Magma: A Ground-Truth Fuzzing Benchmark

Ahmad Hazimeh
EPFL

Mathias Payer
EPFL

ABSTRACT

High scalability and low running costs have made fuzz testing the de-facto standard for discovering software bugs. Fuzzing techniques are constantly being improved in a race to build the ultimate bug-finding tool. However, while fuzzing excels at finding bugs, comparing fuzzer performance is challenging due to the lack of metrics and benchmarks. Crash count, the most common performance metric, is inaccurate due to imperfections in de-duplication methods and heuristics. Moreover, the lack of a unified set of targets results in ad hoc evaluations that inhibit fair comparison.

We tackle these problems by developing Magma, a ground-truth fuzzer evaluation framework enabling uniform evaluations and comparison. By introducing real bugs into real software, Magma allows for a realistic evaluation of fuzzers against a broad set of targets. By instrumenting the injected bugs, Magma also enables the collection of bug-centric performance metrics independent of the fuzzer. Magma is an open benchmark consisting of seven targets that perform a variety of input manipulations and complex computations, presenting a challenge to state-of-the-art fuzzers.

We evaluate six popular mutation-based grey-box fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, and honggfuzz) against Magma over 26,000 CPU-hours. Based on the number of bugs reached, triggered, and detected, we draw conclusions about the fuzzers' exploration and detection capabilities. This provides insight into fuzzer comparisons, highlighting the importance of ground truth in performing more accurate and meaningful evaluations.

1 INTRODUCTION

Fuzz testing [29] is a form of brute-force bug discovery. It is the process of feeding a system large numbers of generated inputs in rapid succession, and monitoring the output for faults. It is an inherently sound but incomplete process (given finite resources) for finding bugs in software. State-of-the-art fuzzers rely on "crashes" to mark faulty program behavior. Such crashes are often triggered by hardware checks, OS signals, or sanitizers to signal a security policy violation. The existence of a crash is generally symptomatic of a bug (soundness), but the lack of a crash does not necessarily mean the program is bug-free (incompleteness). In the context of software testing, fuzzing has been used to find thousands of bugs in open-source [2] and commercial off-the-shelf [4, 5, 38] software. The success of the American Fuzzy Lop (AFL) [55] fuzzer—the leading coverage-guided grey-box fuzzer—has been hailed by security professionals and academic researchers alike. AFL has contributed to finding bugs in over 160 software products [56].

Research on fuzz testing has since been given new life, with tens of papers detailing new techniques to improve performance and increase the chance of finding bugs in a reasonable time-frame [7, 8, 10, 33, 39, 46, 54]. In order to highlight improvements, a fuzzer is evaluated against a set of target programs. These target programs can be sourced from a benchmark suite—such as the Cyber Grand Challenge [9], LAVA-M [12], the Juliet Test Suite [30], and Google's Fuzzer Test Suite [17], oss-fuzz [2], and FuzzBench [16]—or from a set of real-world programs [34]. Performance metrics—including coverage profiles, crash counts, and bug counts—are then collected during the evaluation. Unfortunately, while such metrics can provide insight into a fuzzer's performance, they are often insufficient to compare different fuzzers. Moreover, the lack of a unified set of target programs makes for unfounded comparisons.

Most fuzzer evaluations consider crash count a reasonable metric for assessing fuzzer performance. However, crash count is often inflated [25], even after attempts at de-duplication (e.g., via coverage profiles or stack hashes), highlighting the need for more accurate forms of root-cause identification. Moreover, evaluations are often inconsistent with respect to campaign run times, timeouts, seed files, and target programs [25], prohibiting cross-paper comparisons.

Thus, to enable precise fuzzer evaluation and comparison, a unified set of target programs and metrics should be developed and used as a fuzzing benchmark. The target programs should be realistic and exercise diverse behavior, and the collected metrics should accurately measure the fuzzer's ability to achieve its main objective: *finding bugs*. To understand the rationale behind fuzzing benchmark design, it is important to dissect and classify the evaluated aspects. A good fuzzing strategy must balance two aspects: *program state exploration* and *fault detection and reporting*.

For example, the LAVA-M [12] test suite aims to evaluate the effectiveness of the program exploration aspect by injecting bugs in different execution paths. However, it only injects a single, simple bug type: an out-of-bounds access triggered by a magic value in the input. This bug type does not represent the statefulness and complexity of bugs encountered in real programs. The Cyber Grand Challenge (CGC) [9] sample set provides a wide variety of bugs, suitable for testing the fault detection capabilities of a fuzzer, but the relatively small size and simplicity of these synthetic programs does not enable thorough evaluation of the fuzzer's exploration aspect. BugBench [28] and Google Fuzzer Test Suite (FTS) [17] both present real programs with real bugs, making them potential candidates for testing both aspects, but the sparsity of bugs in each program and the lack of an automatic method for triaging crashes hinder adoption and make them unsuitable for evaluating fuzzers. Google FuzzBench [16] is a fuzzer evaluation platform which relies solely on coverage profiles as a performance metric. The intuition behind this approach is that fuzzers which exercise larger code paths are more likely to discover bugs. However, a previous case-study [25] has shown that there is a weak correlation between coverage-deduplicated crashes and ground-truth bugs, implying that higher coverage does not necessarily indicate better fuzzer effectiveness. Finally, open-source programs and libraries [34] (e.g., GNU binutils [13]) are also often used as fuzz targets. However, the lack of ground-truth knowledge about encountered crashes makes

Technical Report, ,
2020.

it difficult to provide an accurate quantitative evaluation, and the inconsistencies in used versions makes comparisons futile.

Considering the lengthy nature of the fuzzing process (from a few hours up to several weeks), a benchmark target must also make the best use of the limited resources by collecting diverse information about a fuzzer's performance. This is possible by injecting the target with a number of bugs large enough to collect metrics along multiple dimensions, without sacrificing the fuzzer's performance. Thus, the cost of running the benchmark is another important concern.

Finally, *ground truth* is the availability of information that allows accurate mapping of observed effects back to their root cause. For crashes and other faults, access to ground truth allows the identification of the specific bug that caused the observed fault. This is one of the key reasons behind LAVA-M's popularity: whenever an injected bug is triggered, an identifying message is printed to `stdout`. This allows crashes to be de-duplicated; multiple crashes can be mapped back to a single bug, and the fuzzer's performance can be measured by the number of bugs it manages to discover.

Based on the previous observations, we distill the following requirements for an effective fuzzing benchmark:

- Few targets with many bugs;
- Real targets with real bugs;
- Diverse targets to test different exploration aspects; and
- Easy access to ground truth.

Based on these observations, we design Magma, a ground-truth fuzzer benchmark suite based on real programs with real bugs. Magma is a set of real-world open-source libraries and applications. For each target, we manually collect bug reports and inspect the fix commits. We then re-insert defective code snippets into the latest version of the code base, along with bug oracles to detect and report if the bugs are reached or triggered. Here, we present the rationale behind Magma, the challenges encountered during its creation, implementation details, and some preliminary results that show the effectiveness of Magma as a fuzzing benchmark.

In summary, this paper makes the following contributions:

- Derivation of a set of desired benchmark properties that should exist when evaluating binary fuzzers;
- Design of Magma, a ground-truth binary fuzzing benchmark suite based on real programs with real instrumented bugs that satisfy the benchmark properties above;
- Development of an open-source corpus of target programs to be used in future fuzzer evaluations;
- A proof-of-concept evaluation of our proposed benchmark suite against six well-known binary fuzzers.

2 BACKGROUND

While in an ideal world, we would have precise specification and test cases for each project, in reality, the number of tests is small and the specification lacking. Testing is thus the process of empirically evaluating code to find flaws in the implementation.

2.1 Fuzz Testing

Fuzz testing is the process by which a target program is evaluated for flaws by feeding it automatically-generated inputs and *concretely* running it on the host system, with the intention of triggering a fault. The process of input generation often relies both on input structure and on program structure. Grammar-based fuzzers (e.g., Superion [50], Peachfuzz [32], and QuickFuzz [18]) leverage the specified structure of the input to intelligently craft (parts of) the input, based on data width and type, and on the relationships between the different input fields. Mutational fuzzers (e.g., AFL [55], Angora [10], and MemFuzz [11]) leverage pre-programmed mutation techniques to iteratively modify the input. Orthogonal to the awareness of input structure, white-box fuzzing [14, 15, 37] leverages program analysis to infer knowledge about the program structure. Black-box fuzzing [3, 52] blindly generates inputs and looks for crashes. Grey-box fuzzing [10, 27, 55] leverages program instrumentation instead of program analysis to collect runtime information. Knowledge of the program structure enables guided input generation in a manner more likely to trigger a crash.

A fuzzing campaign is an instance of the fuzz testing process on a target, based on a specified set of configuration parameters, such as seed files, timeouts, and target programs. Fuzzing campaigns output a set of *interesting* input files that trigger a crash or some other security-critical or undesirable behavior (e.g., resource exhaustion).

Fuzzers are thus programs that perform fuzz testing through fuzzing campaigns. The process of fuzz testing is not constrained by the provided input configurations, such as seed files and target programs. Some proposed fuzzing methods pre-process the provided seeds and select or generate a set of seeds with which to bootstrap the fuzzing process [40, 49]. Other methods analyze the provided target programs and modify them or generate new programs to use as execution drivers for fuzzing the target codebase [23, 33]. Hence, to maintain generality, a fuzzing benchmark must not assume a fixed set of seed files, target programs, or other possible configurations, but should only provide those as starting points when evaluating a fuzzer.

When designing a fuzzing benchmark, it is essential to consider different types of fuzzers and their use cases, in order to clearly define the scope of the benchmark and to permit fair comparisons between equivalent classes of fuzzers.

2.2 Fuzzer Evaluation

The rapid emergence of new and improved fuzzing techniques means that fuzzers are constantly compared against one another, in order to empirically show that the latest fuzzer supersedes previous state-of-the-art fuzzers. Unfortunately, these evaluations have so far been ad hoc and haphazard. For example, Klees et al. found that, of the 32 fuzzer publications that they examined, *all* lacked some important aspect in regards to the evidence presented to support their claims of fuzzer improvement [25]. Notably, their study highlights a set of criteria which should be uniformly adopted across all evaluations. This set of criteria includes: the number of trials (campaigns), seed selection, trial duration (timeout), performance metrics (coverage, crashes, or others), and target programs. This study further demonstrates the need for a ground-truth fuzzing benchmark suite. Such a suite serves as a solution both for replacing heuristic performance metrics and for using a unified set of targets.

2.3 Crashes as a Performance Metric

Most, if not all, state-of-the-art fuzzers implement fault detection as a simple crash listener. If the target program crashes, the operating system informs the fuzzer. A crash, on Linux systems, is a program termination due to some instruction-level security violation. Divisions by zero and unmapped or unprivileged page accesses are two examples of architectural security violations.

