How to Compare Fuzzers

Philipp Görz
CISPA Helmholtz Center for Information Security

Björn Mathis
CISPA Helmholtz Center for Information Security

Keno Hassler
CISPA Helmholtz Center for Information Security

Emre Güler
Ruhr-Universität Bochum

Thorsten Holz
CISPA Helmholts Center for Information Security

Andreas Zeller
CISPA Helmholts Center for Information Security

Rahul Gopinath
University of Sydney

Abstract

Fuzzing is a key method to discover vulnerabilities in programs. Despite considerable progress in this area in the past years, measuring and comparing the effectiveness of fuzzers is still an open research question. In software testing, the gold standard for evaluating test quality is *mutation analysis*, assessing the ability of a test to detect synthetic bugs; if a set of tests fails to detect such mutations, it will also fail to detect real bugs. Mutation analysis subsumes various coverage measures and provides a large and diverse set of faults that can be arbitrarily hard to trigger and detect, thus preventing the problems of saturation and overfitting. Unfortunately, the cost of traditional mutation analysis is exorbitant for fuzzing, as mutations need independent evaluation.

In this paper, we apply *modern mutation analysis techniques* that pool multiple mutations; allowing us, for the first time, to *evaluate and compare fuzzers with mutation analysis.* We introduce an *evaluation bench* for fuzzers and apply it to a number of popular fuzzers and subjects. In a comprehensive evaluation, we show how it allows us to assess fuzzer performance and measure the impact of improved techniques. While we find that today's fuzzers can detect only a small percentage of mutations, this should be seen as a *challenge* for future research—notably in improving (1) detecting failures beyond generic crashes; and (2) triggering mutations (and thus faults).

1 Introduction

Fuzzing is the key method to test the robustness of programs against malformed inputs. Since it reveals inputs that crash or hang programs, and as these failures often can be turned into actual exploits, fuzzing is also the prime method to discover security vulnerabilities. However, fuzzing is computationally expensive. Hence, researchers and practitioners need the means to determine which fuzzing tools and techniques are the most effective. In a recent survey, 63% of fuzzing practitioners [1] named *measures for fuzzer comparison* as one of the top three challenges that need to be solved.

Testing techniques, including fuzzers, are often assessed by *code coverage* obtained [2]. This is reasonable, because to find a bug at some location, the test must cover this very location in the first place. But code coverage alone is not sufficient to actually find bugs. Coverage alone cannot evaluate the quality of sanitizers or fuzzers set up to produce inputs specifically crafted to induce bugs [3, 4]. Furthermore, there is only a moderate association between bugs and coverage when using test generators: Previous research shows a correlation coefficient $R^2 = 0.72$ between Randoop generated test cases and mutation score [5]. Another alternative to evaluate fuzzers is to run them on a benchmark of programs with *known faults,* and compare fuzzers by bugs found [2, 6, 7]. A general concern with such approaches is that the distribution of available faults may not be uniform or related to the actual possible fault distribution in the program [8]. Furthermore, when the faults are known in advance, competition in the field may force fuzzing researchers to *overfit* the fuzzer parameters or even the technique itself to find these faults [9, 10, 11, 1] and saturate them. Indeed, if the experience in other fields is anything to go by, the existence of benchmarks can be an open call to abuse [12].

In software testing, the technique of *mutation analysis* has established itself as the gold standard to evaluate tests and test generators [13]. In mutation analysis, one injects *synthetic faults* (mutations) into the program code by creating random variations (so-called *mutants*). The assumption is that a test set should be able to detect ("kill") these mutations, just as it should be able to detect real faults. As an example, Listing 1 shows a number of possible mutations for a C code fragment. We see that mutations such as changing the type of a variable (①) or a comparison (②, ③) may all impact the ability of a program to handle invalid inputs. A good test set should be able to trigger these faults; and the more mutants a test set detects ("kills"), the higher its quality.

In contrast to coverage metrics, mutation analysis also assesses the ability of the tests to *detect* the (injected) faults. Indeed, a test can have 100% coverage, but if it does not check any computation result, it will fail to detect errors. In a

```c
① unsigned int len = message_length(msg);
if (len ② < >= MAX_BUF_LEN ③ + 16) {
    copy_message(msg);
} else {
    // Invalid length, handle error
}
```

Listing 1: Mutations in C code. Mutation ① deletes `unsigned`; mutation ② replaces `<` with `>=`; mutation ③ adds `+ 16`.

fuzzing context, mutation analysis thus also assesses whether the fuzzer can detect issues beyond generic errors. And in contrast to curated faults, tests cannot overfit, as the actual mutations being applied are many, diverse, and randomly distributed. This lack of *bias* in mutations (i.e., anything can happen, anywhere) is often touted as a big advantage of mutation analysis. Indeed, while it may be tempting to model mutants after past fixes [14, 6], this biases test assessment towards past issues, which in turn puts less weight on the ability of tests to find yet *unknown* issues. (We are not aware how the Heartbleed [15], "goto fail;" [16], or log4shell [17] vulnerabilities could have been predicted from past issues.)

Several studies have confirmed the correlation between the ability to detect (mostly simple) mutations and the ability to detect (possibly complex) real faults [18, 19, 20]; and several works have explored how mutation analysis can be applied to security testing [21, 22, 23, 24, 25, 26]. Yet, mutation testing has a big disadvantage: it is *expensive*. Every single mutation induces a code change which needs to be evaluated by an entire run of all tests to assess whether they detect the mutation—and this for thousands of mutations. Multiplied with the dynamic tests produced by fuzzers, this makes mutation analysis *prohibitively expensive* for evaluating fuzzers. Furthermore, fuzzers can react to faults [4, 3, 27], limiting traditional avenues for optimizations in mutation analysis that assume static test suites. Recent advances in mutation analysis, however, can significantly reduce this complexity. Notably, the concept of a *supermutant* [28] enables us to evaluate multiple mutants together in a single test run. Also, we can proceed in multiple phases, first having the fuzzer quickly achieve maximum coverage producing a seed corpus for the next phase. Such a seed corpus is then used as a static test suite to kill a number of trivial mutants, leaving the few remaining ("stubborn") ones as potential fuzzing targets.

In this paper, we show that such optimizations actually allow applying *full mutation analysis to evaluate and compare fuzzers at scale,* using a myriad of *unbiased mutations* that assess the ability of a fuzzer to find any kind of bugs, including, but not limited to, software vulnerabilities. Our approach is implemented and available as a large-scale evaluation bench for fuzzers. We demonstrate the usefulness of the evaluation bench by applying them on a number of popular fuzzers (AFL, AFL++, libFuzzer, and Honggfuzz), evaluating them on a number of test subjects (`cares`, `vorbis`, `woff2_new`, `libevent`, `guetzli`, `re2`, and `curl`). We show that it is possible to conduct a full-scale evaluation and comparison of fuzzers using 16.36 CPU years of computation time, just under a month with our available hardware—a nontrivial, yet affordable amount of resources. To the best of our knowledge, the present work thus

- is the first study to apply mutation analysis to fuzzers, using traditional mutation operators as well as operators from a security context (a total of 31 operators);
- shows how optimizations in mutation analysis induce a speedup of nearly *two orders (95×) of magnitude,* making it applicable to fuzzers;
- demonstrates that fuzzers indeed differ in their ability to detect mutations, and consequently, faults; and
- shows that improvements in failure detection—notably the use of sanitizers—results in a better mutation detection.

These results demonstrate that our evaluation bench can be used to compare fuzzer performance and assess improvements.

In our study, we also found that only a small percentage of mutants is detected by at least one of the fuzzers. The best fuzzer in our evaluation, AFL++, covers 30.9% of all mutations and detects 28.3% of these covered mutants—that is, 8.8% of all mutations. These numbers may seem low on an *absolute* scale. However, note that mutation analysis produces *shallow mutants* that are easy to find, but also *deep mutants* that are very hard to trigger, and *subtle mutants* whose effects can be hard to detect. There even are mutants that, while changing the *syntax* of code or output, leave their *semantics* untouched, and therefore cannot be detected by any approach in the first place. But while a mutation detection rate of 100% is thus unlikely to ever being achieved, the detection rate is very useful as a *relative* measure to compare approaches and measure progress. In our study, for instance, using an enhanced oracle such as Address Sanitizer (ASAN) [29] results in a moderate improvement in detection to 34.2% of covered mutants (9.1% of total mutants).

We find that our results reveal important directions for research in fuzzing:

- First, *fuzzers can profit from better oracles*—that is, predicates that check for the presence of failures. Right now, fuzzers that use *generic oracles* such as crashes or hangs, are quite limited as not every vulnerability (and not every mutation) manifests itself this way. Our manual analysis shows that out of the mutants that were not detected, few could have been found by a crash oracle, see Section 5.2.1. However, a majority of these mutants would *still produce a behavioral divergence* from the original program, and hence can represent a vulnerability in a security context.

- Second, *fuzzers can profit from more targeted intelligence*—that is, generating inputs based on possible bug locations. Out of all mutants killed by AFL++ (28.3% of covered mutants), 94.4% were found with inputs generated on an unmutated binary. That is, at most 5.6% of the mutants were possibly killed because of targeted intelligence such as crafted inputs [3] or directing fuzzers towards potential problems [4, 27]. This is significant as according to an analysis of our syntactic fault patterns, we find that 71% of recent 100 CVEs were coupled to a mutation. Hence, fuzzers could significantly benefit from better targeting.

In summary, mutation analysis provides *ambitious* and *unbiased* goals for fuzzing and testing that suffer neither from saturation nor potential overfitting—and thus pose great challenges for future research. Our optimizations to mutation analysis, as introduced in this paper, give researchers and practitioners the means to check whether and which fuzzers meet these challenges. To this end, our evaluation bench will be made available as open source at https://github.com/xxxxxx.

2 Technical Background

We use the IEEE [30] nomenclature: A **fault** is a code artifact causing a *failure* (aka *bug*). A **failure** is an incorrect program behavior. An **error** is a human action that led to the fault. An **error model** defines the kinds of faults expected.

Given a limited compute budget, a fuzzing practitioner needs to choose a fuzzer that is most likely to find the most bugs, and is typically accomplished using *coverage criteria* or a set of curated bugs [8], which we discuss next.

