**Magma: A Ground-Truth Fuzzing Benchmark**

Ahmad Hazimeh
EPFL

Adrian Herrera
Australian National University & Defence Science and Technology Group

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 in the wild, evaluating and comparing fuzzer performance is challenging due to the lack of metrics and benchmarks. For example, crash count—perhaps the most commonly-used performance metric—is inaccurate due to imperfections in deduplication techniques. Additionally, the lack of a unified set of targets results in ad hoc evaluations that hinder fair comparison.

We tackle these problems by developing Magma, a ground-truth fuzzing benchmark that enables uniform fuzzer evaluation and comparison. By introducing real bugs into real software, Magma allows for the realistic evaluation of fuzzers against a broad set of targets. By instrumenting these 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 widely-used mutation-based greybox fuzzers (AFL, AFLFast, AFL++, FAIRFuZZ, MOpt-AFL, and honggfuzz) against Magma over 200,000 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 fuzzed performance evaluation, highlighting the importance of ground truth in performing more accurate and meaningful evaluations.

**1 INTRODUCTION**

Fuzz testing (“fuzzing”) is a widely-used dynamic bug discovery technique. It involves generating a large number of inputs and subjecting a target program to these inputs with the aim of triggering a fault (i.e., discovering a bug). Fuzzing is an inherently sound but incomplete bug-finding process (given finite resources). In particular, state-of-the-art fuzzers rely on crashes to mark faulty program behavior. The existence of a crash is generally symptomatic of a bug (soundness), but the lack of a crash does not necessarily mean that the program is bug-free (incompleteness). Fuzzing has been wildly successful in finding thousands of bugs in open-source software, but the lack of a crash does not necessarily mean that the program is bug-free. Fuzzing techniques claiming to improve bug-finding performance [34] are constantly being improved in a race to build the ultimate bug-finding tool. However, while fuzzing excels at finding bugs in the wild, evaluating and comparing fuzzer performance is challenging due to the lack of metrics and benchmarks. For example, crash count—perhaps the most commonly-used performance metric—is inaccurate due to imperfections in deduplication techniques. Additionally, the lack of a unified set of targets results in ad hoc evaluations that hinder fair comparison.

While these metrics provide some insight into a fuzzer’s performance, we argue that they are insufficient for use in fuzzer comparisons. Furthermore, the set of target programs that these metrics are evaluated on can vary wildly across papers, making cross-paper comparisons impossible. The deficiencies of these three metrics are discussed in turn.

**Crash counts.** The simplest method for evaluating a fuzzer is to count the number of crashes triggered by that fuzzer, and compare this crash count with that achieved by another fuzzer on the same target program. Unfortunately, crash counts often inflate the number of actual bugs in the target program [29]. Moreover, deduplication techniques (e.g., coverage profiles, stack hashes) fail to accurately identify the root cause of these crashes [9, 29].

**Bug counts.** Identifying a crash’s root cause is preferable to simply reporting raw crashes, as it avoids the inflation problem inherent in crash counts. Unfortunately, obtaining an accurate ground-truth bug count typically requires extensive manual triage, which in turn requires someone with extensive domain expertise and experience [1].

**Code-coverage profiles.** Due to the difficulty in obtaining ground-truth bug counts, code-coverage profiles are another performance metric commonly used to evaluate and compare fuzzing techniques. Intuitively, covering more code correlates with finding more bugs. However, previous work [29] 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.

The deficiencies of existing performance metrics calls for a rethink of fuzzer evaluation practices. In particular, the performance metrics used in these evaluations must accurately measure a fuzzer’s ability to achieve its main objective: finding bugs. Similarly, the target programs that are used to assess how well a fuzzer meets this objective must be realistic and exercise diverse behavior. This allows a practitioner to have confidence that a given fuzzing technique will yield improvements when deployed in real-world environments.