However, not every bug manifests as an architectural violation: out-of-bound memory accesses within mapped and accessible page boundaries, use-after-free, data races, and resource starvation may not necessarily cause a crash. Fuzzers commonly leverage *sanitizers* to increase the likelihood of detecting such faults. A sanitizer instruments the target program such that additional run-time checks are performed (e.g., object bounds, type, validity, ownership) to ensure that detected faults are reported, similarly terminating the program with a non-zero exit code. Due to its simplicity, crash count is the most widely-used performance metric for comparing fuzzers. However, it has been shown to yield highly-inflated results, even when combined with de-duplication methods (e.g., coverage profiles and stack hashes) [25].

A high-level approach to evaluating two fuzzers is to compare the number of bugs found by each: if fuzzer *A* finds more bugs than fuzzer *B*, then *A* is superior to *B*. Unfortunately, there is no single formal definition for a bug. Defining a bug in its proper context is best achieved by formally modeling program behavior. However, modeling programs is a difficult and time-consuming task. As such, bug detection techniques tend to create a blacklist of faulty behavior, mislabeling or overlooking some classes of bugs in the process. This often leads to incomplete detection of bugs and root-cause misidentification, which results in a duplication of crashes and an inflation in the performance metrics.

3 DESIRED BENCHMARK PROPERTIES

A benchmark suite is an all-in-one framework which allows for a representative and fair evaluation of a system under test. It outputs normalized performance metrics which allow for a comparison between different systems. To achieve fairness and representativeness, a benchmark needs to be compatible with all systems under test and diverse enough to test the features of all systems equally, without showing bias to one particular feature. A benchmark should also define its scope, which comprises the set of systems the benchmark is designed to test.

In the context of fuzz testing, a benchmark should satisfy the following properties:

Diversity (P1) The benchmark has a wide variety of bugs and programs to mimic real software testing scenarios.

Accuracy (P2) The benchmark yields consistent metrics that accurately evaluate the fuzzer's performance.

Usability (P3) The benchmark is accessible and has no significant barriers for adoption.

3.1 Diversity (P1)

A fuzzing benchmark is a set of targets that are representative of a larger population. Fuzzers are tools for finding bugs in real programs; therefore the goal of a benchmark is to test a fuzzer against features most often encountered in real programs. To this end, a benchmark must include a diverse set of bugs *and* programs. This implies a diversity of bugs with respect to:

Type spatial and temporal memory safety, type confusion, arithmetic, resource starvation, concurrency;

Distribution "depth", fan-in (number of paths which execute the bug), spread (ratio of buggy/total path count);

Complexity number of input bytes involved in triggering a bug, range of input values which triggers the bug, and transformations performed on the input.

It also implies the diversity of programs with respect to:

Application domains file and media processing, network protocols, document parsing, cryptography primitives, or data encoding;

Operations checksum calculations, parsing, indirection, transformation, state management, or validation;

Input structure binary/text, data grammars, or data size.

Satisfying this diversity property requires having injected bugs that closely represent real bugs encountered in in-the-wild production environments. However, *real programs are the only source of real bugs*. Therefore, a benchmark designed to evaluate fuzzers should include such programs with a large variety of real bugs, thus ensuring diversity and avoiding bias. Whereas finding real bugs is desirable, performance metrics based on an unknown set of bugs (with an unknown distribution) makes it impossible to compare fuzzers as we neither know how many bugs there are nor how "easy" they are to trigger. Instead, performance should be measured on a known set of bugs for which ground truth is available.

3.2 Accuracy (P2)

Existing ground-truth benchmarks do not provide a straightforward mapping between crashes and their root cause. For example, the CGC sample set provides crashing test cases for all included bugs, but it does not provide a method for de-duplicating crashes to map them back to unique bugs. Similarly, the Google FTS provides crashing test cases where possible, in addition to a script to triage crashes and map them back to unique bugs. This is achieved by parsing the sanitizer's crash report or by specifying a line of code at which to terminate the program. However, this approach is limited and does not allow for the detection of complex bugs—e.g., where simply executing a line of code is not sufficient to trigger the bug. In contrast, LAVA-M specifies which bug triggers a crash, but the fuzzer can only access this information by piping `stdout` to its own buffers. Unless a fuzzer is tailored to collect LAVA-M metrics, it cannot perform real-time monitoring of its progress. Thus, collection of metrics must be done by replaying the test cases in a post-processing step. Google's FuzzBench relies solely on coverage profiles (rather than fault-based metrics) to evaluate and compare fuzzers. FuzzBench dismisses the need for ground truth, which we believe sacrificies the significance of the results: more coverage does not necessarily imply higher effectiveness.

Crash count, the current widely-used performance metric, suffers from high variability, double-counting, and inconsistent results across multiple trials. The ultimate performance metric for a fuzzer is a measure of unique bugs found. Such a metric enables accurate evaluation of the fuzzer's performance and allows for meaningful comparisons among fuzzers. Thus, leveraging ground truth allows for accurate mapping between reported crashes and unique bugs.

To this effect, a fuzzing benchmark should provide *easy access to ground truth metrics* describing how many bugs the fuzzer can reach, trigger, and detect.

3.3 Usability (P3)

Fuzzers have evolved from simple black-box random-input generation to full-fledged data- and control-flow analysis tools. Each fuzzer might introduce its own instrumentation into a target binary (e.g., AFL [55]), launch the program in a specific execution engine (e.g., QSYM [54], Driller [46]), or feed its input through a specific channel (e.g., libFuzzer [27]). Fuzzers come in many different forms, so a benchmark suite designed to evaluate them must have a *portable* implementation that does not exclude some class of fuzzers within the scope of the benchmark. Running a benchmark should also be a manageable and straightforward process: it should not require constant user intervention, and it should finish within a reasonable time frame. The inherent randomness of most fuzzers increases campaign times, as multiple trials are required to get statistically-meaningful results.

Some targets within existing ground-truth benchmarks—e.g., CGC [9] and Google Fuzzer Test Suite (FTS) [17]—contain multiple vulnerabilities, so it is not sufficient to only run the fuzzer until the first crash is encountered. Additionally, these benchmarks do not specify a method by which crashes are mapped to bugs. Therefore, the fuzzer must run until all bugs are triggered. The imperfections of de-duplication techniques mean that it is often not enough to simply match the number of crashes to the number of bugs. Thus, additional time must be spent triaging crashes to obtain ground truth about bugs. Finally, an ideal benchmark suite should also include a reporting component which collects and aggregates test results to present to the user.

These requirements make the benchmark *usable* by fuzzer developers without introducing insurmountable or impractical barriers. To achieve this property, a benchmark should thus provide *a small set of targets with a large number of discoverable bugs*, and it should provide a framework which can measure and report fuzzer progress and performance.

4 MAGMA: APPROACH

To tackle the aforementioned problems, we present Magma, a ground-truth fuzzing benchmark suite that satisfies the previously-discussed benchmark properties. Magma is a collection of targets with widespread use in real-world environments. The initial set of targets has been carefully selected for their diverse computation operations and the variety of security-critical bugs that have been reported throughout their lifetime.

The idea behind Magma is to publish a ground-truth corpus of targets for uniform fuzzer evaluations. By porting bugs from previous revisions of real programs to the latest version—hereafter referred to as *forward-porting*—we satisfy the bug *diversity and complexity* property (**P1**). Additionally, by inserting minimal instrumentation at bug sites, we establish *ground-truth knowledge of bugs*—allowing for reproducible evaluations and comparisons—thus satisfying the *accuracy* property (**P2**). Finally, by *reducing the steps and requirements to run the benchmark* we fulfill the *usability* property (**P3**) that permits fuzzer developers to seamlessly integrate the benchmark into their development cycle.

4.1 Workflow

For each target, we manually inspect bug and vulnerability reports (CVEs) to assess the type of bug, the proof-of-vulnerability (PoV), and the corresponding fix patches. Following this, we re-introduce the bug into the latest version of the code through forward-porting. We insert minimal source-code instrumentation—coined "canary"—to collect statistics about a fuzzer's ability to reach and trigger the bug. A bug is *reached* when the faulty line of code is executed, and *triggered* when the fault condition is satisfied. Finally, Magma also ships with a *monitoring utility* which runs in parallel with the fuzzer to collect real-time statistics.

We then launch fuzz campaigns, and the monitoring utility collects information about bugs reached and triggered during the campaign. At the end of a campaign, we evaluate the fuzzer's findings against the benchmark to determine which bugs the fuzzer *detected*: i.e., which bugs the fuzzer could identify as manifesting faulty behavior. Using the number of bugs *reached*, *triggered*, and *detected*, as well as timestamps when each such event occurred for every bug, we obtain a set of invariant metrics that we use to evaluate fuzzers and enable accurate comparisons among them.

These metrics allow evaluation of the exploration and detection aspects of a fuzzer. The instrumentation can only yield usable information when the fuzzer exercises the instrumented code paths, thus allowing the collection of the *reached* metric. The data-flow generated by the fuzzer then registers the *triggered* metric when it satisfies the bug trigger conditions. Finally, the fuzzer's fault detection techniques flag a bug as a fault or crash, enabling the collection of the *detected* metric in a post-processing step.

Magma provides a *fatal canaries* mode, where, if a canary's condition is satisfied, the program is terminated (similar to LAVA-M). As soon as the condition is satisfied, the canary emits a `SIGSEGV` signal, crashing the process. The fuzzer then saves this "crashing input" in its findings directory for post-processing. *Fatal canaries* are a form of *ideal sanitization*, in which triggering a bug immediately results in a crash, regardless of the nature of the bug. Fatal canaries allow developers to evaluate their fuzzers under ideal sanitization assumptions without incurring extra sanitization overhead. This mode increases the number of executions during an evaluation, reducing the cost of evaluating a fuzzer with Magma.

4.2 Reached, Triggered, and Detected

It is important to distinguish between *reaching* and *triggering* a bug. A *reached* bug refers to a bug whose oracle was called, implying that the executed path reaches the context of the bug, without necessarily triggering a fault. A *triggered* bug, on the other hand, refers to a bug that was reached, and *whose triggering condition was satisfied*, indicating that a fault occurred. Whereas triggering a bug implies that the program has transitioned into a faulty state, the symptoms of the fault may not be directly observable at the oracle injection site. When a bug is triggered, the oracle only indicates that the conditions for a fault have been satisfied, but this does not imply that the fault was encountered or detected by the fuzzer.