2.1 Coverage Criteria for Fuzzer Comparison

Numerous coverage criteria exist [5] that can be used for judging the effectiveness of test suites and test generators such as fuzzers. Most feedback driven fuzzers such as AFL use some form of code coverage for guidance. Hence, code coverage achieved in target programs can be seen as a reasonable criteria for comparing fuzzers. The main problem with using code coverage, however, is that it is insufficient on its own for evaluating fuzzers. In particular, code coverage is unable to judge the quality of oracles such as sanitizers, which are important to ensure effective fuzzing. Another limitation is coverage saturation. That is, once a program element is saturated, there is little extra information available [31]. Many fuzzers [4, 32, 3], include intelligence to craft inputs (e.g., calling an API with invalid values) which, while important, is invisible if using coverage for fuzzer comparison.

2.2 Benchmarks Using Curated Faults

Fuzzers can be evaluated using curated fault benchmarks [2, 33, 7, 34]. However, such benchmarks are inherently limited to known faults. As specific benchmarks are used to measure effectiveness of a technique, the published improvements in the technique can become influenced by the benchmark in non-obvious ways. For example, if the faults were mined from existing faults, one may end up with numerous faults, or types of faults that are easy to detect. Further, if a given tool (such as AFL) was being used to find and eliminate faults during development, the faults that are detectable by such a tool may no longer be found in the faults mined from released versions— which does not mean that the effectiveness of the tool has been reduced. For example, exchanging AFL for another tool that does well on such mined faults may not produce the improvement a practitioner was hoping for.

That is, both techniques have inherent limitations. Next, we discuss how mutation analysis overcomes both limitations.

2.3 Mutation Analysis

Mutation analysis is a key technique for evaluating the fault-revealing power of test suites on a given program.

For mutation analysis, we start with the following *error model*: Any token in a program is a possible location for a fault to exist, and faults are likely caused during transcription of the concept in the developer's mind to the code artifact. Further, we assume that the developer uses automatic tools such as compilers, which removes some categories of faults. This gives us a way to generate possible faults with minimal human bias[1]: Generate all instances of a fault type for each source code element that will get past the compiler. However, many faults in the real world can be complex, containing multiple sub-faults. Modeling them with complex faults containing multiple sub-faults can lead to a combinatorial explosion, which is avoided by using two well-studied axioms —the *finite neighborhood hypothesis* and the *coupling effect*.

The *finite neighborhood hypothesis* (also called *competent-programmer hypothesis*) states that faults, if present in the program, are within a limited edit distance away from the correct formulation [35]. The *coupling effect* claims that simple faults are coupled to complex faults, such that tests capable of detecting failures due to simple faults will, with high probability, detect the failures due to complex faults. Hence, the probability of fault masking is very low [36]. Both axioms are well researched, with well-founded theory [37, 38, 39, 40], and confirmed in large number of real-world software [41, 40, 42, 43]. With

[1] There remains an unavoidable human bias due to the selection of fault types. However, generation for a fault type avoids bias as it is done exhaustively.

We define the following mutation related terms that we use throughout this paper:

Mutation. A small syntactic change that can be induced in the program.

Mutation operator. Transformation pattern that describes how mutations are induced in the program. A mutation operator, when applied to a matching location in the program, will produce a *mutant*.

Mutant. A new program that contains differences (mutations) from the original. A *first order* mutant contains only a single mutation. A *higher order* mutant contains multiple mutations. A *supermutant* contains all possible mutations that can be applied at once.

Trivial mutants. Mutants that can be killed without targeted intelligence. That is, any input whose execution covers their location will kill them.

Stubborn mutants. Mutants that remain alive even after coverage reached their mutation locations.

Intelligent mutants. Mutants killed by fuzzers on individual evaluation (i.e., they are not killed simply by covering their location).

Equivalent mutants. An equivalent mutant is a mutant that, while different from the original program syntactically, has the same semantics.

these two axioms, we can limit the faults we need to test. This allows us to focus on changes to the smallest program elements, such as tokens and statements, and still expect that the created mutations are representative of real bugs.

Given this error model, the idea of mutation analysis is to collect possible fault patterns (a single fault pattern is called a *mutation operator*), identify possible faults in the program (called *mutations*), generate corresponding faulty programs (called *mutants*) each containing a single *mutation*, and finally evaluate each *mutant* separately using each fuzzer and check whether the fuzzer is able to detect the changed behavior of the mutant (called *killing the mutant*).

In summary, we can use the huge body of work on mutation analysis as an effective method to compare fuzzers by using the number of mutants killed by each fuzzer as criteria.

2.3.1 Computational Requirements

Cost is a major concern with mutation analysis, as each mutant needs independent evaluation. Furthermore, for fuzzing, we need to evaluate each input produced independently on each mutant. We cannot tell if a mutant will be killed by an input without executing the mutant on an input. Indeed, we cannot even assume that the fuzzer will produce the same input on both the original and the mutant because the fuzzer may detect the mutation in the program, and take steps to induce failure on a perceived fault. That is, there is an effective quadratic increase in the number of program executions with program size.

There are several traditional optimizations to make mutation analysis less costly. However, these techniques assume static test suites, which makes them inapplicable to fuzzers. For example, the most effective optimization is to find the statements in the program that are covered by the specific tests in the test suite, then only run tests against mutants they cover (or that lead to a state infection [44]). This technique is inapplicable to fuzzers because fuzzers are non-deterministic, and the possibility of introspection on the source code for input generation resulting in different inputs for original and the mutant (violating the clean program assumption [43]). The same problem affects usage of *weak mutations* [45], split stream execution [46, 47], equivalence modulo states [48], and function memoization [49]. Thus, these traditional optimization techniques do not work well for fuzzers. We describe in Section 3 how we still achieve a significant reduction by using supermutants [28].

2.3.2 Residual Defects

One of the main reasons for using mutation analysis is that it provides the best estimate for *residual defects*—the number of defects that remain in a program after testing is completed and all found defects have been fixed [50, 51], i.e., the *residual defects* are those that have not been found (and fixed). Undetected faults that mirror a mutation (a single incorrect token) are accounted for with our mutation analysis based approach, as all mutations are applied to a program exhaustively. The larger and more complex types of faults are also subsumed by these mutations due to *coupling effect* hypothesis (see Section 2.3). That is, the number of mutants that remain undetected tracks the number of defects that remain undetected (residual defects) closely, and can be considered a *true ordinal measure* [50, 52] of the number of residual defects in a program.

We also note that the residual defect density can be estimated from the number of mutants found using statistical estimation tools such as *population estimation* [53] and *species richness* estimation as suggested by Böhme [54].

2.3.3 Design of Mutation Operators

Mutations are typically modeled on human errors such as exchanging a token in source code for another, or forgetting to add a statement. Some traditional mutation operators are also carefully chosen so that a test suite capable of detecting the resulting mutants also satisfies statement, branch [55], data-flow [56, 57], and various logic criteria [58]. That is, the test objective represented by a given criterion can be satisfied by the detection of a subset of mutants [13]. Beyond the traditional operators, operators are also chosen that reflects known fault patterns in specific domains [13].

2.3.4 Equivalent Mutants

One of the problems with traditional mutation analysis is equivalent mutants. These are mutants that are semantically the same as the original program. For example, in the following fragment,

```
if (cache.has(key)) return cache.get(key);
return compute(key);
```

removing the cache check need not induce a failure. Previous studies show that 10% to 23% of generated mutants could be equivalent [59, 60], which can limit the usefulness of mutation score (the absolute number of mutants killed).

We do not expect equivalent mutants to be a concern (beyond computational expenditure) for the following reasons: (1) In fuzzer *comparison*, only the *relative mutant kills* matter. (2) We apply manual analysis on a random sample of 100 stubborn mutants, and found only 11 equivalent mutants. This is also inline with the literature [59, 60]. This gives us statistical confidence ($89 \pm 6\%$ CI at 99% CL) that most of the mutants generated induced faults. See Section 5.2.1 for this analysis.

3 Approach

Based on this background information, we now describe how we adapt mutation analysis to fuzzers.

3.1 Selecting Mutation Operators

As fuzzers are mainly used to detect security issues, we create mutations that focus on generating vulnerable code to compare them. We started with the traditional mutation operators representing common programming errors as the baseline. Next, we went through the list of Common Weakness Enumerations (CWEs) for C [61] and C++ [62] creating mutations for interesting and feasible vulnerability types. Finally, we investigated recent CVEs for C and C++ projects adding unrepresented mutations modeling vulnerabilities that have been reported. The current list of supported mutation operators can be found in Table 10 in the Appendix.

3.2 Detecting Mutations

Another essential component is the detection of true mutant kills. To make this process robust against possible targeted manipulation, this needs to be done in a way that makes it difficult for fuzzers to cheat. For example, if we would classify a mutant as killed if the fuzzer reports a crash, it would be trivial for a fuzzer to just report a crash in every run. Similarly, we cannot base this decision on a version of the subject that is instrumented by a fuzzer, as during the instrumentation process, a fuzzer could itself introduce a crashing change. Thus, to confirm that any input that a fuzzer reports as crashing, the input is rerun on the non-instrumented versions of the original as well as the mutant (see Section 4 for a detailed description). A mutation is killed if the original exits with a different exit code than the mutant does. We require the original to exit without an error signal to avoid the case that there is a crashing bug in the original.

3.3 Reducing Computational Requirements

We analyze the fuzzer performance in two phases:

I. How much of the program is the fuzzer able to cover?

II. Does the fuzzer identify injected faults?

For Phase I, we are only interested in finding the maximum coverage obtainable, and for that, we use the fuzzer under test on the original program to generate a set of seed files that cover as much of the program as possible (called *coverage seeds* from now on). Since no known faults are present, the incentive for the fuzzer is purely to cover the maximum amount of source code. Next, we use the coverage seeds as a static test suite, where we can apply traditional mutation analysis optimizations and quickly remove any mutants that are killed by the coverage seed files. This allows us to eliminate trivial mutants (those that only need coverage to crash), which are a significant chunk of the total set of mutants, with limited computational overhead.