To satisfy these criteria, we present Magma, a ground-truth fuzzing benchmark based on real programs with real bugs. Magma consists of seven widely-used open-source libraries and applications, totalling 2 MLOC. For each Magma workload, we manually analyze security-relevant bug reports and patches, reinserting defective code back into these seven libraries and applications (in total, 118 bugs were analyzed and reinserted). Additionally, each reinserted bug is accompanied by an oracle that detects and reports if the bug is reached or triggered. This distinction between reaching and triggering a bug—additionally, a triggered bug—presents a new opportunity to evaluate a fuzzer across multiple dimensions (again, focusing on ground-truth bugs).
The remainder of this paper presents the motivation behind Magma, the methodology behind Magma’s design and choice of performance metrics, implementation details, and a set of preliminary results that demonstrate Magma’s utility. We make the following contributions:

- A set of bug-centric performance metrics that should be measurable in a fuzzer benchmark, allowing for a fair and accurate evaluation and comparison of fuzzers.
- A methodology for selecting workloads (both targets and bugs) in a fuzzing benchmark.
- The design and implementation of Magma, a ground-truth fuzzing benchmark based on real programs with real bugs.
- An evaluation of Magma against six widely-used fuzzers.

Magma is open-source and available from https://hexhive.epfl.ch/magma.

2 BACKGROUND AND MOTIVATION

This section introduces fuzzing as a software testing technique, and how new fuzzing techniques are currently evaluated and compared against existing ones. This aims to motivate the need for new fuzzer evaluation practices.

2.1 Fuzzing

A fuzzer is a dynamic analysis tool that discovers software flaws by running a target program with a large number of automatically-generated inputs. Notably, these inputs are generated with the intention of triggering a crash in the target program. This input generation process is dependent on a fuzzer’s knowledge of the target’s input format and program structure.

For example, grammar-based fuzzers (e.g., Superion [55], Peachfuzz [37], and QuickFuzz [21]) leverage the target program’s input format (which must be specified a priori) to intelligently craft inputs (e.g., based on data width and type, and on the relationships between different input fields). In contrast, mutational fuzzers (e.g., AFL [58], Angora [12], and MemFuzz [13]) require no a priori knowledge of the input format. Instead, mutational fuzzers leverage preprogrammed mutation operations to iteratively modify the input.

Fuzzers are also classified by their knowledge of the target’s program structure. For example, whitebox fuzzers [16, 17, 41] leverage program analysis to infer knowledge about the program structure. In comparison, blackbox fuzzers [4, 56] blindly generate inputs in the hope of discovering a crash. Finally, greybox fuzzers [12, 31, 58] leverage program instrumentation (instead of program analysis) to collect runtime information. Program-structure knowledge guides input generation in a manner more likely to trigger a crash.

Importantly, fuzzing is a highly stochastic bug-finding process. This statement remains true irrespective of whether the fuzzer synthesizes inputs from a grammar (grammar-based fuzzing), transforms an existing set of inputs to arrive at new inputs (mutational fuzzing), has no knowledge of that target program’s internals (blackbox fuzzing), or uses sophisticated program analyses to understand the target program (whitebox fuzzing). This inherent randomness makes evaluating and comparing fuzzers difficult. This problem is exacerbated by existing fuzzer evaluation metrics and benchmarks.

2.2 The Current State of Fuzzer Evaluation

The rapid emergence of new and improved fuzzing techniques [34] means that fuzzers are constantly compared against one another, in order to empirically demonstrate that the latest fuzzer supersedes previous state-of-the-art fuzzers. To enable fair and accurate fuzzer evaluation, it is critical that fuzzing campaigns are conducted on a suitable benchmark that uses an appropriate set of metrics. Unfortunately, fuzzer evaluations have so far been ad hoc and haphazard. For example, Klees et al.’s study of 32 fuzzing papers found that none of the surveyed papers provided sufficient detail to support their claims of fuzzer improvement [29]. Notably, their study highlights a set of criteria that should be adopted across all fuzzer evaluations. These criteria include:

- **Performance metrics:** How the fuzzers are evaluated and compared. This is typically one of the approaches previously discussed (crash count, bug count, or coverage profiling).
- **Targets:** The software being fuzzed. Fuzzers should be evaluated on diverse, realistic workloads.
- **Seed selection:** The initial set of inputs that bootstrap the fuzzing process. This initial set of inputs should be consistent across repeated trials and the fuzzers under evaluation.
- **Trial duration (timeout):** The length of a single fuzzing trial should also be consistent across repeated trials and the fuzzers under evaluation. We use the term trial to refer to an instance of the fuzzing process on a target program, while a fuzzing campaign is a set of N repeated trials on the same target program.
- **Number of trials:** The highly-stochastic nature of fuzzing necessitates a large number of repeated trials, allowing for a statistically sound comparison of results.

Klees et al.’s study demonstrates the need for a ground-truth fuzzing benchmark. Such a benchmark must use suitable performance metrics and present a unified set of target programs.

2.2.1 Existing Fuzzer Benchmarks. Fuzzers are typically evaluated on a set of target programs sourced from one of the following benchmarks. These benchmarks are summarized in Table 1.

The LAVA-M [14] test suite aims to evaluate the effectiveness of a fuzzer’s exploration capability by injecting bugs in different execution paths. However, the LAVA bug injection technique only injects a single, simple bug type: an out-of-bounds memory access triggered by a “magic value” comparison. This bug type does not accurately represent the statefulness and complexity of bugs encountered in real-world software.

In contrast, the Cyber Grand Challenge (CGC) [11] sample set provides a wider variety of bugs that are suitable for testing a fuzzer’s fault detection capabilities. Unfortunately, the relatively small size and simplicity of the CGC’s synthetic workloads does not enable thorough evaluation of the fuzzer’s ability to explore complex programs.

BugBench [32] and the Google Fuzzer Test Suite (FTS) [19] both contain real programs with real bugs. However, each target program only contains one or two bugs (on average). This sparsity of bugs, combined with the lack of automatic methods for triaging crashes, hinders adoption and makes both benchmarks unsuitable for fuzzer evaluation. In contrast, Google FuzzBench [18]—the successor to
Table 1: A summary of existing fuzzer benchmarks and our benchmark, Magma. We characterize benchmarks across two dimensions: the target programs that make up the benchmark workloads and the bugs that exist across these workloads. For both dimensions we count the number of workloads/bugs (#) and classify them as Real or Synthetic. Bug density is the mean number of bugs per workload. Finally, ground truth may be available (√), available but not easily accessible (Ψ), or unavailable (✓).

| Benchmark      | Workloads | Bugs | Bug Density | Ground truth |
|----------------|-----------|------|-------------|--------------|
| LAVA-M [14]    | R         | S    | 2265        | 566.25       |
| CGC [11]       | R         | S    | 590         | 4.50         |
| BugBench [32]  | R         | R    | 19          | 1.12         |
| Google FTS [19]| R         | R    | 47          | 1.96         |
| Google FuzzBench [18] | R | R | 21 | 1.26 |
| Open-source software | R | R | ? | ? |
| Magma          | R         | R    | 118         | 16.86        |

the Google FTS—is a fuzzer evaluation platform that relies solely on coverage profiles as a performance metric. As previously discussed, this metric has severe limitations when evaluating fuzzers on their bug-finding capability.

Finally, popular open-source software (OSS) is often used to evaluate fuzzers [10, 29, 30, 33, 38, 54]. Although real-world software is used, the lack of ground-truth knowledge about the triggered crashes makes it difficult to provide an accurate, verifiable, quantitative evaluation. Furthermore, it is often unclear which software version is used, making fair cross-paper comparisons impossible.