Another distinction is the difference between *triggering* and *detecting* a bug. Whereas most security-critical bugs manifest as a low-level security policy violation for which state-of-the-art sanitizers are well-suited—e.g., memory corruption, data races, invalid arithmetic—some classes of bugs are not easily observable. Resource exhaustion bugs are often detected after the fault has manifested, either through a timeout or an out-of-memory indication. Even more obscure are semantic bugs whose malfunctions cannot be observed without some specification or reference. Different fuzzing techniques have been developed to target such evasive bugs, such as SlowFuzz [36] and NEZHA [35]. Such advancements in fuzzer technologies could benefit from an evaluation which accounts for *detection* rate as another dimension for comparison.

4.3 Boolean Bugs

A software bug is a state transition from a valid state of the program to some undefined, but reachable, state [24]. This transition is only taken when the current state of the program and its inputs satisfy some boolean condition. The program state is comprised of variables, memory, and register content, and even the program counter (PC). As such, the transition condition can be evaluated at runtime before the state transition is undertaken, given that the evaluation procedure does not alter the current program state.

Although it is likely infeasible to perform in-line calculation of the condition without altering the PC, software bugs are rarely, if ever, dependent on individual values of the PC, but rather on basic blocks or lines-of-code. This assumption allows us to insert bug canaries at the source-code level, within the context of the original bug. Based on information obtained from the bug fix, we formulate a boolean expression representing the condition for the undefined transition and evaluate it at runtime as an oracle for identifying triggered bugs.

In practice, the components of the boolean expressions injected are often program variables readily available within the scope of the code block where the bug exists. In rare cases, we insert program variables to simplify the evaluation of the expression and to avoid performing redundant calculations.

4.4 Forward-Porting

In contrast to the process of back-porting fixes from newer software versions to previous releases, we coin the term *forward-porting* for re-purposing bugs from previous releases and injecting them into later versions of the code.

Historical fuzzing benchmarks, like the Google FTS and BugBench, rely on fixed versions of public code bases with known and identified bugs. However, this approach restricts the benchmark to those previous versions with limited sets of bugs. The forward-porting approach allows the benchmark to use a code base with its entire history of bugs as a target for evaluating fuzzers.

An alternative approach to forward-porting bugs would be back-porting canaries. Given a library *libfoo* with a previous release *A* and the latest stable release *B*, the history of bugs fixed from *A* to *B* can be used to identify the bugs present in *A*, formulate oracles, and inject canaries in an old version of *libfoo*. However, when we use *A* as the code base for our target, we could potentially miss some bugs in the back-porting process. This increases the possibility that the instrumented version of *A* has bugs for which no ground-truth is being collected. In contrast, when we follow the forward-porting approach, *B* is used as a code base, ensuring that all *known* bugs are fixed, and the bugs we re-introduce will have ground-truth oracles. It is still possible that with the new fixes and features added to *B*, more bugs could have been re-introduced, but the forward-porting approach allows the benchmark to constantly evolve with each published bug fix.

One key difference between these two processes is that, while back-porting builds on top of code that exists both in the latest and previous versions, forward-porting makes no such assumption. Future code changes that follow a bug fix could be drastic enough to make the bug obsolete or its trigger conditions unsatisfiable. Without verification, forward-porting is thus prone to injecting bugs which cannot be triggered. To minimize the cost of manually verifying injected bugs, we leverage fuzzing to generate PoVs by evaluating them against Magma. Whenever a PoV for a bug is found, it serves as a witness that the bug is triggerable and we add the bug to the list of verified bugs which participate in evaluating the performance of other fuzzers. Note that adding triggerable bugs to the benchmark skews the benchmark towards bugs that can be found through fuzz testing. This is OK because any newly discovered PoV will update the benchmark, balancing the distribution fairly.

To forward-port a bug, we first identify, from the reported bug-fix, which code changes we must revert to re-introduce the flaw. Bug-fix commits can contain multiple fixes to one or more bugs, so it is necessary to disambiguate so as not to inadvertently introduce un-intended bugs into the code. Other bug-fixes may also be spread over multiple commits, as the original fix might not have covered all edge cases. Manual effort remains an important aspect of the forward-porting process and we currently rely on it for the injection of all bugs in Magma. In a following step, we identify what program state is involved in evaluating the trigger condition, and we introduce additional program variables to access that state, if necessary. Finally, we determine the condition expression which evaluates the program state and the constraints on it to serve as an oracle for identifying a triggered bug. We then identify a point in the program where we inject a canary before the bug can manifest faulty behavior.

4.5 Benchmark-Tuned Fuzzers

Since Magma is not a dynamically-generated benchmark, there is a risk of fuzzers over-fitting to its injected bugs. This issue exists among other performance benchmarks, as the developers of the systems-under-test tweak their systems to fare better on a specific benchmark instead of real workloads. One way to counter this issue is to dynamically synthesize the benchmark and ensure the evaluated system is not aware of the synthesis parameters. This comes at the risk of generating workloads different from real-world scenarios, rendering the evaluation biased or incomplete. While program generation and synthesis is a long-studied topic [6, 19, 23], it remains hard to scale and the generated programs are often not faithful to real development styles and patterns. An alternative is to provide a static benchmark that resembles real workloads, and continuously update it and introduce new features to evaluate different aspect of the system. This is the approach followed by Magma, as well as benchmarks in other domains, such as the widely-used SPEC benchmark suite [45]. The underlying assumption of these benchmarks is that if the evaluated system performs well on the sample workloads provided, then it will perform similarly well on real workloads, as the benchmark is designed to be representative of such settings.

Magma's forward-porting process also allows the targets to be updated. For every injected bug, we maintain only the code changes needed to re-introduce the faulty code and in-line instrumentation. It is then possible to apply updates to the target, which could introduce new bugs or features. This allows us to update Magma as its targets change and evolve. In case the latest code changes conflict with an injected bug, the following course of action would be to either ignore those changes, or to remove the conflicting bug, as it would become obsolete with later releases of the target.

These measures ensure that Magma remains representative of real complex targets and suitable for evaluating fuzzers.

5 DESIGN DECISIONS

To maintain the portability and usability of the benchmark, as well as the integrity and significance of collected performance metrics, we must address the following challenges.

5.1 Bug Chain

Consider the contrived example in Listing 1. When `tmp.len == 0`, the condition for bug 17 on line 10 is not satisfied, and bug 9 is captured on line 13 and triggered on line 14, resulting in a divide-by-zero crash. When `tmp.len > 16`, bug 17 is captured on line 9 and triggered on line 10; `tmp.len` is overwritten in the struct by a non-zero value, and bug 9 is not triggered. However, when `tmp.len == 16`, bug 17 is triggered, overwriting `tmp.len` in the struct with the null terminator and setting its value to 0 (in the case of a Little-Endian system). This leads to triggering bug 9, despite the test case not explicitly specifying a zero-length `str`.

When a fuzzer generates an input that triggers a bug that goes undetected, and execution continues past the triggered bug, the program transitions into an undefined state. Any information collected about program state from this point forward is not reliable.

```
 1 void libfoo_baz(char *str) {
 2     struct {
 3         char buf[16];
 4         size_t len;
 5     } tmp;
 6     tmp.len = strlen(str);
 7
 8     // possible OOB write in strcpy()
 9     magma_log(17, tmp.len >= sizeof(tmp.buf));
10     strcpy(tmp.buf, str);
11
12     // Possible div-by-zero if tmp.len == 0
13     magma_log(9, tmp.len == 0);
14     int repeat = 64 / tmp.len;
15     int padlen = 64 % tmp.len;
16 }
```

Listing 1: The bug chain problem can result in execution traces which do not exist in the context of normal program behavior.

To address this issue, we allow the fuzzer to run the entire execution trace, but only collect bug oracle statistics *before and until* the first bug is triggered. Oracles are not meant to signify that a bug has been executed; they only indicate whether the conditions to execute the bug have been satisfied.

5.2 Leaky Oracles

The introduction of oracles into the targets might interfere with a fuzzer's exploration aspect and allow it to over-fit. For example, if oracles were implemented as if-statements, fuzzers that maximize branch coverage could detect the oracle's branch and find an input to execute it.

Splitting the targets into instrumented and un-instrumented binaries is one possible solution to leaking oracle information. The fuzzer's input would be fed into both binaries, but the fuzzer would only collect instrumentation it needs from the un-instrumented (with respect to bug canaries) binary. The instrumented (canary) binaries collect data from the oracles and report it to an external crash monitoring utility. This approach, however, introduces other challenges in regards to duplicating the execution trace between two binaries (e.g., replicating the environment, piping standard I/O streams, passing identical launch parameters, and maintaining synchronization between executions), which can greatly complicate the implementation and introduce runtime overheads.

Alternatively, we employ always-evaluate memory writes, e.g., `bugs[b] = (int)condition`. Although this approach may introduce memory access patterns that could be detected by taint tracking or other data-flow analysis techniques, statistical tests can be used to infer whether the fuzzer over-fits to the canaries. By running campaigns again with un-instrumented binaries, we can verify if the results vary significantly. In Section 6.2, we explore the state-of-the-art in data-flow-oriented fuzzers and discuss why over-fitting is currently a non-issue.

5.3 Proofs of Vulnerability

In order to increase confidence in the injected bugs, a proof-of-vulnerability (PoV) input must be supplied for every bug, to verify that it can be triggered. The process of manually crafting PoVs, however, is arduous and requires domain-specific knowledge, both about the input format and the program or library, potentially bringing the bug-injection process to a grinding halt.

To tackle this issue, we inject bugs into the targets without first supplying PoVs, then we launch multiple fuzz campaigns against the targets and collect inputs that trigger each bug. Bugs which are not triggered, even after multiple trials, are then manually inspected to verify path reachability and satisfiability of trigger conditions. When available, we also extract PoVs from public bug reports.

5.4 Unknown Bugs

Using real-world programs, it is highly likely that bugs exist for which no oracles have been added. A fuzzer might trigger one of these bugs and detect a fault. Due to the imperfections in de-duplication techniques, these crashes are not counted as a definite performance metric, but are instead used to improve the benchmark itself. By manually studying the execution trace, it is possible to determine the root cause and add the new bug to the benchmark.

5.5 Compatibility

Fuzzers are not limited to a specific execution engine under which to analyze and explore the target. Some fuzzers (e.g., Driller [46] and T-Fuzz [33]) leverage symbolic execution (using an engine such as angr [43]) to explore new paths. This could introduce inconsistencies in the environment—depending on how the symbolic execution engine models it—and incompatibilities with the benchmark's instrumentation.

However, the defining trait of fuzzers, in contrast with other types of bug-finding tools, is the fact that they execute the target concretely on the host system. Unlike the CGC which was launched to evaluate all bug-finding tools, Magma is a *fuzzing* benchmark, and is primarily suited for evaluating fuzzers. This includes white-box fuzzers which perform symbolic execution to guide input generation, as long as the target is finally *executed* on the host system, to allow for collecting instrumentation results. We make the following assumption about the tools evaluated with Magma: fuzzers must execute the target programs in the context of an OS process, with unrestricted access to OS's facilities (system calls, dynamic libraries, file system). This allows the accompanying monitor to extract canary statistics using the operating system's services at relatively low overhead and low complexity.