In Phase II, we use the coverage seeds as the starting point to fuzz the remaining stubborn mutants. This ensures that if the fuzzer contains "intelligence" to recognize and target the inserted fault, it can use that intelligence to find and kill the mutation. This method accounts for fuzzers that go beyond coverage and use advanced code analysis to guide fuzzing.

3.3.1 Supermutants

We use an approach based on supermutants [28] to evaluate mutants with less computational resources than traditional analysis. The basic idea is to identify independent mutations, and combine them together in supermutants, to allow a sound evaluation of multiple mutations without fault interactions for the compute cost of a single mutant.

We identify two mutations as independent if there is no seed input that covers both mutations during execution. Determined by whether mutations were covered in Phase I, we create supermutants as follows: Covered mutations are combined into supermutants if they are mutually independent. Non-covered mutations form supermutants by randomly choosing 100 mutations. In both cases, a function contains at most one mutation (due to a technical limitation of our mutation engine).

Note that a fuzzer can cover more than one mutation during a single execution in Phase II if the initial identification was wrong (we examine the likelihood in our evaluation). To minimize the impact of this effect, all fuzzer-reported crashing inputs are rerun to see if multiple mutations are covered, or any mutation was covered in the supermutants composed of non-covered mutations. In such cases, the supermutant is split up to remove the interaction, and reevaluated, this removes the possibility of multiple mutations interfering with each other.

3.3.2 Evaluating supermutants

We evaluate each supermutant using the seed files and fuzzers separately. If during fuzzing, we found that a supermutant cannot be killed, we mark all mutations in this supermutant as alive. If, on the other hand, a supermutant was killed, we identify the particular input or test case that killed it. Next, we use the function coverage to identify the particular mutation that was covered by the test case, which is then marked as killed. This gives us the precise set of ordinary mutants that could be killed.

4 Implementation

Our chosen subjects are C and C++ projects. We implemented an LLVM pass to find mutation locations (*mutation finder*) and another pass to do the actual code changes (*mutator*). See Figure 1 for an overview of the compilation process. The mutation finder identifies all mutation locations and possible mutations, giving each an ID. It also produces an executable (location executable) that can be used to check which mutations are covered for a given input. One comparison executable without any mutations is compiled using the base compiler. Given mutation ID(s), the mutator produces the corresponding (super)mutant bitcode file. The bitcode file is then compiled with a base compiler to create a comparison mutated executable, and each fuzzer compiles its own instrumented version.

To decide which mutations can be put together into one supermutant, we run all seed inputs on the executable produced by the mutation finder. To evaluate a mutation, we need to know if the mutation has been reached and whether it has been killed. An overview of this process is shown in Figure 2. Additionally, during fuzzing, we check if a mutation is covered using the mutated and instrumented mutated executables. To decide if a mutation has been killed, we rerun inputs that a fuzzer reports as crashing on the unmutated and mutated executables, and compare them.

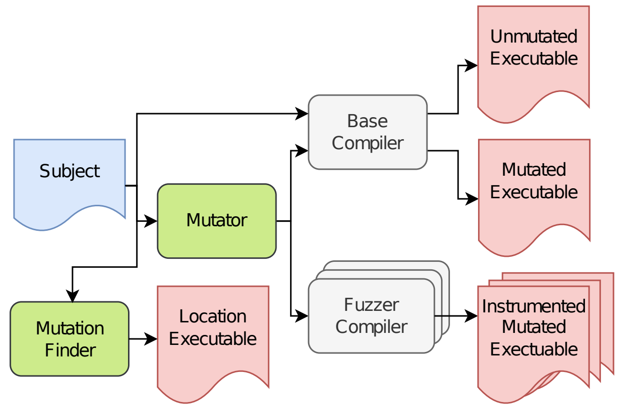

Figure 1: Overview of the different executables created from a single subject.

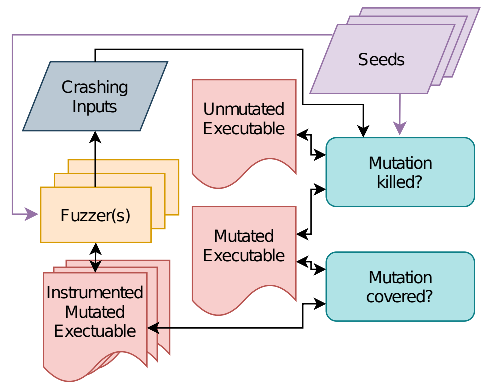

Figure 2: Evaluating a mutation.

5 Evaluation

Our evaluation seeks to answer the central question: *How well can fuzzers be compared using Mutation Analysis?* We evaluate four research questions to study mutation analysis as a comparative metric for fuzzer evaluation.

RQ1. *How do different fuzzers compare in killing mutants?* The question has three parts: (1) What percentage of mutants were killed in Phase I, (2) What percentage of the remaining were killed in Phase II, and (3) How do different fuzzers relate to each other in the mutants killed?

RQ2. *How much can sanitizers improve the results?* Inputs generated by a fuzzer accounts for only a part of the toolchain. Detecting bugs requires some kind of oracle, be it detecting crashes or more sophisticated ones such as sanitizers. With this question we want to assess if we can measure sanitizer influence.

RQ3. *How many real vulnerabilities are coupled to mutations?* For mutation analysis to be useful, the mutations it produces should be semantically coupled to some real

Table 1: Overview of the subjects used for evaluation.

Subject	Description	Version
`cares_name`	DNS query creation	809d5e84
`cares_parse_reply`	DNS reply parsing	809d5e84
`woff2_new`	Font Format	4721483a
`libevent`	Event Notification	5df3037d
`guetzli`	JPEG Encoder	214f2bb4
`re2`	Regular Expressions	58141dc9
`curl`	Data Transfer	curl-7_83_1

Table 2: Overview of the fuzzers used for evaluation. The arguments under Dict are additionally provided if there is a dictionary available.

Fuzzer	Arguments	Dict	Version
AFL	-d	-x	2.57b
AFL++	-d -c cmplog	-x	3.14c
Honggfuzz	-n 1	–dict	2.4
libFuzzer	-fork=1	-dict	11.1

faults. That is, for any real fault, there should exist a mutation such that detecting the mutation guarantees detecting the real fault [41]. Hence, this question seeks to understand whether we have succeeded in capturing the characteristics of real-world vulnerabilities with our mutation operators.

RQ4. *How much can we reduce the cost of mutation analysis on fuzzing?* This question measures the extent to which optimizations have reduced the cost of mutation analysis.

5.1 Setup

We describe our experimental setup next.

Subjects. For our experiments, we looked for robust subjects with no known faults. We selected all OSS-Fuzz [63] subjects (Table 1) that were compilable with gllvm and do not crash when fuzzed for 48 hours.

Fuzzers. Focusing on fuzzers that are commonly used and easy to work with. We chose AFL, AFL++, libFuzzer, and Honggfuzz for evaluation (Table 2), we chose generally recommended flags and use forking mode for libFuzzer to avoid early stopping. If dictionaries are available for subjects, they are provided to the fuzzers.

Hardware. We used four servers providing an Intel Xeon Gold 6230R CPU, each with 52 cores and 188 GB RAM. The compute times for all experiments is shown in Table 3.

Table 3: Actual compute times for all experiments. Experiments were run on four servers, each with 52 cores.

	CPU (Years)	4 Servers (Days)
Seed Collection	1.99	3.50
Default	14.37	25.22
Seed + Default	16.36	28.72
ASAN	15.16	26.61
24 Hours Runs	7.42	13.02
Sum	38.95	68.34

5.2 RQ1: How do different fuzzers compare?

Seed inputs are first created for each fuzzer and subject pair. These are either extracted from subject repositories if available or created manually. Each fuzzer is run on the unmutated base executable of the subject to fuzz for coverage for 48 hours (13 instances per fuzzer). The resulting inputs are then minimized by any fuzzer provided tool. Of the 13 instances for each fuzzer the median run based on covered mutations is selected as the coverage seed corpus. The median run is used to avoid outliers in performance, and is recommended in the literature [2].

We next create supermutants (Section 3.3.1), and evaluate each (Section 3.3.2) on the coverage seed for Phase I. In Phase II, each fuzzer is run for an hour on each supermutant.

Results. The results of this experiment are provided in Table 4. The column *#Mutations* represents the number of mutations produced. *Fuzzer* represents he fuzzer used. *Phase I Covered* represents the number of mutations covered by the seeds. *Phase I Killed* represents the number of mutations killed by the seeds. *Phase II Covered* represents the number of mutations that were covered in the Phase II beyond Phase I. *Phase II Killed* represents the number of mutations that were killed in the Phase II beyond Phase I. *Total Covered* represents the number of mutants that were covered in total, and *Total Killed* represents the number of mutants that were killed in total. The killed mutants are visualized in Figure 3. Our results show that a significant percentage of killed mutants are killed by all fuzzers (78%). We see that AFL++ kills 8.8% of all mutants, closely followed by Honggfuzz with 8.7% and AFL with 8.4%. libFuzzer lags behind at only 7.5%. In terms of covered mutations, AFL++ (30.9%) is again narrowly in front of Honggfuzz (30.4%), followed after a large gap by libFuzzer (25.9%) and AFL (24.6%). We find that coverage alone accounts for most of the mutants killed. The ensemble of all fuzzers in our evaluation (*combined* rows in Table 4) gets 99.95% of its coverage and 97.5% of its kills in Phase I. Only 1.0% of mutants are killed in Phase II. Indeed, none of the evaluated fuzzers employ bug-targeted feedback instrumentation, hence, this is expected.

Table 4: Results of the full benchmark run. The *#Mutations* column contains the number of mutations available. *Phase I* represents 24-hour runs, and *Phase II* represents the one-hour runs, and *Total* both combined. The *Covered* represents the number of mutants covered, while *Killed* represents the number of mutants killed. The **combined** rows show the total number of mutants that were killed by *any* fuzzer.