2.2.2 Crashes as a Performance Metric. Most, if not all, state-of-the-art fuzzers implement fault detection as a crash listener. A program crash can be caused by an architectural violation (e.g., division-by-zero, unmapped/unprivileged page access) or by a sanitizer (a dynamic bug-finding tool that generates a crash when a security policy violation—e.g., object out-of-bounds, type safety violation—occurs [48]).

The simplicity of crash detection has led to the widespread use of crash count as a performance metric for comparing fuzzers. However, crash counts have been shown to yield inflated results, even when combined with deduplication methods (e.g., coverage profiles and stack hashes) [9, 29]. Instead, the number of bugs found by each fuzzer can be compared: 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, deriving formal program models 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, resulting in a duplication of crashes and an inflated set of results.

3 DESIRED BENCHMARK PROPERTIES

Benchmarks are important drivers for computer science research and product development [8]. Several factors must be taken into account when designing a benchmark, including: relevance; reproducibility; fairness; verifiability; and usability [2, 52]. While building benchmarks around these properties is well studied [2, 8, 24, 28, 32, 43, 45, 50, 52], the highly-stochastic nature of fuzzing introduces new challenges for benchmark designers.

For example, reproducibility is a key benchmark property that ensures a benchmark produces "the same results consistently for a particular test environment" [52]. However, individual fuzzing trials vary wildly in performance, requiring a large number of repeated trials for a particular test environment [29]. While performance variance exists in most benchmarks (e.g., the SPEC CPU benchmark [50] uses the median of three repeated trials to account for small variations across environments), this variance is far greater in fuzzing. Furthermore, a fuzzer may actively modify the test environment (e.g., T-Fuzz [38] and FuzzGen [26] transform the target program, while Skyfire [54] generates new seed inputs for the target). This is very different to traditional performance benchmarks (e.g., SPEC CPU [50], DaCapo [8]), where the workloads and their inputs remain fixed across all systems-under-test (SUT).

This leads us to define the following set of properties that we argue must exist in a fuzzing benchmark:

- **Diversity (P1)**: The benchmark contains a wide variety of bugs and programs that resemble real software testing scenarios.
- **Verifiability (P2)**: The benchmark yields verifiable metrics that accurately describe performance.
- **Usability (P3)**: The benchmark is accessible and has no significant barriers for adoption.

These three properties are explored in the following sections, while Section 4 describes how Magma satisfies these criteria.

3.1 Diversity (P1)

Fuzzers are actively used to find bugs in a variety of real programs [3, 5, 6, 44]. Therefore, a fuzzing benchmark must evaluate fuzzers against programs and bugs that resemble those encountered in the "real world". To this end, a benchmark must include a diverse set of bugs and programs.

- Bugs should be diverse with respect to:
  - **Class**: Common Weakness Enumeration (CWE) [36] bug classes include memory-based errors, type errors, concurrency issues, and numeric errors.
  - **Distribution**: "Depth", fan-in (i.e., the number of paths which execute the bug), and spread (i.e., the ratio of faulty-path counts to the total number of paths).
  - **Complexity**: Number of input bytes involved in triggering a bug, the range of input values which triggers the bug, and the transformations performed on the input.
Similarly, programs (i.e., the benchmark workloads) should be diverse with respect to:

**Application domain:** File and media processing, network protocols, document parsing, cryptography primitives, and data encoding.

**Operations performed:** Parsing, checksum calculation, indirection, transformation, state management, and data validation.

**Input structure:** Binary, text, formats/grammars, and data size.

Satisfying this property requires bugs that resemble those encountered in real-world environments. Both LAVA-M and Google FuzzBench fail this requirement: the former contains only a single bug class (an out-of-bounds memory access), while FuzzBench does not measure for *any* bugs. However, both Google FTS and FuzzBench contain workloads from a wide variety of application domains, including cryptography, image parsing, text processing, and compilers. Finally, BugBench primarily focuses on memory corruption vulnerabilities, but also contains uninitialized read, memory leak, data race, atomicity, and semantic bugs (for a total of nine bug classes).