6 IMPLEMENTATION

6.1 Bug Injection

In the context of open-source projects, we approximate a bug by the lines of code affected by a fix in response to a CVE or bug report, and a boolean expression representing its triggering condition. The fix commit(s) often patch the code at the first point in the program flow where enough information is available to determine if the input is invalid. For every such bug fix, we revert the commit and insert an oracle that: (1) reports when that line of code is reached, and (2) reports when the input satisfies the conditions for faulty behavior (triggers the bug). Unlike LAVA-M, bug trigger conditions need not necessarily be a check for a magic value in the input: they are often checks on state variables and transformations of (parts of) the input, which represents the complexity of real bugs. To demonstrate such complexity, Listing 2 highlights a bug in libpng where the `row_factor` variable—calculated from input values—is used as the denominator in an integer division, leading to a divide-by-zero crash under the right conditions. Triggering this bug requires very specific values for the width, channels, bit depth, and interlaced input fields to yield a zero row factor through integer overflow.

```c
void png_check_chunk_length(ptr_t png_ptr, uint32_t
     length) {
    size_t row_factor =
          png_ptr->width       // uint32_t
          * png_ptr->channels // uint32_t
          * (png_ptr->bit_depth > 8? 2: 1)
          + 1
          + (png_ptr->interlaced? 6: 0);

    if (png_ptr->height > UINT_32_MAX/row_factor)
          idat_limit = UINT_31_MAX;
}
```

Listing 2: Divide-by-zero bug in libpng. The input bytes undergo non-trivial transformations to trigger the bug.

```c
1  void libfoo_bar() {
2    // uint32_t a, b, c;
3    magma_log(42, (a == 0) | (b == 0));
4    // possible divide-by-zero
5    uint32_t x = c / (a * b);
6  }
7  void magma_log(int id, bool condition) {
8    extern struct magma_bug *bugs; // = mmap(...)
9    extern bool faulty; // = false initially
10   bugs[id].reached   += 1         & (faulty ^ 1);
11   bugs[id].triggered += condition & (faulty ^ 1);
12   faulty = faulty | condition;
13 }
```

Listing 3: Magma bug instrumentation.

6.2 Instrumentation

For every injected bug, we add an oracle before the defective line of code leading to undefined or unexpected behavior. To simplify the implementation of the benchmark and make it less restrictive, we inject the instrumentation into the same binaries that are run by the fuzzer. An injected oracle first evaluates a boolean expression representing the bug's triggering condition: this often involves a binary comparison operator. Compilers (e.g., gcc, clang) generally translate a value-taken binary comparison operator as a `cmp` followed by a `set` instruction, embedding the calculation directly into the execution path. The canary then writes the result of the evaluation at a fixed location in memory, where Magma's instrumentation is maintained. While this does increase the odds that data-flow or memory access patterns are recognizable, this is unlikely given the current state of the art. Notably, MemFuzz [11] ignores memory accesses where the *address* does not depend on program input. Orthogonally, VUzzer [39] instruments `cmp` instructions to determine, through dynamic taint tracking, which input bytes are compared to which values. While this may give VUzzer an unfair insight into the canaries, it does not yield any results since the bugs' trigger conditions are more complex than a simple one-to-one comparison between an input byte and an immediate value. Statistical significance tests can still be performed, by comparing performance against instrumented and un-instrumented binaries, to determine if a fuzzer has indeed over-fit to the canaries.

Listing 3 shows the canary's implementation and how an oracle is injected at a bug's location. We implement the canaries as always-evaluated memory accesses, as in lines 10 and 11, to reduce the chance that coverage-based exploration triggers bugs by solving constraints on branch conditions or prioritizing undiscovered branches. The `faulty` flag, on line 12, is set to 1 whenever the first bug is encountered in an execution. This addresses the *bug chain* problem by disabling future canaries to avoid recording corrupted states. In order to extract canary statistics from the running programs, we use a memory-mapped file (line 8). An external monitoring utility then reads that file to extract the data. It is worth noting that, on Unix systems, a file can also be memory-backed (e.g., tmpfs), such that access latency is very low, and the overhead of executing the instrumentation becomes negligible.

Table 1: The targets, driver programs, and bugs incorporated into Magma. The base versions are the latest at the time of writing.

Target	Drivers	Base version	File type	Bugs
libpng	`read_fuzzer`, `readpng`	1.6.38	PNG	7
libtiff	`read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF	14
libxml2	`read_memory_fuzzer`, `xmllint`	2.9.10	XML	19
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF	22
openssl	`asn1parse`, `bignum`, `x509`, `server`, `client`	3.0.0	*Binary Blobs*	21
sqlite3	`sqlite3_fuzz`	3.32.0	SQL Queries	20
php	`json`, `exif`, `unserialize`, `parser`	8.0.0-dev	*Various*	18

To inject an oracle, we insert a call to `magma_log()` at the bug's location, just before the buggy code is executed (line 5). Compound trigger conditions—i.e., those including the logical `and` and `or` operators—often generate implicit branches at compile-time, due to short-circuit compiler behavior. To mitigate the effects of leaking ground-truth information through coverage, we provide custom x86-64 assembly blocks to evaluate the `&&` and `||` operators in a single basic block, without short-circuit behavior. In case the compilation target is not x86-64, we fall back to using C's bitwise operators, `&` and `|`, which are more brittle and susceptible to safety-agnostic compiler passes [44].

6.3 Reporting

Magma also ships with a monitoring utility to collect canary statistics in real time by constantly polling the shared file. This allows the visualization of the fuzzer's progress and its evolution over time, without complicating the instrumentation.

The monitor collects data about reached and triggered canaries, which mainly evaluate the program exploration aspect of the fuzzer. To study the fault detection capabilities of the fuzzer, we inspect its output, in a post-processing step, to identify which bugs it detected. This is easily done by replaying the crashing test cases produced by the fuzzer against the benchmark canaries to determine which bugs were triggered and hence detected. It is also possible that, since the benchmark's target programs are not synthetic, there might be new crashing test cases not corresponding to any injected bug. In such scenarios, the new bug is triaged and added to the benchmark for other fuzzers to find.

7 TARGET SELECTION

Fulfilling the diversity property requires including targets from different application domains, with different input structures, and which perform a variety of operations and transformations. The targets should also have a long history of diverse bugs. To that end, we select the following initial set of targets for inclusion in Magma, summarized in Table 1 and classified in Table 2.

libpng. A PNG image processing library. It presents a significant attack surface due to its widespread use in many applications [41]. CVEDetails.com shows that, among the reported vulnerabilities, denial of service is the most common, followed by buffer overflows, and code execution. Libpng relies on structured input (PNG [21] file format) with several hard checks (ASCII chunk types, checksums, enum field values). It implements parser mechanics and

Table 2: Evaluated features present in Magma. Fuzzers have to overcome these challenges in real-world targets.

	Magic Values	Recursive Parsing	Compression	Checksums	Global State
libpng	✓	✗	✓	✓	✗
libtiff	✓	✗	✓	✗	✗
libxml2	✓	✓	✗	✗	✗
poppler	✓	✓	✓	✓	✗
openssl	✓	✗	✓	✓	✓
sqlite3	✓	✓	✗	✗	✓
php	✓	✓	✗	✗	✗

(de)compression functions, which are common operations often prone to bugs.

libtiff. A TIFF image processing library. Not unlike libpng, libtiff is extensively relied on by end-user applications, e.g., media viewers [51]. The TIFF [1] file format does not include checksums or ASCII tag types, but it introduces indirection (offsets into the file itself) and linked lists of directories. These additions further complicate the implementation of libtiff's parsers and state machines and give rise to more complex bugs.

libxml2. An XML document parsing library. XML [53] is a highly-structured and human-readable mark-up language. It is used even more extensively than the first two libraries, mainly as a text-based data exchange format, and it introduces a very different type of structured input. CVEDetails.com displays a vulnerability type distribution for libxml2 similar to the other libraries, but the verbose human-readable XML format presents new challenges for fuzzers to generate inputs that could pass any syntax validation checks.

poppler. A PDF document parsing library. PDF [20] is a widely-used document file format, often with in-browser viewing support. It is also a very highly-structured file format, which makes it a good target for fuzzers. Its vast exposure and large attack surface makes it an ideal target to test against due to its high security impact.

openssl. An SSL/TLS network protocol handler. OpenSSL [31] introduces new forms of operations, such as cryptographic transformations, big number arithmetic, and state management, to name a few. These operations are well-suited for evaluation against realistic workloads, as they represent a large population of real applications.

sqlite. A SQL database engine. SQL [22] is a powerful language for querying and managing relational databases. The addition of *sqlite* into Magma enhances the benchmark with more complex operations involving the parsing and processing of query statements, as well as maintaining an in-memory representation of a database. Custom memory allocators, various data structures, selectors and filters, and regular expressions are features and components comprising the *sqlite* code base, to name a few.

php. A general-purpose scripting language. PHP [47] is a scripting language highly-suited for web development and used by over 75% of public websites [48]. The preprocessor engine incorporates a parser and intermediate compiler for the PHP code, as well as an interpreter for the generated byte-code. Programming language processors present new exploration fields for fuzzers due to the intricacies of the involved language grammars, making them a suitable target for characterizing finer-grain fuzzer capabilities.

7.1 Injected Bugs

Magma's forward-porting process relies mostly on human effort. While we strive to inject as many bugs in the included targets as possible, resources remain limited until the mainstream adoption of Magma allows contributions from the community. We list the 121 injected bugs in Table A1.

We do not prioritize one type of bug over another. Instead, we prioritize more recent bugs, since the code changes are less likely to be drastic, and the bug more likely to be port-able. Note that a shot-gun approach at bug injection is not wise: with too many bugs injected, the fuzzer may spend most of its time replaying crashes and resetting state instead of exploring more of the program. Additionally, the more injected bugs there are, the higher the chance of interaction between them, which could render some bug conditions unsatisfiable due to the *bug chain* problem.

7.2 Drivers and Seeds

Whereas a *target* represents the code base into which we inject bugs and canaries, a *driver* is the executable program used for evaluating a fuzzer by providing an accessible interface into the target. FuzzGen [23] and T-Fuzz [33] are two fuzzers that exercise custom driver programs—either by generating new ones or modifying existing ones—to achieve more coverage. As more exotic fuzzers emerge, it is important to consider target programs as another configuration parameter to fuzz campaigns. Seed files are another such example, where certain fuzzing techniques opt to generate new seeds [49] or modify existing seeds [40] to improve fuzzer performance.

