usually inapplicable; there is no suite for C compilers (precluding Yang et al. [2], Groce et al. [4], Le et al. [35], Lidbury et al. [13]), JavaScript interpreters (precluding Holler et al. [1]), the Rust compiler (precluding Dewey et al. [24]), or SMT solvers (precluding Brummayer et al. [14]). Additionally, suite reuse risks overfitting fuzzers to small sets of known bugs [37]. Despite the problems, such suites are popular, and all 7 meaningful bug-based evaluations in Klees et al. [5] use them.

5) Trigger Count: This metric simply counts triggers, ignoring the grouping problem. Often, only crashes are considered, simplifying input classification. Counting triggers is very practical, at the cost of near-total accuracy loss; 0 triggers means no bugs were found, and non-zero means *at least* one bug was found. Yang et al. [2] explain that this metric can be easily gamed: randomly find a trigger, then generate the same trigger ad infinitum. Chen et al. [7] show that trigger counts often do not correlate with bug counts. This metric is used in Brummayer et al. [14], Yang et al. [2], and Lidbury et al. [13].

6) Trigger Percentage: This metric uses trigger count, but reports the percentage of inputs which were triggers instead of the count. This requires the same work as trigger count, making it similarly practical. However, its accuracy is even worse, as it can hide poor fuzzer generation rates. For example, if a fuzzer takes days to generate one input, but the input is a trigger, its trigger percentage is 100%. This metric is used in Yang et al. [2] and Lidbury et al. [13].

VI. Conclusion

Fuzzer appeal is a new methodology and metric for comparing fuzzers. Fuzzer appeal avoids the problems introduced when comparing fuzzers via bug-counting, which is necessarily resource-intensive, ad-hoc, and non-reproducible. Fuzzer appeal requires only that the SUT used in comparison of the fuzzers has an extensive version history; the actual computation of fuzzer appeal can be completely automated. There are no restrictions on the fuzzer types being compared (white-box, black-box, etc.) or on the types of SUT inputs and outputs, however they are derived. As long as there is a way to compare SUT outputs for equality, fuzzer appeal is applicable.

We evaluate fuzzer appeal via three case studies: fuzzing the Z3 SMT solver, the GNU C compiler, and the SolC smart contract compiler. Our evaluation shows that fuzzer appeal agrees with prior results but is much easier to compute. Additionally, fuzzer appeal can meaningfully compare fuzzers even without bugs, unlike bug- or trigger-based metrics. While further case studies are necessary to conclusively demonstrate fuzzer appeal's superiority, our evaluation shows that fuzzer appeal is promising and worth further investigation.

References

[1] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*,

ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[3] W. M. McKeeman, "Differential testing for software." *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[4] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 78–88.

[5] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2123–2138.

[6] W. E. Howden, "Theoretical and empirical studies of program testing," in *Proceedings of the 3rd International Conference on Software Engineering*, ser. ICSE '78. Piscataway, NJ, USA: IEEE Press, 1978, pp. 305–311.

[7] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 197–208.

[8] J. Holmes and A. Groce, "Causal distance-metric-based assistance for debugging after compiler fuzzing," in *International Symposium on Software Reliability Engineering*, 2014, pp. 166–177.

[9] M. Zalewski, "American fuzzy lop," 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[10] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 67–82.

[11] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 612–622.

[12] J. Holmes and A. Groce, "Using mutants to help developers distinguish and debug (compiler) faults," *Software Testing, Verification and Reliability*, p. e1727, 2020.

[13] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 65–76.

[14] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: ACM, 2009, pp. 1–5.

[15] R. Rajaram and B. Castellani, "An entropy based measure for comparing distributions of complexity," *Physica A: Statistical Mechanics and its Applications*, vol. 453, pp. 35–43, 2016, exported from https://app.dimensions.ai on 2018/08/20. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1027991479

[16] L. Masisi, V. Nelwamondo, and T. Marwala, "The use of entropy to measure structural diversity," in *2008 IEEE International Conference on Computational Cybernetics*, Nov 2008, pp. 41–45.

[17] R. M. Gray, *Entropy and Information Theory*. Berlin, Heidelberg: Springer-Verlag, 1990.

[18] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[19] "Solidity compiler," 2019. [Online]. Available: https://github.com/ethereum/solidity

[20] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335.

[21] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 281–294.

[22] K. R. M. Leino, "Dafny: an automatic program verifier for functional correctness," in *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 348–370.

[23] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," Department of Computer Science, The University of Iowa, Tech. Rep., 2015, available at www.SMT-LIB.org.

[24] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 482–493.

[25] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 91–96.

[26] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'brien, "Tstl: The template scripting testing language," *Int. J. Softw. Tools Technol. Transf.*, vol. 20, no. 1, pp. 57–78, Feb. 2018.

[27] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 547–558.

[28] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.

[29] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445.

[30] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 255–268.

[31] Y. Wei, B. Meyer, and M. Oriol, "Empirical software engineering and verification," B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. Is Branch Coverage a Good Measure of Testing Effectiveness?, pp. 194–212.

[32] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, 2011.

[33] K. Aaltonen, P. Ihantola, and O. Seppälä, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 153–160.

[34] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, September 2017.

[35] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226.

[36] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[37] M. B. Dwyer, S. Person, and S. G. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, 2006, pp. 92–104.