Ultimately, real programs are the only source of real bugs. Therefore, a benchmark designed to evaluate fuzzers must include *real programs with a variety of real bugs,* thus ensuring diversity and avoiding bias (e.g., towards a specific bug class). Whereas discovering and reporting real bugs is desirable (i.e., when OSS is used), performance metrics based on an unknown set of bugs (with an unknown distribution) make it impossible to compare fuzzers. Instead, fuzzers should be evaluated on workloads containing known bugs for which ground truth is available and *verifiable.*

### 3.2 Verifiability (P2)

Existing ground-truth fuzzing benchmarks lack a straightforward mechanism for determining a crash’s root cause. This makes it difficult to *verify* a fuzzer’s results. Crash count, the current widely-used performance metric, suffers from high variability, double-counting, and inconsistent results across multiple trials (see Section 2.2.2). Automated techniques for deduplicating crashes are not reliable, and hence should not be used to verify the bugs discovered by a fuzzer. Ultimately, a fuzzing benchmark should provide a set of known bugs for which ground truth can be used to verify a fuzzer’s findings.

While the CGC sample set provides crashing inputs—also known as a *proof of vulnerability (PoV)*—for all known bugs, it does not provide a mechanism for determining the root cause of a fuzzergenerated crash. Similarly, the Google FTS provides PoVs (for 87% of bugs) and a script for triaging and deduplicating crashes. This script parses the crash report or looks for a specific line of code at which to terminate program execution. 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 to the CGC and Google FTS benchmarks, for which ground truth is available but not easily accessible, LAVA-M clearly reports the bug triggered by a crashing input. However, LAVA-M does not provide a runtime interface for accessing this information. Unless a fuzzer is specialized to collect LAVA-M metrics, it cannot monitor progress in real-time. Thus, a post-processing step is required to collect metrics. Finally, Google 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 sacrifices the significance of the results: more coverage does not necessarily imply higher bug-finding effectiveness.

Ground-truth bug knowledge allows for a fuzzer’s findings to be verified, enabling accurate performance evaluation and allowing meaningful comparisons between fuzzers. To this end, a fuzzing benchmark must provide *easy access to ground-truth metrics* describing the bugs a fuzzer can reach, trigger, and detect.

### 3.3 Usability (P3)

Fuzzers have evolved from simple blackbox random-input generation to complex data- and control-flow analysis tools. Each fuzzer may introduce its own instrumentation into a target binary (e.g., AFL [58]), launch the program in a specific execution engine (e.g., QSYM [57], Driller [51]), or provide inputs through a specific channel (e.g., libFuzzer [31]). Fuzzers come in a variety of forms (described in Section 2.1), so a fuzzing benchmark must not exclude a particular type of fuzzer. Additionally, using a benchmark must be manageable and straightforward: it should not require constant user intervention, and benchmarking should finish within a reasonable time frame. The inherent randomness of fuzzing complicates this, as multiple trials are required to achieve statistically-meaningful results.

Some existing benchmark workloads (e.g., those from CGC and Google FTS) contain multiple bugs, so it is not sufficient to only run the fuzzer until the first crash is encountered. However, the lack of easily-accessible ground truth makes it difficult to determine if/when all bugs are triggered. Moreover, inaccurate deduplication techniques mean that the user cannot simply equate the number of crashes with the number of bugs. Thus, additional time must be spent triaging crashes to obtain ground-truth bug counts, further complicating the benchmarking process.

In summary, a benchmark should be *usable* by fuzzer developers, without introducing insurmountable or impractical barriers to adoption. To satisfy this property, a benchmark must thus provide a *small set of targets with a large number of discoverable bugs,* and it must provide a *usable framework that measures and reports fuzzer progress and performance.*

### 4 MAGMA: APPROACH

We present Magma, a ground-truth fuzzing benchmark that satisfies the previously-discussed benchmark properties. Magma is a collection of seven targets with widespread use in real-world environments. These initial targets have been carefully selected for their *diversity* and the variety of security-critical bugs that have been reported throughout their lifetimes (satisfying P1).