As part of Magma, we include driver programs and seed sets provided by the developers of the fuzz targets. Developers have the required domain knowledge to write driver programs representative of real applications that use these targets and exercise high coverage. This intuition is inspired by Google's OSS-Fuzz project [2] which includes a large set of real targets and execution drivers written by experts in their domains.

8 EXPERIMENTS AND RESULTS

8.1 Methodology

In order to establish the versatility of our metrics and benchmark suite, we evaluated several fuzzers. We chose a set of six *mutational grey-box fuzzer* fuzzers whose source code was available, namely: AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, and honggfuzz.

For each fuzzer/target combination, we launched ten identical fuzzing campaigns, using seed files included in the Magma source package, and monitored their progress for 24 hours. In total, this amounted to 26,000 CPU-hours of fuzzing. We configured the Magma monitoring utility to poll canary information every five seconds.

Since all evaluated fuzzers implement fault detection as a simple crash or hang listener, it is possible to leverage the *fatal canaries* mode to evaluate the two fuzzer aspects individually: (1) program exploration; and (2) fault detection. To determine which bugs are *reached* and *triggered*, we compiled and fuzzed the Magma binaries with *fatal canaries*.

To determine which bugs are *detected* by the fuzzers' crash listeners, we recompiled the benchmark binaries with sanitization to produce hardened binaries. We then execute the hardened binaries with the crashing test cases reported by the fuzzers in the first phase; if the target hangs or exceeds its memory limit, or if the sanitizers report a crash, then the fuzzer can detect it. The default timeout for a hanging input is *50 milliseconds*, and the default memory limit is *50 MiB*—the same as AFL's defaults. We used *AddressSanitizer* [42] for fault detection. While sanitization incurs a performance overhead that affects execution speeds and timeouts, and thus the fuzzer's ability to detect some faults, we choose the minimum default thresholds. This ensures that, in the worst case, the evaluation does not undermine the fuzzer's capabilities. Instead, it assumes the fuzzer is configured with strict fault detection parameters and can thus flag more bugs as detected. Although this may introduce false-positive detections in the post-processing step—i.e., we flag a bug as detected, whereas in fact the fuzzer might not have detected it—it skews results in favor of the evaluated fuzzers as they detect more bugs. On one hand, this method of evaluation measures the capabilities of sanitizers to detect violations. On the other hand, it highlights what bugs fuzzers can realistically detect when fuzzing without ground truth.

Using fatal canaries to evaluate *reached* and *triggered* bugs is only applicable to fuzzers whose fault detection relies solely on hangs or crashes; if a fuzzer implements a different method for identifying erroneous states (e.g., semantic bug-finding tools *à la* NEZHA [35]), then their method must be used when evaluating the fuzzer's fault detection capability. It is important not to introduce bias into the evaluation: if a fuzzer like NEZHA is compared against one like AFL, then AFL-fuzzed targets must not be compiled with *fatal canaries*. This is because fatal canaries stop the target's execution prematurely upon finding a bug, thus potentially providing AFL with an unfair performance advantage. Benchmark parameters must be identical for all evaluated fuzzers to ensure fairness.

In our evaluation of the six fuzzers, all experiments were run on three machines, each with an Intel® Xeon® Gold 5218 CPU and 64 GB of RAM, running Ubuntu 18.04 LTS 64-bit. The targets were compiled for x86-64.

8.2 Expected Time-to-Bug

To measure fuzzer performance, we record the time it needs to *reach* and to *trigger* a bug, across ten identical runs. The intuition behind this metric is that, in practice, compute resources are limited, and fuzz campaigns are consequently run for a specific duration before being reset (due to diminishing returns). As such, the faster a fuzzer can find a bug, the better its performance.

Due to the highly-stochastic nature of fuzzing, performance can vary widely between identical runs. It is even possible that a fuzzer occasionally fails to find some bugs within the alloted time, leading to missing measurements. Therefore, it is invalid to measure the fuzzer's performance based only on its recorded measurements. Instead, missing measurements must factor into the fuzzer's performance with respect to a particular bug.

Consider a fuzzer whose performance is evaluated against a target for finding a bug. The fuzz campaign runs for a fixed duration, T. The campaign is repeated N times, to account for the effects of randomness. In M out of the N times, the fuzzer manages to find

Table 3: The mean number of bugs found by each fuzzer across ten campaigns. The standard deviation is also given. The best performing fuzzer(s) (per target) is highlighted in green.

Target	honggfuzz	afl	moptafl	aflfast	afl++	fairfuzz
libpng	3.6 ± 0.52	1.6 ± 0.70	1.0 ± 0.00	1.3 ± 0.48	1.2 ± 0.42	1.5 ± 0.53
libtiff	4.7 ± 0.67	4.8 ± 0.42	4.4 ± 0.52	4.2 ± 0.42	3.4 ± 0.52	4.8 ± 1.23
libxml2	5.0 ± 0.47	2.9 ± 0.32	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	2.6 ± 0.52
openssl	2.8 ± 0.63	2.9 ± 0.57	3.0 ± 0.00	2.6 ± 0.52	3.0 ± 0.67	2.0 ± 0.47
php	2.8 ± 0.42	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	1.6 ± 0.70
poppler	3.8 ± 0.92	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	3.0 ± 0.00	2.9 ± 0.32
sqlite3	3.2 ± 0.63	0.8 ± 0.92	1.2 ± 1.14	1.2 ± 0.79	1.6 ± 0.84	1.5 ± 0.53

the bug and produces a time-to-bug measure, t_i, where $1 \le i \le M$. In the remaining $N - M$ runs, no measurement is recorded.

We model the time-to-bug as a random variable X with an Exponential distribution, $Exp(\lambda)$, where $\lambda > 0$ is the *rate parameter* of the distribution. The expected value $E(X) = \frac{1}{\lambda}$ specifies the expected time it takes the fuzzer to find the bug. The exponential distribution is typically used to measure waiting time between independent events in a Poisson point process. It has the *memoryless* property, which states that the waiting time until an event does not depend on how much time had already elapsed. This model fits the behavior of black-box fuzzers, where fuzzer performance does not depend on any accumulated state since the start of a campaign. Although coverage-guided fuzzers, including those we evaluated, maintain and build up state throughout each run, we use this model as an over-approximation for expected time-to-bug values.

Let $\{t_i\}_{i=1}^{M}$ be the recorded time-to-bug values when the fuzzer manages to find the bug within the allotted time T. Using the least-squares method, we fit an exponential distribution over the results of each identical set of campaigns by minimizing the cost function:

$$L(\hat{\lambda}) = \sum_{i=1}^{M} (t_i - \frac{1}{\hat{\lambda}})^2 + (N - M)(\frac{T}{\lambda_t} - \frac{1}{\hat{\lambda}})^2, \quad \text{where} \quad \lambda_t = \ln(\frac{N}{N-M})$$

$\frac{\lambda_t}{T}$ is the rate parameter of an exponential distribution whose cumulative density until the cut-off (area under curve for $X \le T$) is equal to $\frac{M}{N}$, which represents the fraction of times the fuzzer managed to find the bug within the duration T.

This allows us to find an approximate of λ, namely $\hat{\lambda}$. We are interested in calculating $E(X)$, the expected time-to-bug, which can be expressed as:

$$E(X) := \frac{1}{\hat{\lambda}} = \frac{M \times \bar{t} + (N - M) \times \frac{T}{\lambda_t}}{N}$$

where $\bar{t}$ is the arithmetic mean of the recorded measurements.

While this measure may not be exact, it provides intuition about the expected fuzzer performance for each bug, and insight into the complexity of different bugs found across all campaigns.

8.3 Analysis of Results

Table 3, Figure 1, and Table 4 present the results of our fuzzing campaigns. We refer to Table A2 and Table A3 for additional results.

Table 4: Expected time-to-trigger-bug for each fuzzer, in *s*econds, *m*inutes, or *h*ours. Shaded cells indicate that the bug was never detected by the fuzzer, while a "–" indicates that the event was never encountered throughout the evaluation. The Aggregate column gives the expected time-to-bug across all campaigns by all fuzzers for that bug. This is used as a metric to sort bugs by "difficulty".

Bug ID	hfuzz	afl	moptafl	aflfast	afl++	fairfuzz	Aggregate
AAH003	11s	10s	10s	10s	10s	10s	10s
AAH037	10s	15s	16s	15s	38s	15s	18s
AAH041	10s	15s	16s	15s	38s	15s	18s
JCH207	1m	2m	2m	1m	1m	1m	1m
AAH056	14m	13m	11m	11m	10m	8m	11m
AAH015	14s	34m	30m	10m	42m	10m	21m
MAE016	10s	3m	4m	4m	5m	2h	25m
AAH032	2m	1h	26m	34m	36m	10h	1h
AAH020	2h	2h	2h	55m	2h	49m	1h
AAH055	2m	25m	4h	2h	1h	4h	1h
MAE008	7h	1m	1m	1m	1m	17h	2h
MAE014	4h	5m	8m	8m	9m	205h	2h
AAH052	14m	3h	2h	9h	2m	13h	4h
AAH022	34m	7h	4h	4h	4h	12h	5h
JCH215	40m	20h	9h	10h	7h	8h	7h
AAH017	205h	19h	12h	9h	88h	5h	13h
JCH201	–	16h	12h	12h	14h	14h	14h
JCH232	1h	86h	48h	22h	30h	8h	17h
MAE115	34h	19h	18h	53h	16h	205h	22h
AAH008	3h	24h	–	49h	89h	20h	29h
AAH014	–	4h	47h	–	–	23h	58h
AAH007	57s	–	–	–	–	–	109h
AAH045	1h	–	–	–	–	–	109h
AAH013	4h	–	–	–	–	–	110h
AAH026	6h	–	–	–	–	–	110h
JCH226	9h	–	–	–	–	–	126h
AAH024	7h	–	–	–	–	–	146h
JCH228	50h	206h	86h	–	87h	–	146h
AAH010	17h	–	–	–	–	88h	146h
AAH001	18h	206h	–	–	–	–	172h
MAE104	–	205h	–	47h	86h	–	205h
AAH016	–	–	–	86h	–	86h	324h
AAH053	35h	–	–	–	–	–	325h
AAH009	–	–	205h	–	87h	–	444h
JCH212	52h	–	–	–	–	–	445h
AAH050	90h	–	–	–	–	–	685h
AAH025	205h	–	–	–	–	–	1404h
AAH048	206h	–	–	–	–	–	1404h

8.3.1 Bug Count and Statistical Significance. Table 3 shows the mean number of bugs found per fuzzer (across ten 24 hour campaigns). These values are susceptible to outliers, limiting the conclusions that we can draw about fuzzer performance. We therefore conducted a statistical significance analysis of the collected sample-set pairs to calculate p-values using the Mann-Whitney U-test. P-values provide a measure of how different a pair of sample sets are, and how significant these differences are. Because our results are collected from independent populations (i.e., different fuzzers), we make no assumptions about their distributions. Hence, we apply the Mann-Whitney U-test to measure statistical significance. Figure 1 shows the results of this analysis.