Program	#Mutations	Fuzzer	Phase I Covered	Phase I Killed	Phase II Covered	Phase II Killed	Total Covered	Total Killed
		afl	88	17	0	3	88	20
		aflpp	88	18	0	2	88	20
cares_name	4822	honggfuzz	88	17	0	1	88	18
		libfuzzer	86	17	0	0	86	17
		combined	88	18			88	20
		afl	937	292	0	29	937	321
		aflpp	941	289	0	26	941	315
cares_parse_reply	4822	honggfuzz	940	290	1	23	941	313
		libfuzzer	932	292	0	5	932	297
		combined	941	305			941	324
		afl	9,935	2,328	89	68	10,024	2,396
		aflpp	11,713	2,593	69	82	11,782	2,675
curl	29118	honggfuzz	10,195	2,299	506	150	10,701	2,449
		libfuzzer	8,895	2,099	120	39	9,015	2,138
		combined	12,459	2,807			12,477	2,857
		afl	6,943	1,691	6,189	1,827	13,132	3,518
		aflpp	12,564	3,205	685	378	13,249	3,583
guetzli	22961	honggfuzz	13,586	3,698	0	72	13,586	3,770
		libfuzzer	9,923	2,816	32	26	9,955	2,842
		combined	13,610	3,810			13,610	3,912
		afl	425	75	0	6	425	81
		aflpp	427	78	0	3	427	81
libevent	17234	honggfuzz	421	77	1	3	422	80
		libfuzzer	417	77	0	2	417	79
		combined	427	80			427	81
		afl	4,825	4,461	0	1	4,825	4,462
		aflpp	11,563	4,457	65	134	11,628	4,591
re2	21407	honggfuzz	11,636	4,431	0	97	11,636	4,528
		libfuzzer	11,164	4,328	0	20	11,164	4,348
		combined	11,639	4,571			11,639	4,627
		afl	5,386	955	2	70	5,388	1,025
		aflpp	5,578	1,044	1	69	5,579	1,113
woff2_new	40914	honggfuzz	5,555	1,018	1	111	5,556	1,129
		libfuzzer	5,018	896	2	20	5,020	916
		combined	5,631	1,139			5,634	1,232

Coverage accounts for most mutants detected (97.5%) in our evaluation.

We found two anomalous results regarding AFL: For `guetzli`, the median run covers only around 7,000 mutations, this is caused by wildly inconsistent runs covering from 2,500 to 12,000 mutations, a disadvantage from which AFL recovers surprisingly well in Phase II. Additionally, for `re2`, AFL (the fuzzer) crashes for most mutations. Our preliminary analysis hints at a conflict between ASAN and AFL instrumentation.

How do these fuzzers relate to each other? See Figure 3 for a Venn diagram of killed mutants per fuzzer. The fuzzers have a large intersection of killed mutations (78.6%). Consider AFL++: it finds 94.9% of all killed mutants. Adding a second fuzzer provides only a marginal improvement: Honggfuzz (+3.2%), AFL (+ 2.3%) or libFuzzer (+1.3%).

Is an extra hour of fuzzing beyond 24 hours of fuzzing for coverage sufficient for targeted fuzzing? To examine this question, we sampled 104 (to keep a multiple of CPU count) stubborn mutants for each subject, and fuzzed them for 24 hours on every fuzzer. Note that as we do not use supermutants for this experiment, here we use one mutation for a mutant, this doubles as a test if supermutants introduces incon-

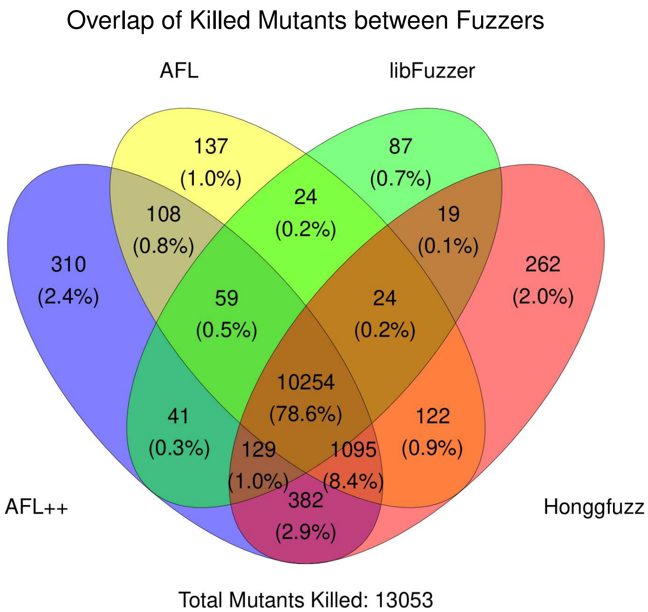

Figure 3: Venn diagram of killed mutants for each fuzzer.

sistencies. As only *three* of these 624 mutants are killed, we believe that one hour runs are indeed sufficient for Phase II. Additionally, we feel that this result confirms our choice of using supermutants. The detailed results can be found in Table 6.

5.2.1 Manual Analysis of Mutations

To assess whether mutations that we introduce result in a semantic change, we manually analyze 100 mutations that are not found during the 24 hour runs. Of these, we identify 11 mutations as equivalent, five that potentially lead to crashes, and the remaining 84 introduce a semantic change though they are unlikely to be found with a simple crash oracle.

5.3 RQ2: Evaluating the sanitizer contribution

To answer this question, we re-run the previous experiment with ASAN. To analyze the results of this experiment, we compare the percentage of killed mutants out of all covered mutations. This is visualized for AFL++, Honggfuzz and libFuzzer in Figure 4. The full results can be seen in Table 5. We see a clear increase of the number of killed mutants when using ASAN. Interestingly, enabling ASAN also results in some crashes in the original no longer being reported, which indicates a surprising effect of ASAN instrumentation in some cases. Overall, with ASAN, AFL++ found 9.1% (+0.3%) of mutations, Honggfuzz found 9.0% (+0.3%), AFL found 8.5% (+0.1%), and libFuzzer found 7.9% (+0.4%).

ASAN moderately increases the number of mutants killed.

5.4 RQ3: Mutant and Vulnerability Coupling

We analyzed 100 recent CVEs referencing GitHub commits that patch C or C++ files. A mutation is identified as *coupled* to a vulnerability if this mutation reintroduces the vulnerability into the patched program. We will provide detailed results and justifications for this manual analysis along with our code.

For the evaluated CVEs, the way our mutations reintroduce bugs can be classified broadly as: (1) The mutation cleanly reintroduces the bug. (2) The program behavior is modified such that the bug is reintroduced, but some functionality breaks. (3) No mutation that reintroduces the bug can be found.

```
- if (instr[y].size < 29)
+ if (instr[y].size >= 4 && instr[y].size < 29)
```

Listing 2: Patch for CVE-2022-34927.

An example patch for the first category can be seen in Listing 2. The added check for `size >=4` can be defeated by our UNSIGNED GREATER THAN EQUALTO mutation, which will change the right-hand constant to `0` here, effectively reintroducing the original bug without side effects.

```
+ if (nft_chain_is_bound(chain))
+     return -EINVAL;
```

Listing 3: Patch for CVE-2022-39190.

A very common pattern among the studied patches is the introduction of a new branch checking an error condition. These fall mostly in the second category, because the buggy behavior can be triggered by inverting the branch condition (REDIRECT BRANCH), albeit rejecting valid inputs. Listing 3 shows an example for this situation: After applying the mutation, only invalid `chains` are accepted, and valid input is discarded.

```
- length = dir->length;
+ length += dir->length;
```

Listing 4: Patch for CVE-2022-29379.

Patches in the third category sometimes require a more specialized mutation, such as the replacement of `+=` with `=` for the code in Listing 4. In other cases, patches may be impossible to revert due to information being lost (e.g., a deleted function call cannot be re-inserted).

For this analysis, we count vulnerabilities in categories (1) and (2) as coupled to a mutation, as per definition, the vulnerability is reintroduced. As a result, 71 out of 100 analyzed CVEs are covered by our mutations.

Table 5: Results of the benchmark run *when ASAN is enabled*. The *#Mutations* column contains the number of mutations available. *Phase I* represents 24-hour runs, and *Phase II* represents the one-hour runs, and *Total* both combined. The *Covered* represents the number of mutants covered, while *Killed* represents the number of mutants killed. The **combined** rows show the total number of mutants killed by *any* fuzzer.

Program	#Mutations	Fuzzer	Phase I Covered	Phase I Killed	Phase II Covered	Phase II Killed	Total Covered	Total Killed
		afl	88	21	0	0	88	21
		aflpp	88	22	0	0	88	22
cares_name	4822	honggfuzz	88	21	0	1	88	22
		libfuzzer	87	21	0	1	87	22
		combined	88	22			88	22
		afl	938	429	0	1	938	430
		aflpp	941	418	0	17	941	435
cares_parse_reply	4822	honggfuzz	940	427	1	7	941	434
		libfuzzer	906	427	0	2	906	429
		combined	941	432			941	438
		afl	9,981	3,068	87	38	10,068	3,106
		aflpp	11,680	3,458	68	73	11,748	3,531
curl	28779	honggfuzz	10,155	3,057	481	162	10,636	3,219
		libfuzzer	8,833	2,728	130	80	8,963	2,808
		combined	12,349	3,729			12,387	3,774
		afl	6,946	1,916	5,985	1,717	12,931	3,633
		aflpp	12,551	3,657	644	385	13,195	4,042
guetzli	22961	honggfuzz	13,584	4,299	0	106	13,584	4,405
		libfuzzer	9,785	3,263	24	29	9,809	3,292
		combined	13,610	4,414			13,612	4,479
		afl	423	117	0	7	423	124
		aflpp	427	119	0	5	427	124
libevent	17234	honggfuzz	421	116	1	7	422	123
		libfuzzer	418	122	0	3	418	125
		combined	427	124			427	126
		afl	11,492	5,193	18	115	11,510	5,308
		aflpp	11,485	5,209	68	183	11,553	5,392
re2	21407	honggfuzz	11,577	5,171	3	167	11,580	5,338
		libfuzzer	11,003	5,009	4	81	11,007	5,090
		combined	11,586	5,347			11,586	5,438
		afl	5,395	1,047	1	53	5,396	1,100
		aflpp	5,574	1,113	2	47	5,576	1,160
woff2_new	40914	honggfuzz	5,555	1,082	5	75	5,560	1,157
		libfuzzer	4,958	990	1	19	4,959	1,009
		combined	5,631	1,193			5,637	1,247

Table 6: Mutants killed during 24 hour runs on 104 stubborn mutants for each subject.