Importantly, Magma’s seven workloads contain 118 bugs for which ground truth is *easily accessible* and *verifiable* (satisfying P2). These bugs are sourced from older versions of the seven workloads, and then *forward-ported* to the latest version contained within Magma. Finally, Magma imposes minimal requirements on the user, allowing fuzzer developers to seamlessly integrate the benchmark into their development cycle (satisfying P3).
4.1 Overview
For each workload, we manually inspect bug and vulnerability reports to find bugs that are suitable for inclusion in Magma (e.g., ensuring that the bug affects the core codebase). For these bugs, we reintroduce (“inject”) each bug into the latest version of the code through a process we call forward-porting (described more in Section 4.3). In addition to the bug, we also insert minimal source-code instrumentation—a canary—to collect data about a fuzzer’s ability to reach and trigger the bug (discussed more in Section 4.4). A bug is reached when the faulty line of code is executed, and triggered when the fault condition is satisfied. Finally, Magma provides a runtime monitor that runs in parallel with the fuzzer to collect real-time statistics. These statistics are used to evaluate the fuzzer (see Section 4.5).

Fuzzer evaluation is based on the number of bugs reached, triggered, and detected. The Magma instrumentation only yields usable information when the fuzzer exercises the instrumented code; this allows us to determine whether a bug is reached. The dataflow generated by the fuzzer triggers a bug when that dataflow satisfies the bug’s trigger conditions. Once triggered, the fuzzer should flag the bug as a fault or crash, enabling us to assess the fuzzer’s bug detection capability. These metrics are described further in Section 4.4.

Finally, Magma provides a fatal canaries mode, where, if a canary’s condition is satisfied, the program is terminated (similar to LAVA-M). The fuzzer then saves this crashing input 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 additional sanitization overhead. This mode increases the number of executions during an evaluation, reducing the cost of evaluating a fuzzer but sacrificing the ability to evaluate a fuzzer’s detection capabilities.

4.2 Target Selection
Magma contains seven targets, which we summarize in Table 2. In addition to these seven targets (the codebases into which bugs are injected), Magma also includes 19 drivers (executable programs that provide a command-line interface to the target) that exercise different functionality within the target. Inspired by Google OSS-Fuzz [3], these drivers are sourced from the original target codebases (as drivers are best developed by domain experts).

Magma’s seven targets were selected for their diversity in functionality (summarized in Table 2). Inspired by existing benchmarks in other fields [8, 43], we apply Principal Component Analysis (PCA) to quantify this diversity. We draw from Section 3.1 and use the following low- and high-level PCA features, extracted from the LLVM intermediate representation of a target program:

Instructions: The type of instructions used (e.g., binary operations, bitwise binary operations, memory access and addressing operations).

Complexity: Cyclomatic [35] and Halstead [23] complexity metrics.

APIs: The libraries and components used by the target.

4.3 Bug Selection and Insertion
Magma contains 118 bugs, spanning 12 bug classes (summarized in Figure 1; the complete list of bugs is given in Table A1). Compared to existing benchmarks, Magma contains the widest variety of bugs (followed by Google FTS, with ten different bug classes). Moreover, Magma has the second-largest “bug density”—the ratio of the number of bugs to the number of targets—after LAVA-M (see Table 1). While LAVA-M’s bug density (566.25) is an order-of-magnitude larger than Magma’s (16.86), it is important to note that LAVA-M is restricted to a single, synthetic bug type.