The test shows that AFL, AFLFast, AFL++, and MOpt-AFL performed similarly against most targets, despite some minor differences in mean bug counts shown in Table 3. The calculated p-values show that, in most cases, those small fluctuations in mean bug

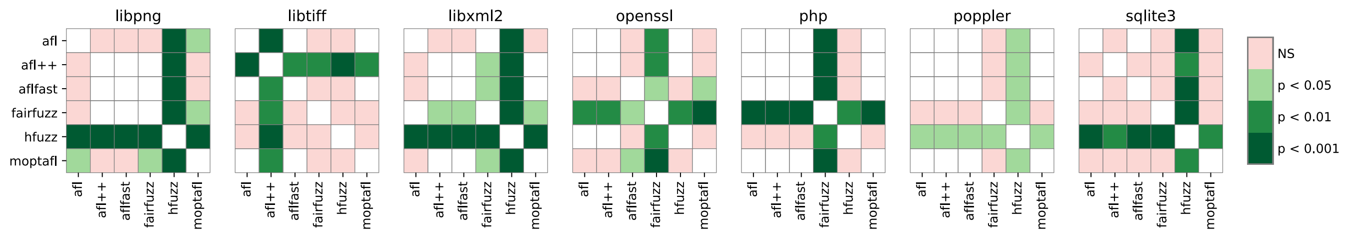

Figure 1: Significance of evaluations of fuzzer pairs using p-values from the Mann-Whitney U-Test. We use $p < 0.05$ as a threshold for significance. Values greater than the threshold are shaded red. Darker shading indicates a lower p-value, or higher statistical significance. White cells indicate that the pair of sample sets are identical.

counts are not significant, and the results are thus not sufficiently conclusive. One oddity is the performance of AFL++ against *libtiff*. Table 3 reveals an overall lower mean bug count for AFL++ compared to all other fuzzers, and Figure 1 shows that this difference is significant. While Table 4 shows that AFL++ triggered five *libtiff* bugs across the ten campaigns, its performance was inconsistent, resulting in a low mean bug count. *We therefore conclude that the performance of all AFL-based fuzzers with default configurations against Magma is, to a large degree, equivalent.*

FairFuzz [26] also displayed significant performance regression against *libxml2*, *openssl*, and *php*. Despite the evaluation of FairFuzz in the original paper showing that it achieved the highest coverage against `xmllint`, that improvement was not reflected in our results.

Finally, honggfuzz performed significantly better than all other fuzzers in four out of seven targets, possibly owing to its wrapping of memory-comparison functions.

8.3.2 *Time-to-bug.*

A summary of individual bugs is presented in Table 4. Of the 46 verified Magma bugs, the evaluated fuzzers triggered 38 in total, and no single fuzzer triggered more than 33 bugs. The bugs in Table 4 are presented in increasing discovery difficulty, where expected time-to-bug ranges from seconds to hundreds of hours. This suggests that the evaluated set of fuzzers still have a long way to go in improving program exploration.

We argue to dismiss time-to-bug measurements as "flukes" of randomness if the ratio $\frac{M}{N}$ is small (e.g., if the bug is only found once across ten campaigns). The expected time-to-bug metric converges towards $T \times N$ for large N. For $N = 10$, this value is 207 hours, which explains the recurrence of 205 and 206 hour values in Table 4; such bugs were only found by chance, not because of the fuzzer's capabilities.

Table 4 highlights a common set of "simple" bugs which all fuzzers find consistently within 24 hours. These bugs serve as a baseline for detecting performance regression: if a new fuzzer fails to discover these bugs, then its policy or implementation should be revisited.

Again, it is clear that honggfuzz outperforms all other fuzzers, finding 11 additional bugs not triggered by other fuzzers. In addition to its finer-grained instrumentation, honggfuzz natively supports persistent fuzzing. Our experiments showed that honggfuzz's execution rate was at least three times higher than that of AFL-based fuzzers using persistent drivers. This undoubtedly contributed to honggfuzz's strong performance.

Table A3 presents the results of evaluating these fuzzers for seven days against a subset of the bugs in Magma, displaying a similar pattern.

8.3.3 *Seed Coverage.*

Our evaluation used seeds provided by the developers of the Magma targets. These seeds may exercise different code paths that intersect with Magma's injected bugs, making it easier for coverage-guided fuzzers to find and trigger these bugs. Although we do not use a specific seed selection policy, we provide the same set of seeds across all campaigns to allow for fair evaluations. This is evident in our results, as all seed-coverage bugs are "reached" by the fuzzers around the same time (see Table A2). Most bugs not included in seed coverage show significantly increasing time-to-bug measurements, which highlight different fuzzer specialties and begin to show performance improvements brought upon by the evaluated fuzzers.

8.3.4 *Achilles' Heel of Mutational Fuzzing.*

Bug AAH001 (CVE-2018-13785, shown in Listing 2), is a divide-by-zero bug in *libpng*. It is triggered when the input is a non-interlaced 8-bit RGB image, whose width is exactly `0x55555555`. This "magic value" is not encoded anywhere in the target, and is easily calculated by solving the constraints for `row_factor == 0`. However, random mutational fuzzing struggles to discover these types of bugs. This is because the fuzzer has a large input space from which to sample from, making it unlikely to pick the exact byte sequence (here, `0x55555555`). This manifests in our results as a high expected time-to-bug: the only fuzzer to trigger this bug was AFL, and it was only able to do so once (in ten trials).

8.3.5 *Magic Value Identification.*

AAH007 is a dangling pointer bug in *libpng*, and illustrates how some fuzzer features improve bug-finding ability. To trigger this bug, it is sufficient for a fuzzer to provide a valid input with an `eXIF` chunk (which is then not marked for release upon object destruction, leading to the dangling pointer). Unlike the AFL-based fuzzers, honggfuzz is able to consistently trigger this bug relatively early in each campaign. We posit that this is due to honggfuzz replacing the `strcmp` function with an instrumented wrapper that incrementally satisfies string magic-value checks.

8.3.6 *Semantic Bug Detection.*

AAH003 (CVE-2015-8472) is a data inconsistency in `libpng`'s API, where two references to the same piece of information (color-map size) can yield different values. Such a semantic bug does not produce observable behavior that violates a known security policy, and it cannot be detected by state-of-the-art sanitizers without a specification of expected behavior. This is evident in our results, as all fuzzers manage to reach this bug very early in each campaign, but it consistently remains undetected.

Semantic bugs are not always be benign. Privilege-escalation and command injection are two of the most security-critical logic bugs that are still found in modern systems, but they remain difficult to detect with standard sanitization techniques. This observation highlights the shortcomings of current fault detection mechanisms and the need for more fault-oriented bug-finding techniques (e.g., NEZHA [35]).

8.4 Discussion

8.4.1 Existing Benchmarks. Magma allows us to make precise measurements for our selected performance metric: time-to-bug. This enables accurate comparisons between fuzzers across several dimensions: bugs *reached*, *triggered*, and *detected*. Previous work on ground-truth benchmarks, namely LAVA-M, yields only a boolean result for each injected bug: triggered or not triggered. Inference of time-to-bug is not straightforward, as it relies on querying the file system for the creation date of the crashing test cases—a feature not necessary supported by all file systems. LAVA-M also provides no measure for bugs reached, and it treats all triggered bugs as crashes. It is thus not suitable to evaluate a fuzzer's fault detection abilities. Google's FuzzBench [16] only measures edges covered as a performance metric, dismissing fault-based metrics such as crash counts or bug counts. Edge coverage, however, is an approximation of control-flow path coverage, and relying on it as the only metric for performance could result in biased evaluations, as our results highlighted for FairFuzz.

8.4.2 Ground Truth and Confidence. Ground truth enables us to map a fault to its root cause and facilitates accurate crash deduplication. Source code canaries and our monitor simplify access to ground truth information. In this sense, Magma provides ground-truth knowledge for the 121 injected bugs. Orthogonally, a bug's PoV is a witness that it can be triggered. If a fuzzer does not trigger a bug with an existing PoV then we can state that this fuzzer *fails to trigger that bug*; we cannot give conclusions for bugs where no PoV exists. Of the 121 bugs, 46 have known PoVs which are included in Magma; only these bugs can be accounted for in measuring a fuzzer's performance with high confidence. Bugs without a PoV may still be useful: any fuzzer evaluated against Magma could find a PoV, increasing the benchmark's utility in the process. Adoption of the benchmark will reduce the number of bugs without witnesses.

8.4.3 Beyond Crashes. Although Magma's instrumentation does not collect information about *detected* bugs—simply because detection is a characteristic function of the fuzzer and not the bug itself—Magma enables the measurement of this metric through a post-processing step, supported by the *fatal canaries* mode where applicable. Bugs are not restricted to crash-triggering faults. Some bugs result in resource starvation (out-of-control loops or mallocs, worst-case execution), privilege escalation, or undesirable outputs. Fuzzer developers acknowledge the need for bug metrics other than crashes: AFL has a hang timeout, and SlowFuzz searches for inputs that trigger worst-case behavior. The exclusion of non-crashing bugs from the evaluation leads to an under-approximation of real bugs. Their inclusion, however, enables better bug detection tools. Evaluating fuzzers based on bugs *reached*, *triggered*, and *detected* allows us to classify fuzzers and compare the advantages/disadvantages of different approaches along multiple dimensions—e.g., bugs reached allows an evaluation of the path exploration aspect, bugs triggered and detected allows a distinctive analysis of a fuzzer's constraint generation and solving component. It also allows us to identify which types of bugs continue to evade state-of-the-art sanitization techniques, and in what proportions.

8.4.4 Magma as a Lasting Benchmark. Magma leverages libraries with a long history of security bugs to build an extensible framework with a monitoring utility which collects ground truth measurements. Like most benchmarks, the wide adoption of Magma defines its utility. Benchmarks provide a common basis through which systems are evaluated and compared. For instance, the community continues to use LAVA-M to evaluate and compare fuzzers, although most of its bugs have been found and they are of a single synthetic type. Magma aims to provide an evaluation benchmark that incorporates realistic bugs in real software.

Automation is the next step in making Magma more comprehensive. However, the presented manual approach was necessary to assess the bug landscape, i.e., understanding bug types, their locations, their distribution, and challenges for finding them.

9 CONCLUSION

We designed Magma, an open ground-truth fuzzing benchmark suite that enables accurate and consistent fuzzer evaluation and performance comparison. Magma provides real targets with real bugs, along with ground-truth bug instrumentation which allows for real-time measurements of a fuzzer's performance through Magma's monitoring utility. To achieve this, we inspected vulnerability reports and identified fix commits, allowing us to forward-port bugs from old to the latest versions of real software. After carefully selecting targets with a wide variety of uses and applications, and a known history of security-critical bugs, we forward-ported 121 reported bugs and injected instrumentation that serves as ground-truth knowledge. Hence, we created Magma to be a *diverse*, *accurate*, and *usable* fuzzing benchmark.