Prog	Total	libfuzzer	afl	aflpp	honggfuzz
re2	104	0	0	0	0
cares_parse_reply	104	0	0	0	0
woff2_new	104	0	0	0	1
curl	104	1	0	0	0
guetzli	104	0	0	0	1
libevent	104	0	0	0	0

The mutations induced by our mutation operators are coupled to real faults.

5.5 RQ4: Computational Reduction

The computational reduction achieved using supermutants is provided in Table 7. The initial supermutant based reduction of mutations that need to be analyzed was found to be between 1.76× and 19.95× (a mean reduction of 3.8×).

Further, any non-independent mutants recorded during Phase II are given in Table 8. We find that except for `curl`, the number of multi-covered mutants is minimal. For `curl` the interaction between mutations is caused by mutations that result in an error when handling http, these are then retried with the http2 protocol, covering functions and mutations there. This kind of interactions can also occur in utility functions.

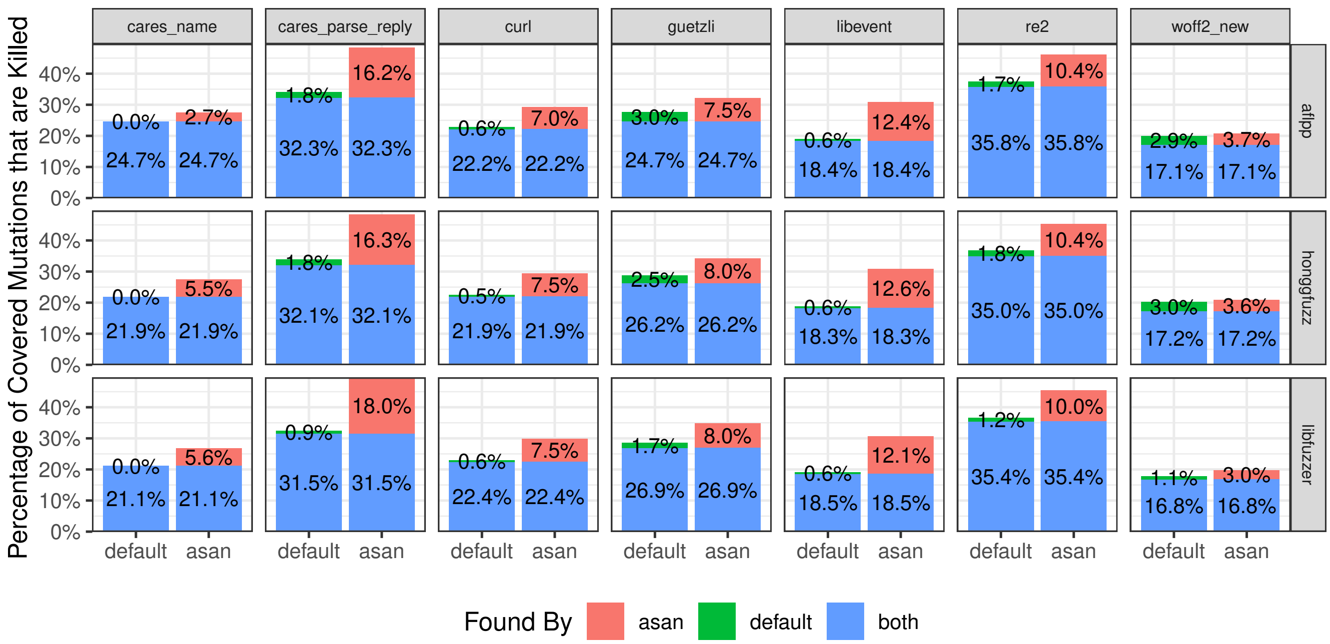

Figure 4: Percentage of covered mutants (in both experiments) that are killed with and without ASAN. "Both" shows mutations killed with or without ASAN, "default" those only found without ASAN and "asan" shows kills only found using ASAN.

Table 7: Computational Reduction, the last row contains the sums and average reduction.

	No. Mutants			CPU years		
Subject	Mutants	Supermutants	Reduction	Naive	Actual	Reduction
`curl`	29,118	5,804	5.02	319.00	4.44	71.85
`guetzli`	22,961	13,040	1.76	251.00	4.76	52.73
`woff2_new`	40,914	5,930	6.90	448.00	2.64	169.70
`cares _name`	4,822	550	8.77	52.00	0.53	98.11
`cares_parse_reply`	4,822	1,288	3.74	52.00	0.76	68.42
`libevent`	17,234	864	19.95	188.00	0.66	284.85
`re2`	21,407	9,670	2.21	234.00	2.40	97.50
	141,278	37,146	3.80	1544.00	16.20	95.31

Our approach achieves a reduction of required CPU time by a factor of 95× on average. This is largely due to the seed generation phase, which allows us to shorten the fuzz runs for supermutants to one hour without overlooking intelligent mutants, as our 24-hour runs show.

> *On average, our optimizations speed up mutation analysis by a factor of* 95×.

6 Discussion

Our paper demonstrates how to perform a principled, yet practical comparison of two fuzzers to determine whether (1) using one is better than using the other, or (2) using both together is better than using each in isolation (Figure 3). Furthermore, we can measure the improvement in a fuzzer compared to an older version without biasing the results on previously discovered faults. Additionally, as our approach can account for sanitizers and other strong oracles, we enable a more principled comparison of oracles. In summary, with this paper we hope to encourage fuzzing researchers to develop better fuzzers without being unduly influenced by benchmarks.

Mutation analysis is the gold measure of test suite effectiveness in software testing, and comes with solid theory and empirical support. The only restriction in using it for fuzzing so far has been the computational cost associated. We have taken the first step in solving this issue, making mutation analysis practical for fuzzing. As proof that our approach indeed works in the real world, we have compared four popular fuzzers on seven programs provided by the OSS-Fuzz benchmark, the results of which we now present.

Our experiments largely confirm the fuzzer ranking established in previous literature [7, 34, 6]. AFL++ leads the board, clearly outperforming AFL and libFuzzer (Honggfuzz was not part of the evaluation in [6]). Furthermore, our evaluation shows again that the fuzzers are very similar and either AFL++ or Honggfuzz represent a solid choice. As expected, the fuzzers in our evaluation have very limited targeted bug finding capability, since there is nearly no improvement in the second phase. The re-evaluation with ASAN demonstrates the importance of improved oracles, as the number of killed mutants increases. It also shows that our approach is indeed capable of evaluating the complete toolchain.

Implications. Why should we trust mutation analysis any more than the numerous metrics out there such as coverage measures [64], defect based benchmarks[7, 33], etc.?

As discussed in Section 2.3.2, mutants that remain alive represent possible undetected faults in the program, and with the detection of each new mutant, that possibility decreases monotonically. That is, unlike coverage, mutations alive is a good proxy measure for residual defects in a program. Compared to defect based benchmarks, mutation analysis minimizes bias and manual effort. Finally, the mutations themselves can be studied to provide examples to improve fuzzers, without the limitations of coverage or the bias of defect based benchmarks.

Our results call for a reorientation of priorities in security testing. We have focused on fuzzing so far. However, we need to move beyond simple fuzzers using weak crash oracles, and look at how to produce stronger oracles that can identify other behavior divergences. This is not simple, however, and is known as the *oracle problem* in software testing [65]. The difficulty is that there is no general way to identify and extract the *intended logic* of a given program. A number of promising approaches exist that can help. These include: (1) metamorphic relations [66, 67] and equivalence modulo inputs [68]; (2) differential oracles [69, 70]; and (3) invariants such as those mined by Daikon [71] and the like. We believe that future research should also be spent on improving the effectiveness of these techniques rather than just pursuing the ever shrinking returns on specifically constructing inputs that induce crashes.

7 Limitations and Future Work

Our work is subject to the following important limitations.

Missing patterns and faults. We rely on a set of mutation operators that we mined from real faults, which are unlikely to be exhaustive. This can be mitigated by future adaption of the supported mutations.

Cost of mutation analysis. While we reduced the cost of applying mutation analysis to fuzzers by nearly two orders, a further reduction would still improve the practicality of this approach. One such approach could involve sampling mutations, to trade of compute time with accuracy.

Coupling effect. We rely on the *coupling effect* hypothesis in mutation analysis to ensure that the mutants we generate are similar to real faults. The coupling effect is well attested in literature [40, 41]. However, relying on the coupling effect ignores subtle faults due to fault-interactions (between faults). While there is some evidence that such interactions are rare [72], they are still important to address. One direction is exploring a larger neighborhood, with multiple mutations in a mutant. Unlike supermutants, however, we need mutations that interact, and *callability* that we used for supermutants can provide a first level approximation. Indeed, given that mutation analysis is a form of fuzz-testing the software test suite, the fuzzing community may be able to make better progress here.

Allocated Time for Phases. It is possible that a future fuzzer may not attempt to expand the coverage front, and instead scan for possible vulnerability patterns instead using static analysis. In such a case, the time provided for developing the coverage seed may not be useful to the fuzzer, and may lead to unfair comparison. While no such fuzzers currently exist, how to fairly compare such fuzzers without the overhead of a full mutation analysis is an open question. We note that while the coverage optimization may not work, the supermutant optimization will work even in such cases.

Supermutants. We use supermutants to check whether any fuzzer can find mutations that were not covered in the initial seed files. This however ignores fault masking which may make such faults unreachable. This is a limitation of current work. For future work, we will be looking at better ways of identifying independent mutations using control-flow and data-flow analysis.

8 Related Work

The comparison and evaluation of fuzzers is an important foundation to meaningfully improve fuzzers. In recent work, this research area has seen a lot of activity. A number of fuzzing platforms exist [33, 7, 34] that seek to provide a way to compare fuzzers under a common framework.

One such approach is LAVA [33], a tool to insert bugs into subjects. These bugs cause crashes when triggered by finding specific values in unused parts of user controlled input.