Magma's simple design and implementation allows it to be easily improved, updated, and extended, making it ideal for open-source collaborative development and contribution. Current weaknesses will be addressed by increasing adoption: the more fuzzers are evaluated—ideally by the authors of the fuzzers—the better the metrics are defined and the more accurate the results are. Through repeated evaluations, the reachability and satisfiability of bugs can then be satisfied or disproved through discovered PoVs (or lack thereof). Additionally, Magma is extensible to support approximate bug complexity/depth metrics. Such metrics provide further insight about injected bugs, paving the way for establishing unified performance measures that allow direct comparisons between fuzzers.

We evaluated Magma against six popular, publicly-accessible, mutation-based grey-box fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, and honggfuzz). The results showed that ground-truth enables accurate measurements of fuzzer performance. Our evaluation provides tangible insight on comparing existing fuzzers, why crash counts are often misleading, and how randomness affects fuzzer performance. It also brought to light the shortcomings of some existing fault detection methods employed by fuzzers.

Despite best practices, evaluating fuzz testing remains challenging. With the adoption of ground-truth benchmarks like Magma, fuzzer evaluation will become reproducible, allowing researchers to showcase the true contributions of new fuzzing approaches.

REFERENCES

[1] Adobe Systems. [n.d.]. TIFF Specification Version 6.0. https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf. Accessed: 2019-09-06.

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. Accessed: 2019-09-09.

[3] Branden Archer and Darkkey. [n.d.]. radamsa: A Black-box mutational fuzzer. https://gitlab.com/akihe/radamsa. Accessed: 2019-09-09.

[4] Brad Arkin. 2009. Adobe Reader and Acrobat Security Initiative. http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html. Accessed: 2019-09-09.

[5] Abhishek Arya and Cris Neckar. 2012. Fuzzing for security. https://blog.chromium.org/2012/04/fuzzing-for-security.html. Accessed: 2019-09-09.

[6] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[9] Brian Caswell. [n.d.]. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/.

[10] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 711–725. https://doi.org/10.1109/SP.2018.00046

[11] N. Coppik, O. Schwahn, and N. Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 48–58. https://doi.org/10.1109/ICST.2019.00015

[12] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[13] Free Software Foundation, Inc. [n.d.]. GNU Binutils. https://www.gnu.org/software/binutils/. Accessed: 2019-09-02.

[14] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[15] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[16] Google. [n.d.]. FuzzBench. https://google.github.io/fuzzbench/. Accessed: 2020-05-02.

[17] Google. [n.d.]. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite. Accessed: 2019-09-06.

[18] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: an automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 13–20. https://doi.org/10.1145/2976002.2976017

[19] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1836089.1836091

[20] International Organization for Standardization. [n.d.]. Portable Document Format 2.0. https://www.iso.org/standard/63534.html. Accessed 2019-09-03.

[21] Internet Engineering Task Force (IETF). [n.d.]. PNG (Portable Network Graphics) Specification Version 1.0 (RFC 2083). https://tools.ietf.org/html/rfc2083. Accessed: 2019-09-06.

[22] ISO/IEC Information Technology Task Force (ITTF). [n.d.]. SQL/Framework. https://www.iso.org/standard/63555.html. Accessed: 2019-04-30.

[23] Kyriakos K. Ispoglou. 2020. FuzzGen: Automatic Fuzzer Generation. In *Proceedings of the USENIX Conference on Security Symposium*.

[24] A. S. Kalaji, R. M. Hierons, and S. Swift. 2009. Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM). In *2009 International Conference on Software Testing Verification and Validation*. 230–239. https://doi.org/10.1109/ICST.2009.29

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[26] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485. https://doi.org/10.1145/3238147.3238176

[27] LLVM Foundation. [n.d.]. libFuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed: 2019-09-06.

[28] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*.

[29] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[30] National Institute of Standards and Technology. [n.d.]. Juliet Test Suite. https://samate.nist.gov/SARD/testsuite.php. Accessed: 2019-09-03.

[31] OpenSSL Software Foundation. [n.d.]. OpenSSL. https://www.openssl.org/. Accessed: 2020-04-30.

[32] Peach Tech. [n.d.]. Peach Fuzzer Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform/. Accessed: 2019-09-09.

[33] H. Peng, Y. Shoshitaishvili, and M. Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056

[34] Hao Chen Peng Chen. [n.d.]. Fuzzing Benchmark - Real world programs. https://github.com/AngoraFuzzer/FuzzingRealProgramBenchStatistics. Accessed: 2019-09-09.

[35] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. https://doi.org/10.1109/SP.2017.27

[36] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[37] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 543–553. https://doi.org/10.1145/2970276.2970316

[38] Tim Rains. 2012. Security Development Lifecycle: A Living Process. https://www.microsoft.com/security/blog/2012/02/01/security-development-lifecycle-a-living-process/. Accessed: 2019-09-09.

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. http://dx.doi.org/10.14722/ndss.2017.23404

[40] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[41] Greg Roelofs. [n.d.]. Applications with PNG Support. http://www.libpng.org/pub/png/pngapps.html. Accessed: 2019-09-13.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[44] Daan Sprenkels. [n.d.]. LLVM provides no side-channel resistance. https://dsprenkels.com/cmov-conversion.html. Accessed: 2020-02-13.

[45] Standard Performance Evaluation Corporation. [n.d.]. SPEC Benchmark Suite. https://www.spec.org/. Accessed: 2020-02-12.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://dx.doi.org/10.14722/ndss.2016.23368

[47] The PHP Group. [n.d.]. PHP: Hypertext Preprocessor. https://www.php.net/. Accessed: 2019-04-30.

[48] W3Techs. [n.d.]. Usage statistics of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language. Accessed: 2020-05-02.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[50] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[51] Frank Warmerdam, Andrey Kiselev, Mike Welles, and Dwight Kelly. [n.d.]. TIFF Tools Overview. http://www.libtiff.org/tools.html. Accessed: 2019-09-13.

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 511–522. https://doi.org/10.1145/2508859.2516736

[53] World Wide Web Consortium (W3C). [n.d.]. XML Specification Version 1.0. https://www.w3.org/TR/xml. Accessed: 2019-09-06.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 745–761. http://dl.acm.org/citation.cfm?id=3277203.3277260

[55] Michal Zalewski. [n.d.]. American Fuzzy Lop (AFL) Technical Whitepaper. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2019-09-06.

[56] Michal Zalewski. [n.d.]. List of AFL-discovered bugs. http://lcamtuf.coredump.cx/afl/#bugs.

A BUGS AND REPORTS

Table A1: The bugs injected into Magma, and the original bug reports.

	Bug ID	Report	Type	PoV
libpng	AAH001	CVE-2018-13785	Integer overflow, divide by zero	✓
	AAH002	CVE-2019-7317	Use-after-free	✓
	AAH003	CVE-2015-8472	API inconsistency	✓
	AAH004	CVE-2015-0973	Integer overflow	✗
	AAH005	CVE-2014-9495	Integer overflow, Buffer overflow	✓
	AAH007	(Unspecified)	Memory leak	✓
	AAH008	CVE-2013-6954	0-pointer dereference	✓
libtiff	AAH009	CVE-2016-9535	Heap buffer overflow	✓
	AAH010	CVE-2016-5314	Heap buffer overflow	✓
	AAH011	CVE-2016-10266	Divide by zero	✗
	AAH012	CVE-2016-10267	Divide by zero	✗
	AAH013	CVE-2016-10269	OOB read	✓
	AAH014	CVE-2016-10269	OOB read	✓
	AAH015	CVE-2016-10270	OOB read	✓
	AAH016	CVE-2015-8784	Heap buffer overflow	✓
	AAH017	CVE-2019-7663	0-pointer dereference	✓
	AAH018	CVE-2018-8905	Heap buffer underflow	✓
	AAH019	CVE-2018-7456	OOB read	✗
	AAH020	CVE-2016-3658	Heap buffer overflow	✓
	AAH021	CVE-2018-18557	OOB write	✗
	AAH022	CVE-2017-11613	Resource Exhaustion	✓
libxml2	AAH024	CVE-2017-9047	Stack buffer overflow	✓
	AAH025	CVE-2017-0663	Type confusion	✓
	AAH026	CVE-2017-7375	Xml external entity	✓
	AAH027	CVE-2018-14567	Resource exhaustion	✗
	AAH028	CVE-2017-5130	Integer overflow, heap corruption	✗
	AAH029	CVE-2017-9048	Stack buffer overflow	✗
	AAH030	CVE-2017-8872	OOB read	✗
	AAH031	ISSUE #58 (gitlab)	OOB read	✗
	AAH032	CVE-2015-8317	OOB read	✓
	AAH033	CVE-2016-4449	XML external entity	✗
	AAH034	CVE-2016-1834	Heap buffer overflow	✗
	AAH035	CVE-2016-1836	Use-after-free	✗
	AAH036	CVE-2016-1837	Use-after-free	✗
	AAH037	CVE-2016-1838	Heap buffer overread	✓
	AAH038	CVE-2016-1839	Heap buffer overread	✗
	AAH039	BUG 758518	Heap buffer overread	✗
	AAH040	CVE-2016-1840	Heap buffer overflow	✗
	AAH041	CVE-2016-1762	Heap buffer overread	✓
poppler	AAH042	CVE-2019-14494	Divide-by-zero	✓
	AAH043	CVE-2019-9959	Resource exhaustion (memory)	✓
	AAH045	CVE-2017-9865	Stack buffer overflow	✓
	AAH046	CVE-2019-10873	0-pointer dereference	✓
	AAH047	CVE-2019-12293	Heap buffer overread	✓
	AAH048	CVE-2019-10872	Heap buffer overflow	✓
	AAH049	CVE-2019-9200	Heap buffer underwrite	✓
	AAH050	Bug #106061	Divide-by-zero	✓
	AAH051	ossfuzz/8499	Integer overflow	✓
	AAH052	Bug #101366	0-pointer dereference	✓
	JCH201	CVE-2019-7310	Heap buffer overflow	✓
	JCH202	CVE-2018-21009	Integer overflow	✗
	JCH203	CVE-2018-20650	Type confusion	✗
	JCH204	CVE-2018-20481	0-pointer dereference	✗
	JCH206	CVE-2018-19058	Type confusion	✗
	JCH207	CVE-2018-13988	OOB read	✓
	JCH208	CVE-2019-12360	Stack buffer overflow	✗
	JCH209	CVE-2018-10768	0-pointer dereference	✗
	JCH210	CVE-2017-9776	Integer overflow	✗
	JCH211	CVE-2017-18267	Resource exhausion (CPU)	✗
	JCH212	CVE-2017-14617	Divide-by-zero	✓
	JCH214	CVE-2019-12493	Stack buffer overread	✗
openssl	AAH053	CVE-2016-0705	Double-free	✓
	AAH054	CVE-2016-2842	OOB write	✗
	AAH055	CVE-2016-2108	OOB read	✓
	AAH056	CVE-2016-6309	Use-after-free	✓
	AAH057	CVE-2016-2109	Resource exhaustion (memory)	✗
	AAH058	CVE-2016-2176	Stack buffer overread	✗
	AAH059	CVE-2016-6304	Resource exhaustion (memory)	✗
	MAE100	CVE-2016-2105	Integer overflow	✗
	MAE102	CVE-2016-6303	Integer overflow	✗
	MAE103	CVE-2017-3730	0-pointer dereference	✗
	MAE104	CVE-2017-3735	OOB read	✓
	MAE105	CVE-2016-0797	Integer overflow	✗
	MAE106	CVE-2015-1790	0-pointer dereference	✗
	MAE107	CVE-2015-0288	0-pointer dereference	✗
	MAE108	CVE-2015-0208	0-pointer dereference	✗
	MAE109	CVE-2015-0286	Type confusion	✗
	MAE110	CVE-2015-0289	0-pointer dereference	✗
	MAE111	CVE-2015-1788	Resource exhaustion (CPU)	✗
	MAE112	CVE-2016-7052	0-pointer dereference	✗
	MAE113	CVE-2016-6308	Resource exhaustion (memory)	✗
	MAE114	CVE-2016-6305	Resource exhaustion (CPU)	✗
	MAE115	CVE-2016-6302	OOB read	✓
sqlite3	JCH214	CVE-2019-9936	Heap buffer overflow	✗
	JCH215	CVE-2019-20218	Stack buffer overread	✓
	JCH216	CVE-2019-19923	0-pointer dereference	✗
	JCH217	CVE-2019-19959	OOB read	✗
	JCH218	CVE-2019-19925	0-pointer dereference	✗
	JCH219	CVE-2019-19244	OOB read	✗
	JCH220	CVE-2018-8740	0-pointer dereference	✗
	JCH221	CVE-2017-15286	0-pointer dereference	✗
	JCH222	CVE-2017-2520	Heap buffer overflow	✗
	JCH223	CVE-2017-2518	Use-after-free	✗
	JCH225	CVE-2017-10989	Heap buffer overflow	✗
	JCH226	CVE-2019-19646	Logical error	✓
	JCH227	CVE-2013-7443	Heap buffer overflow	✗
	JCH228	CVE-2019-19926	Logical error	✓
	JCH229	CVE-2019-19317	Resource exhaustion (memory)	✗
	JCH230	CVE-2015-3415	Double-free	✗
	JCH231	CVE-2020-9327	0-pointer dereference	✗
	JCH232	CVE-2015-3414	Uninitialized memory access	✓
	JCH233	CVE-2015-3416	Stack buffer overflow	✗
	JCH234	CVE-2019-19880	0-pointer dereference	✗
php	MAE001	CVE-2019-9020	Use-after-free	✗
	MAE002	CVE-2019-9021	Heap buffer overread	✗
	MAE004	CVE-2019-9641	Uninitialized memory access	✗
	MAE006	CVE-2019-11041	OOB read	✗
	MAE008	CVE-2019-11034	OOB read	✓
	MAE009	CVE-2019-11039	OOB read	✗
	MAE010	CVE-2019-11040	Heap buffer overflow	✗
	MAE011	CVE-2018-20783	OOB read	✗
	MAE012	CVE-2019-9022	OOB read	✗
	MAE013	CVE-2019-9024	OOB read	✗
	MAE014	CVE-2019-9638	Uninitialized memory access	✓
	MAE015	CVE-2019-9640	OOB read	✗
	MAE016	CVE-2018-14883	Heap buffer overread	✗
	MAE017	CVE-2018-7584	Stack buffer underread	✗
	MAE018	CVE-2017-11362	Stack buffer overflow	✗
	MAE019	CVE-2014-9912	OOB write	✗
	MAE020	CVE-2016-10159	Integer overflow	✗
	MAE021	CVE-2016-7414	OOB read	✗

B REACHED BUGS

Table A2: Expected time-to-reach-bug values for ten 24-hour campaigns. Of the 121 injected bugs, 72 were reached.

Bug ID	hfuzz	fairfuzz	afl	aflfast	afl++	moptafl	Aggregate
AAH020	5s	5s	5s	5s	5s	5s	5s
AAH054	5s	5s	5s	5s	5s	5s	5s
JCH207	5s	5s	5s	5s	5s	5s	5s
MAE016	5s	5s	5s	5s	5s	5s	5s
MAE105	5s	5s	5s	5s	5s	5s	5s
AAH003	10s	5s	5s	5s	5s	5s	6s
AAH037	10s	5s	5s	5s	5s	5s	6s
AAH007	5s	10s	10s	10s	10s	10s	9s
AAH032	5s	10s	10s	10s	10s	10s	9s
AAH001	10s	10s	10s	10s	10s	10s	10s
AAH004	10s	10s	10s	10s	10s	10s	10s
AAH005	10s	10s	10s	10s	10s	10s	10s
AAH008	10s	10s	10s	10s	10s	10s	10s
AAH011	10s	10s	10s	10s	10s	10s	10s
AAH024	10s	10s	10s	10s	10s	10s	10s
AAH026	10s	10s	10s	10s	10s	10s	10s
AAH029	10s	10s	10s	10s	10s	10s	10s
AAH034	10s	10s	10s	10s	10s	10s	10s
AAH041	10s	10s	10s	10s	10s	10s	10s
AAH049	10s	10s	10s	10s	10s	10s	10s
AAH055	10s	10s	10s	10s	10s	10s	10s
AAH056	10s	10s	10s	10s	10s	10s	10s
JCH201	10s	10s	10s	10s	10s	10s	10s
JCH202	10s	10s	10s	10s	10s	10s	10s
JCH212	10s	10s	10s	10s	10s	10s	10s
MAE004	10s	10s	10s	10s	10s	10s	10s
MAE006	10s	10s	10s	10s	10s	10s	10s
MAE008	10s	10s	10s	10s	10s	10s	10s
MAE014	10s	10s	10s	10s	10s	10s	10s
MAE104	10s	10s	10s	10s	10s	10s	10s
MAE111	10s	10s	10s	10s	10s	10s	10s
MAE114	10s	10s	10s	10s	10s	10s	10s
MAE115	10s	10s	10s	10s	10s	10s	10s
AAH035	10s	10s	10s	10s	15s	10s	11s
AAH052	15s	10s	10s	10s	10s	10s	11s
AAH048	10s	10s	15s	10s	12s	10s	11s
AAH059	10s	10s	15s	15s	10s	15s	12s
JCH204	20s	10s	15s	15s	10s	15s	14s
AAH045	14s	15s	15s	15s	15s	15s	15s
AAH031	15s	15s	15s	15s	34s	15s	18s
AAH051	10s	20s	20s	20s	35s	20s	21s
MAE103	28s	20s	28s	25s	20s	25s	24s
JCH210	20s	25s	30s	25s	28s	25s	26s
AAH053	35s	26s	30s	30s	28s	30s	30s
JCH214	45s	25s	31s	26s	31s	26s	31s
AAH042	20s	31s	40s	34s	34s	34s	32s
AAH015	10s	1m	1m	52s	1m	50s	60s
AAH022	10s	1m	1m	52s	1m	50s	60s
AAH043	47h	20s	25s	20s	20s	20s	1h
AAH047	47h	20s	25s	20s	20s	20s	1h
JCH215	15s	3h	2h	48m	1h	14m	1h
JCH220	12s	3h	2h	54m	1h	6h	1h
JCH229	16s	3h	2h	1h	1h	6h	1h
AAH018	4m	36m	3h	59m	4h	2h	2h
JCH230	22s	3h	3h	1h	2h	7h	2h
JCH223	30s	4h	3h	1h	2h	7h	2h
JCH231	36s	4h	3h	1h	3h	7h	2h
JCH233	12m	3h	3h	1h	3h	7h	2h
AAH050	47h	20s	–	20s	25s	20s	5h
AAH009	50h	22h	5h	24h	5h	4h	10h
JCH232	43m	4h	86h	15h	14h	48h	13h
AAH010	11h	14h	10m	–	–	10m	13h
AAH017	205h	5h	19h	8h	88h	12h	13h
JCH222	21m	24h	28h	18h	30h	–	22h
JCH228	2h	24h	205h	–	20h	47h	30h
AAH014	–	23h	4h	–	–	47h	58h
AAH013	4h	–	–	–	–	–	110h
JCH226	4h	–	–	–	–	–	110h
AAH016	–	50h	–	47h	–	–	205h
JCH227	87h	–	–	–	–	–	684h
JCH219	206h	–	–	207h	–	–	684h
AAH025	205h	–	–	–	–	–	1404h

C WEEK-LONG CAMPAIGNS

Table A3: Expected time-to-trigger-bug for a subset of Magma targets and bugs over 7-day campaigns.

Bug ID	hfuzz	afl	aflfast	afl++	moptafl	fairfuzz	Aggregate
AAH003	10s	10s	10s	10s	10s	10s	10s
AAH041	10s	15s	16s	15s	15s	15s	14s
AAH056	46s	5m	5m	3m	4m	6m	4m
AAH015	14s	10m	13m	12m	9m	1h	19m
AAH052	16m	12h	2m	2h	9m	3m	2h
AAH032	17s	16m	51m	21m	58h	58m	5h
AAH022	2m	22h	1h	2h	11h	4h	7h
AAH055	19m	2h	33m	1h	42h	2h	8h
AAH017	212h	9h	10h	8h	10h	12h	24h
AAH008	5h	57h	144h	105h	–	24h	73h
AAH045	1m	1450h	142h	122h	–	–	174h
AAH010	21h	616h	189h	88h	70h	348h	202h
AAH053	106h	1450h	213h	1441h	367h	101h	231h
AAH014	75h	120h	621h	–	92h	96h	233h
AAH018	529h	1444h	343h	1438h	116h	614h	503h
AAH001	157h	364h	–	1440h	–	164h	504h
AAH016	277h	1451h	351h	605h	209h	611h	612h
AAH007	33m	–	–	–	–	–	767h
AAH024	1448h	–	1449h	233h	–	–	1445h
AAH025	1436h	–	–	1447h	–	–	4792h
AAH013	5h	–	–	–	1443h	–	9830h
AAH026	1445h	–	–	–	–	–	9830h
AAH009	–	1448h	–	–	–	–	9831h
AAH035	1450h	–	–	–	–	–	9831h