Later analysis showed that the introduced bugs are dissimilar to real-world vulnerabilities [2], are not coupled to real faults (reported CVEs) [11], tend to overfit [9], and are "solved" by modern fuzzers [73]. In comparison, bugs from mutation analysis are not guaranteed to be triggerable. However, this is a trade-off making it possible to create a comprehensive set of bugs spanning trivially detectable to subtle and hard to detect ones. A similar approach that can insert bugs into subjects is Evil Coder [74]. Potentially vulnerable source code locations are detected using data flow analysis, focusing on user controlled inputs that lead to sensitive functions. A bug is introduced by removing security relevant checks, such as input sanitization. While our approach is not as targeted, mutation testing will not only generate similar bugs but also a wider range of bugs.

The challenge binaries of the Cyber Grand Challenge (CGC) [75] are also sometimes used for comparison of fuzzers [2]. The binaries were especially created for the CGC. Hence, challenge binaries necessarily have a bias to be used in the CGC and consist mostly of command line tools.

A recent benchmarking approach is Magma [7]. It uses real-world vulnerabilities and re-inserts them into newer version of the projects. Additionally, it provides an assertion that tests if an input results in a state that would trigger the bug. This assertion is used to measure bug detection capability. As in other benchmarks, the number of bugs is limited because of the manual effort to port them to the current version and has a necessary bias towards bugs that can be re-inserted.

Another project is Fuzzbench [34], a service running on Google infrastructure. Fuzzbench provides both a coverage and bug based benchmarks. The bugs are based on real world projects and manually inserted into a subject, combining as many bugs as possible, with the goal of reducing required computation resources. As before, the bug source causes a bias, and fault interaction is a concern.

FIXREVERTER [6] is a method that mines a restricted set of syntactic patterns from recent bugfixes that were associated with vulnerabilities, and injects these bugs where the bug inducibility can be guaranteed. This approach suffers from several limitations. First, the researchers could identify only three general patterns accounting for 170 CVEs from a study of over 814 CVEs (20.9%). Second, using patterns with semantic analysis to guarantee bug inducibility restricts the number of fixes that can be reverted. Such a guarantee also limits what kinds of bugs can be simulated, as the specific bug patterns and corresponding bug semantics that were mined represent only a small fraction of the possible bugs that can be present in a given program (in contrast to mutation analysis). This might also open the door to fine-tuning fuzzers specifically to identifying such behaviors. These drawbacks reduce the diversity of bugs and thus the effectiveness of the benchmark [76, 77]. Finally, FIXREVERTER does not address the issue of fault-interactions and fault-masking.

Böhme et al. suggest that coverage-based benchmarking can be unreliable based on a comparison with curated bugs [8]. The paper illustrates the difficulty we face when we rely on an external source of bugs. In particular, because the bugs that the researchers rely on are external, the distribution of such bugs are not related to the actual possibility of bugs in the tested program. To illustrate this consider the following thought experiment: Given a benchmark program that accepts input as JSON, and a blackbox random fuzzer. The fuzzer will find lots of crash bugs in the JSON parser itself but little in the program logic. Any ranking using this source of bugs will favor fuzzers that finds bugs in the JSON parser when compared to, say, a grammar fuzzer that reaches the program internals. Hence, the ranking based on finding such bugs is not a reliable indicator of fuzzer quality. This is precisely what (unbiased) mutation analysis aims to correct.

9 Conclusion

As fuzzing budget is limited, it is important to use fuzzers that are better at finding faults. The available benchmarks are, however, limited or biased towards known bugs, and are susceptible to overfitting and fine-tuning.

In this paper, we demonstrate how the gold standard for measuring test suite quality—mutation analysis—can be adapted for fuzzing. We show that two techniques, eliminating coverage mutants using static seed files and using supermutants for the remaining evaluation, can reduce the cost of mutation analysis by a factor of almost 100×, making it feasible to use for fuzzing. We investigated security faults and converted the identified patterns into security specific mutation operators, which were used for evaluation. Using mutation analysis, practitioners are no longer limited to specific curated benchmarks. Instead, practitioners can evaluate fuzzers on the programs from a specific domain before allocating resources for fuzzing.

Our evaluation demonstrates that with our technique, mutation analysis can now be used for comparing popular fuzzers in real-world programs. Using mutation analysis ensures that the practitioners can rely on the solid theory and decades of empirical research, leading to better fuzzers and sanitizers. Furthermore, the fine-grained results from mutation analysis can directly help fuzzing practitioners to understand the deficiencies in current approaches and take steps to correct them.

References

[1] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections". *IEEE Software* 38.3 (2021), pp. 79–86.

[2] George Klees et al. “Evaluating Fuzz Testing”. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada: ACM, 2018, pp. 2123–2138.

[3] Tielei Wang et al. “TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection”. *2010 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, 2010, pp. 497–512.

[4] Sebastian Österlund et al. “ParmeSan: Sanitizer-guided Greybox Fuzzing”. *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2289–2306.

[5] Rahul Gopinath, Carlos Jensen, and Alex Groce. “Code coverage for suite evaluation by developers”. *ICSE ’14: 36th International Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 72–82.

[6] Zenong Zhang et al. “FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing”. *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA, USA: USENIX Association, 2022, pp. 3699–3715.

[7] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. “Magma: A Ground-Truth Fuzzing Benchmark”. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. “On the Reliability of Coverage-Based Fuzzer Benchmarking”. *ICSE ’22: 44th International Conference on Software Engineering*. Pittsburgh, PA, USA: ACM, 2022, pp. 1621–1633.

[9] Andreas Zeller, Sascha Just, and Kai Greshake. *When Results Are All That Matters: The Case of the Angora Fuzzer*. 2019. URL: https://andreas-zeller.info/2019/10/10/when-results-are (visited on 10/12/2022).

[10] Andreas Zeller, Sascha Just, and Kai Greshake. *When Results Are All That Matters: Consequences*. 2019. URL: https://andreas-zeller.info/2019/10/17/when-results-are (visited on 10/12/2022).

[11] Joshua Bundt et al. “Evaluating Synthetic Bugs”. *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. Virtual Event, Hong Kong: ACM, 2021, pp. 716–730.

[12] Yin Chan, Ashok Sudarsanam, and Andrew Wolfe. “The effect of compiler-flag tuning on SPEC benchmark performance”. *ACM SIGARCH Computer Architecture News* 22.4 (1994), pp. 60–70.

[13] Mike Papadakis et al. “Mutation testing advances: an analysis and survey”. *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[14] Michele Tufano et al. “Learning How to Mutate Source Code from Bug-Fixes”. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Cleveland, OH, USA: IEEE, 2019, pp. 301–312.

[15] Zakir Durumeric et al. “The Matter of Heartbleed”. *Proceedings of the 2014 Conference on Internet Measurement Conference*. Vancouver, BC, Canada: ACM, 2014, pp. 475–488.

[16] Synopsis Editorial Team. *Understanding the Apple ‘goto fail;’ vulnerability*. 2014. URL: https://www.synopsys.com/blogs/software-security/underst (visited on 10/12/2022).

[17] Tatum Hunter and Gerrit De Vynck. “The ‘most serious’ security breach ever is unfolding right now. Here’s what you need to know.” *The Washington Post* (2021). URL: https://www.washingtonpost.com/technology/2021/12/20/log (visited on 10/12/2022).

[18] J. H. Andrews, L. C. Briand, and Y. Labiche. “Is mutation an appropriate tool for testing experiments?” *ICSE ’05: 27th International Conference on Software Engineering*. St. Louis, MO, USA: ACM Press, 2005, p. 402.

[19] René Just et al. “Are mutants a valid substitute for real faults in software testing?” *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[20] Mike Papadakis et al. “Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults”. *ICSE ’18: 40th International Conference on Software Engineering*. Gothenburg, Sweden: ACM, 2018, pp. 537–548.

[21] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. “Mutation Analysis for Security Tests Qualification”. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. Windsor, UK: IEEE, 2007, pp. 233–242.

[22] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. “Mutation-Based Test Generation from Security Protocols in HLPSL”. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. Berlin, Germany: IEEE, 2011, pp. 240–248.

[23] Daniel Woodraska, Michael Sanford, and Dianxiang Xu. "Security mutation testing of the FileZilla FTP server". *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*. TaiChung, Taiwan: ACM Press, 2011, p. 1425.

[24] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. "Testing Security Policies: Going Beyond Functional Testing". *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. Trollhattan, Sweden: IEEE, 2007, pp. 93–102.

[25] Thomas Loise et al. "Towards Security-Aware Mutation Testing". *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Tokyo, Japan: IEEE, 2017, pp. 97–102.

[26] Amit Seal Ami et al. "µSE: Mutation-Based Evaluation of Security-Focused Static Analysis Tools for Android". *ICSE-Companion '21: 43rd International Conference on Software Engineering: Companion*. Madrid, Spain: IEEE, 2021, pp. 53–56.

[27] Lu Yu et al. "Vulnerability-oriented directed fuzzing for binary programs". *Scientific Reports* 12.1 (2022), pp. 1–13.

[28] Rahul Gopinath, Bjorn Mathis, and Andreas Zeller. "If You Can't Kill a Supermutant, You Have a Problem". *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Västerås, Sweden: IEEE, 2018, pp. 18–24.

[29] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA, USA: USENIX Association, 2012, pp. 309–318.

[30] IEEE. "IEEE Standard Classification for Software Anomalies". *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* 0 (2010), pp. 1–23.

[31] Yiqun T. Chen et al. "Revisiting the relationship between fault detection, test adequacy criteria, and test set size". *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Virtual Event, Australia: ACM, 2020, pp. 237–249.

[32] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. "Fuzzing with Data Dependency Information". *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. Genoa, Italy: IEEE, 2022, pp. 286–302.

[33] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, 2016, pp. 110–121.

[34] Jonathan Metzman et al. "FuzzBench: an open fuzzer benchmarking platform and service". *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece: ACM, 2021, pp. 1393–1403.

[35] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Mutations: How Close are they to Real Faults?" *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Naples, Italy: IEEE, 2014, pp. 189–200.

[36] A. Jefferson Offutt. "Investigations of the software testing coupling effect". *ACM Transactions on Software Engineering and Methodology* 1.1 (1992), pp. 5–20.

[37] K. S. How Tai Wah. "A theoretical study of fault coupling". *Software Testing, Verification and Reliability* 10.1 (2000), pp. 3–45.

[38] K. S. How Tai Wah. "Theoretical Insights into the Coupling Effect". *Mutation Testing for the New Century*. Ed. by W. Eric Wong. Boston, MA, USA: Springer US, 2001, pp. 62–70.

[39] K. S. How Tai Wah. "An analysis of the coupling effect I: single test data". *Science of Computer Programming* 48.2 (2003), pp. 119–161.

[40] Rahul Gopinath, Carlos Jensen, and Alex Groce. "The Theory of Composite Faults". *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan: IEEE, 2017, pp. 47–57.

[41] René Just et al. "Are mutants a valid substitute for real faults in software testing?" *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[42] Goran Petrovic et al. "Does Mutation Testing Improve Testing Practices?" *ICSE '21: 43rd International Conference on Software Engineering*. Madrid, Spain: IEEE, 2021, pp. 910–921.

[43] Thierry Titcheu Chekam et al. "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption". *ICSE '17: 39th International Conference on Software Engineering*. Buenos Aires, Argentina: IEEE, 2017, pp. 597–608.

[44] René Just, Michael D. Ernst, and Gordon Fraser. "Efficient mutation analysis by propagating and partitioning infected execution states". *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. San Jose, CA, USA: ACM Press, 2014, pp. 315–326.

[45] W.E. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering* 8.4 (1982), pp. 371–379.

[46] Susumu Tokumoto et al. "MuVM: Higher Order Mutation Analysis Virtual Machine for C". *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Chicago, IL, USA: IEEE, 2016, pp. 320–329.

[47] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Topsy-Turvy: a smarter and faster parallelization of mutation analysis". *ICSE '16: 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, 2016, pp. 740–743.

[48] Bo Wang et al. "Faster mutation analysis via equivalence modulo states". *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Santa Barbara, CA, USA: ACM, 2017, pp. 295–306.

[49] Ali Ghanbari and Andrian Marcus. "Toward Speeding up Mutation Analysis by Memoizing Expensive Methods". *ICSE-NIER '21: 43rd International Conference on Software Engineering: New Ideas and Emerging Results*. Madrid, Spain: IEEE, 2021, pp. 71–75.

[50] Rahul Gopinath, Philipp Görz, and Alex Groce. *Mutation Analysis: Answering the Fuzzing Challenge*. 2022. arXiv: 2201.11303[cs].

[51] Iftekhar Ahmed et al. "Can testedness be effectively measured?" *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: ACM, 2016, pp. 547–558.

[52] Terence Tao. *An introduction to measure theory*. Vol. 126. American Mathematical Society Providence, RI, 2011.

[53] Mauro Pezze and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[54] Marcel Böhme. "Assurances in Software Testing: A Roadmap". *ICSE-NIER '19: 41st International Conference on Software Engineering: New Ideas and Emerging Results*. Montreal, QC, Canada: IEEE, 2019, pp. 5–8.

[55] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage". *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE. 2009, pp. 220–229.

[56] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. "All-uses vs mutation testing: an experimental comparison of effectiveness". *Journal of Systems and Software* 38.3 (1997), pp. 235–253.

[57] Sahitya Kakarla, Selina Momotaz, and Akbar Siami Namin. "An evaluation of mutation and data-flow testing: A meta-analysis". *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 366–375.

[58] Gary Kaminski, Paul Ammann, and Jeff Offutt. "Improving logic-based testing". *Journal of Systems and Software* 86.8 (2013), pp. 2002–2012.

[59] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[60] Xiangjuan Yao, Mark Harman, and Yue Jia. "A study of equivalent and stubborn mutation operators using human analysis of equivalence". *ICSE '14: 36th International Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 919–930.

[61] *CWE VIEW: Weaknesses in Software Written in C*. 2021. URL: https://cwe.mitre.org/data/definitions/658.html (visited on 10/12/2021).

[62] *CWE VIEW: Weaknesses in Software Written in C++*. 2021. URL: https://cwe.mitre.org/data/definitions/659.html (visited on 10/12/2021).

[63] *OSS-Fuzz*. 2021. URL: https://google.github.io/oss-fuzz/ (visited on 10/12/2022).

[64] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. "Coverage and Its Discontents". *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Portland, OR, USA: ACM, 2014, pp. 255–268.

[65] Earl T. Barr et al. "The Oracle Problem in Software Testing: A Survey". *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525.

[66] Sergio Segura et al. "A Survey on Metamorphic Testing". *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 805–824.

[67] Tsong Yueh Chen et al. "Metamorphic Testing for Cybersecurity". *Computer* 49.6 (2016), pp. 48–55.

[68] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs". *ACM SIGPLAN Notices* 49.6 (2014), pp. 216–226.

[69] William M McKeeman. "Differential testing for software". *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[70] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. "Perception and Practices of Differential Testing". *ICSE-SEIP '19: 41st International Conference on Software Engineering: Software Engineering in Practice*. Montreal, QC, Canada: IEEE, 2019, pp. 71–80.

[71] Michael D. Ernst et al. "The Daikon system for dynamic detection of likely invariants". *Science of Computer Programming* 69.1 (2007), pp. 35–45.

[72] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. "Software fault interactions and implications for software testing". *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421.

[73] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". *Network and Distributed System Security (NDSS) Symposium 2019*. San Diego, CA, USA: Internet Society, 2019.

[74] Jannik Pewny and Thorsten Holz. "EvilCoder: Automated Bug Insertion". *Proceedings of the 32nd Annual Conference on Computer Security Applications*. Los Angeles, CA, USA: ACM, 2016, pp. 214–225.

[75] DARPA. *Cyber Grand Challenge*. 2016. URL: https://www.ll.mit.edu/research-and-development/cyber-security-and-information-sciences/cyber-grand-challenge (visited on 10/12/2022).

[76] Rahul Gopinath et al. "On the limits of mutation reduction strategies". *ICSE '16: 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, 2016, pp. 511–522.

[77] Rahul Gopinath et al. "Mutation Reduction Strategies Considered Harmful". *IEEE Transactions on Reliability* 66.3 (2017), pp. 854–874.

A Appendix

Table 8: Number of mutants that were covered together with other mutants (i.e., mutants wrongly thought independent).

Program	afl	aflpp	honggfuzz	libfuzzer
cares_name	4	0	0	0
cares_parse_reply	2	4	4	0
curl	4,850	5,836	4,851	3,852
guetzli	10	24	16	0
libevent	0	2	0	0
re2	39	66	37	47
woff2_new	26	46	56	48

Table 9: Results for each mutation type and fuzzer, for both default and ASAN.

Mutation Type	Fuzzer	Cov Def	Cov Asan	Kill Def	Kill Asan	Mutation Type	Fuzzer	Cov Def	Cov Asan	Kill Def	Kill Asan
MALLOC	honggfuzz	41	41	20	21	SIGNED LESS THAN	honggfuzz	672	671	111	130
	afl	41	41	21	21		afl	447	650	104	118
	aflpp	41	41	21	21		aflpp	670	665	116	129
	libfuzzer	40	41	14	21		libfuzzer	571	557	100	111
SIGNED GREATER THAN	honggfuzz	725	719	42	43	SIGNED LESS THAN EQUALTO	honggfuzz	2	2	0	0
	afl	434	697	33	42		afl	2	2	0	0
	aflpp	719	711	38	42		aflpp	2	2	0	0
	libfuzzer	614	608	28	34		libfuzzer	2	2	0	0
SIGNED GREATER THAN EQUALTO	honggfuzz	4	4	0	0	FREE FUNCTION ARGUMENT	honggfuzz	944	943	940	939
	afl	4	4	0	0		afl	923	922	918	915
	aflpp	4	4	0	0		aflpp	981	979	976	973
	libfuzzer	3	3	0	0		libfuzzer	876	874	870	868
PTHREAD MUTEX	honggfuzz	3	3	0	0	SIGNED TO UNSIGNED	honggfuzz	1,407	1,416	35	43
	afl	3	3	0	0		afl	825	1,365	32	39
	aflpp	3	3	0	0		aflpp	1,393	1,396	38	44
	libfuzzer	3	2	0	0		libfuzzer	1,201	1,200	32	38
UNSIGNED TO SIGNED	honggfuzz	1,981	1,981	52	60	SWITCH SHIFT	honggfuzz	825	828	2	9
	afl	1,470	1,919	46	54		afl	568	805	6	9
	aflpp	1,984	1,981	53	57		aflpp	903	897	5	13
	libfuzzer	1,736	1,739	40	51		libfuzzer	718	719	3	9
CALLOC	honggfuzz	1	1	1	1	DELETE LOCAL STORE	honggfuzz	273	276	107	117
	afl	1	1	1	1		afl	203	260	96	104
	aflpp	1	1	1	1		aflpp	277	278	108	119
	libfuzzer	1	1	1	1		libfuzzer	244	247	100	104
UNSIGNED LESS THAN	honggfuzz	1,125	1,121	180	179	UNSIGNED GREATER THAN	honggfuzz	819	816	77	95
	afl	885	1,085	167	164		afl	625	798	71	85
	aflpp	1,121	1,120	182	170		aflpp	825	823	68	88
	libfuzzer	960	938	145	135		libfuzzer	728	719	54	61
UNSIGNED LESS THAN EQUALTO	honggfuzz	15	15	0	0	UNSIGNED GREATER THAN EQUALTO	honggfuzz	25	25	0	0
	afl	12	13	0	0		afl	23	25	0	0
	aflpp	15	15	0	0		aflpp	25	25	0	0
	libfuzzer	13	13	0	0		libfuzzer	25	25	0	0
COMPARE EQUAL TO	honggfuzz	2,992	2,972	1,074	1,322	SNPRINTF	honggfuzz	14	14	0	0
	afl	2,367	2,877	1,055	1,256		afl	14	14	0	0
	aflpp	3,060	3,050	1,103	1,356		aflpp	14	14	0	0
	libfuzzer	2,620	2,585	974	1,187		libfuzzer	13	13	0	0
NEW ARRAY	honggfuzz	21	21	2	13	SWITCH PLUS MINUS	honggfuzz	5,692	5,687	1,916	2,068
	afl	2	21	2	13		afl	5,001	5,467	1,845	1,899
	aflpp	21	21	2	13		aflpp	5,928	5,889	1,884	1,961
	libfuzzer	21	21	2	13		libfuzzer	4,716	4,626	1,495	1,639
REDIRECT BRANCH	honggfuzz	9,963	9,959	3,311	4,058	DELETE FUNCTION ARGUMENT	honggfuzz	469	469	468	468
	afl	8,120	9,640	3,192	3,731		afl	466	466	465	465
	aflpp	10,061	10,084	3,335	4,062		aflpp	469	469	468	468
	libfuzzer	8,534	8,407	2,819	3,515		libfuzzer	447	447	445	445
DELETE STORE PATTERN	honggfuzz	8,471	8,441	2,002	2,388	DELETE CALL INSTRUCTION PATTERN	honggfuzz	2,022	1,946	192	912
	afl	6,935	8,112	1,885	2,249		afl	1,545	1,884	184	788
	aflpp	8,560	8,529	1,995	2,386		aflpp	2,122	2,040	193	946
	libfuzzer	7,060	7,025	1,752	2,097		libfuzzer	1,699	1,622	166	799
REASSIGN STORE INSTRUCTION	honggfuzz	2,854	2,855	404	464						
	afl	2,421	2,727	368	427						
	aflpp	2,878	2,859	400	447						
	libfuzzer	2,300	2,259	354	392						

Pattern Name	Description	Procedure
MALLOC	Mutating all malloc calls to achieve buffer overflow/out of bounds errors.	We decrease allocated memory byte_size in the malloc call by 16.
FGETS MATCH BUFFER SIZE	Mutating all fgets calls to achieve buffer overflow errors.	We increase the size (n) parameter in the fgets call by increasing the value by 1 and then multiplying it by 5. E.g. 4->5->25.
SIGNED LESS THAN	Mutating all '<' comparisons either between two integer pointers or between 1 signed integer variable and an integer to achieve overflow errors.	For pointer comparison, 8*4=32 is added to the right hand side pointer in the comparison. For integer comparison, the integer on the right hand side is squared if larger than 1024 or smaller than 2, else 32 is added.
SIGNED GREATER THAN	Mutating all '>' comparisons either between two integer pointers or between 1 signed integer variable and an integer to achieve underflow errors.	For pointer comparison, 8*4=32 is subtracted from the right hand side pointer in the comparison. For integer comparison, either the sqrt is taken for integers > 1024*1024, halved for integers > 1024 and either 0 is returned or 32 is substracted, whatever gives the largest result.
SIGNED LESS THAN EQUALTO	Mutating all '<=' comparisons either between two integer pointers or between 1 signed integer variable and an integer to achieve overflow errors.	For pointer comparison, 8*4=32 is added to the right hand side pointer in the comparison. For integer comparison, the integer on the right hand side is squared if larger than 1024 or smaller than 2, else 32 is added.
SIGNED GREATER THAN EQUALTO	Mutating all '>=' comparisons either between two integer pointers or between 1 signed integer variable and an integer to achieve underflow errors.	For pointer comparison, 8*4=32 is subtracted from the right hand side pointer in the comparison. For integer comparison, either the sqrt is taken for integers > 1024*1024, halved for integers > 1024 and either 0 is returned or 32 is substracted, whatever gives the largest result.
FREE FUNCTION ARGUMENT	Mutating all functions that receive a pointer type function argument to achieve double free and possibly illegal memory access errors.	We check for functions that receive a pointer type argument. Before returning at the end of the function, one argument per mutant is freed.
PTHREAD MUTEX	Mutating all pthread_lock and pthread_unlock calls to achieve data races errors.	We remove all pthread_lock and pthread_unlock calls in a function per mutant.
ATOMIC CMP XCHG	Mutating all atomic compare exchanges to achieve data races.	If we have at least one atomic cmpxchg instruction, we replace all atomic cmpxchg return success values (the element with index 1 in the result array) by 1 per function.
ATOMICRMW REPLACE	Mutating all atomicrmw instructions to achieve data races.	Takes the given atomic instruction and replaces it with its non-atomic counterpart for the following instructions: ADD, SUB, AND, OR, XOR, FADD, FSUB. For other operators no mutation is done, the mutant is equivalent.
SIGNED TO UNSIGNED	Mutating all signed integer comparisons to achieve overflow and out of bound errors.	Each of the four integer comparison predicates - ICMP_SGT, ICMP_SGE, ICMP_SLT, ICMP_SLE are transformed into the corresponding unsigned predicates - ICMP_UGT, ICMP_UGE, ICMP_ULT, ICMP_ULE respectively.
UNSIGNED TO SIGNED	Mutating all unsigned integer comparisons to achieve overflow and out of bounds errors.	Each of the four integer comparison predicates - ICMP_UGT, ICMP_UGE, ICMP_ULT, ICMP_ULE are transformed into the corresponding unsigned predicates - ICMP_SGT, ICMP_SGE, ICMP_SLT, ICMP_SLE respectively.
SWITCH SHIFT	Mutating all shift calls to achieve overflow and out of bounds errors.	Replaces an arithmetic shift with a logical shift and vice versa.
CALLOC	Mutating all calloc calls to achieve overflow and out of bounds errors.	The size parameter's value is decreased by 16.
DELETE LOCAL STORE	Mutating all stores on a local variable in one function to achieve uninitialization errors.	The store call is removed.
UNSIGNED LESS THAN	Mutating all '<' comparisons either between two integer pointers or between 1 unsigned integer variable and an integer to achieve overflow errors.	For pointer comparison, 8*4=32 is added to the right hand side pointer in the comparison. For integer comparison, the integer on the right hand side is squared if larger than 1024 or smaller than 2, else 32 is added.
UNSIGNED GREATER THAN	Mutating all '>' comparisons either between two integer pointers or between 1 unsigned integer variable and an integer to achieve underflow errors.	For pointer comparison, 8*4=32 is subtracted from the right hand side pointer in the comparison. For integer comparison, either the sqrt is taken for integers > 1024*1024, halved for integers > 1024 and either 0 is returned or 32 is substracted, whatever gives the largest result.
UNSIGNED LESS THAN EQUALTO	Mutating all '<=' comparisons either between two integer pointers or between 1 unsigned integer variable and an integer to achieve overflow errors.	For pointer comparison, 8*4=32 is added to the right hand side pointer in the comparison. For integer comparison, the integer on the right hand side is squared if larger than 1024 or smaller than 2, else 32 is added.
UNSIGNED GREATER THAN EQUALTO	Mutating all '>=' comparisons either between two integer pointers or between 1 unsigned integer variable and an integer to achieve underflow errors.	For pointer comparison, 8*4=32 is subtracted from the right hand side pointer in the comparison. For integer comparison, either the sqrt is taken for integers > 1024*1024, halved for integers > 1024 and either 0 is returned or 32 is substracted, whatever gives the largest result.
INET ADDR FAIL WITHOUTCHECK	Mutating all calls to the libc function inet_addr to achieve unhandled non-established connection errors.	Replaces all uses of the function return value to the failure value. Also removes the function call from the corpus as a fail of the function call should be simulated. Furthermore, the comparison instructions are flipped, s.t. on failure the 'correct' path is taken, i.e. we simulate a missing check for the error return value.
COMPARE EQUAL TO	Mutating all '==' comparisons between two integers to '='.	The value of integer on the right hand side is assigned to the variable on the left. The condition passes and the inside block is executed as long as the value on the RHS is not equal to 0.
PRINTF	Mutating printf such that the format string gets already filled and then plainly printed.	Mutating printf such that the format string is already filled on printing, so instead of calling printf('%d %s', 10, string); we simulate the call printf('10 <string-value>');. This might cause illegal memory accesses and printing of secrets if the string argument is user controlled.

Continued on next page

Pattern Name	Description	Procedure
SPRINTF	Mutating sprintf such that the format string gets already filled and then plainly printed.	Mutating sprintf such that the format string is already filled on printing, so instead of calling sprintf(’%d %s’, buffer, 10, string); we simulate the call sprintf(’10 <string-value>’, buffer);. This might cause illegal memory accesses and printing of secrets if the string argument is user controlled.
SNPRINTF	Mutating snprintf such that the format string gets already filled and then plainly printed.	Mutating snprintf such that the format string is already filled on printing, so instead of calling snprintf(’%d %s’, size, buffer, 10, string); we simulate the call snprintf(’10 <string-value>’, size, buffer);. This might cause illegal memory accesses and printing of secrets if the string argument is user controlled.
NEW ARRAY	Mutating new[] in (only) cpp files such that the array is allocated lesser memory	We decrease allocated memory size in the ’new’ call by 5 units.
SWITCH PLUS MINUS	Changing a ’+’ operator to a ’-’ operator and vice versa.	Changing a ’+’ operator to a ’-’ operator regardless for integer and floating point numbers.
REDIRECT BRANCH	Negate the result of the branching condition before branching.	Redirecting the control flow by negating the result of the condition before branching.
DELETE FUNCTION ARGUMENT	Mutating all functions in (only) cpp files that receive a pointer type function argument to achieve double delete and possibly illegal memory access errors. N.B. - Can possibly lead to a memory leak when delete is called for arrays instantiated with new[]	We check for functions that receive a pointer type argument. Before returning at the end of the function, one argument per mutant is deleted.
DELETE STORE PATTERN	Deletes all store instructions one by one to simulate a forgotten variable assignment.	Find a store instruction and delete it. As there are no further dependencies on the store, there is nothing else to do.
DELETE CALL INSTRUCTION PATTERN	Deletes all call instructions without return value assignment one by one to simulate a forgotten call to a function.	Find a call instruction without return value assignment and delete it. As there are no further dependencies on the call instruction, there is nothing else to do.
REASSIGN STORE INSTRUCTION	Reassigns the value of a previous store with the same type in this store.	Checks if in this basic block is another store with the same types used and assigns the first operand of the previous store to the memory location denoted by the second operand of the store we are currently at.

Table 10: List of all mutations used in our study.