In addition to a diversity of bug classes, Magma also contains bugs with a range of complexities. We approximate bug complexity by the set of constraints required to trigger a bug: the greater the number of constraints, the more obstacles the fuzzer must overcome to reach and trigger a bug. Specifically, bug complexity is approximated by: (i) minimizing a PoV with afl-tmin [58]; (ii) concolically executing these minimized PoVs in SymCC [42] to produce a set of path constraints; (iii) bit-blasting these path constraints from quantifier-free bitvector SMT to conjunctive normal form (CNF) SAT; and then (iv) counting the number of clauses in the CNF SAT equation (intuitively, the more clauses in the normalized constraint set, the harder the fuzzer must work to satisfy these constraints and thus trigger the bug).

Finally, Magma contains real bugs sourced from bug reports and forward-ported to the most recent version of the target codebase. This is in contrast to existing fuzzing benchmarks (e.g., BugBench, Google FTS) that rely on old, unpatched versions of the target codebase. Unfortunately, using older codebases limits the number of bugs available in each target (as evident by the low bug densities in Table 1). In comparison, forward-porting—which is synonymous to back-porting fixes from newer codebases to older, buggy releases—does not suffer from this issue, making Magma’s targets easily extensible.

Forward-porting begins with the identification—from the reported bug fix—of the code changes that must be reverted to reintroduce the bug. Bug-fix commits can contain multiple fixes to
one or more bugs, so disambiguation is necessary to prevent the introduction of unintended bugs. Alternatively, bug fixes may be spread over multiple commits (e.g., if the original fix did not cover all edge cases). Following the identification of code changes, we identify what program state is involved in evaluating the trigger condition. If necessary, we introduce additional program variables to access that state. From this state, we determine a boolean expression that serves as an oracle for identifying a triggered bug. Finally, we identify a point in the program where we inject a canary before the bug can manifest faulty behavior. This canary helps measure our fuzzer performance metrics, discussed in the following section.

4.4 Performance Metrics

Fuzzer evaluation has traditionally relied on crash counts, bug counts, and/or code-coverage profiles for measuring and comparing fuzzer performance. While the problems with crash counts and code-coverage profiles are well known (see Section 2.2.2), in our view, simply counting the number of bugs discovered is too coarse-grained. Instead, we argue that it is important to distinguish between reaching, triggering, and detecting a bug. Consequently, Magma uses these three bug-centric performance metrics to evaluate fuzzers.

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. This is where coverage profiles fall short: simply covering the faulty code does not mean that the program is in the correct state to trigger the bug. Hence, a triggered bug 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.

Source-code instrumentation provides ground-truth knowledge and runtime feedback of reached and triggered bugs. Each bug is approximated by (a) the lines of code patched in response to a bug report, and (b) a boolean expression representing the bug’s trigger condition. Our source code instrumentation—the canary—reports (i) when the line of code is reached; and (ii) when the input satisfies the conditions for faulty behavior (i.e., triggers the bug). Section 5.4 discusses how we prevent canaries from leaking information to the SUT.

Finally, we also draw a distinction 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), other bug classes are not as easily observed. For example, resource exhaustion bugs are often detected long after the fault has manifested, either through a timeout or an out-of-memory error. Even more obscure are semantic bugs, whose malfunctions cannot be observed without a specification or reference. Consequently, various fuzzing techniques have been developed to target these bug classes (e.g., SlowFuzz [40] and NEZHA [39]). Such advancements in fuzzer techniques may benefit from an evaluation which includes the bug detection rate as another dimension for comparison.

4.5 Runtime Monitoring

Magma provides a runtime monitor that collects real-time statistics from the instrumented target program. This provides a mechanism for visualizing the fuzzer’s progress and its evolution over time, without complicating the instrumentation.

The runtime monitor collects data about reached and triggered bugs (Section 4.4). Because this data primarily relates to the fuzzer’s program exploration capabilities, we post-process the monitor’s output to study the fuzzer’s fault detection capabilities. This is achieved by replaying the crashing inputs (produced by the fuzzer) against the benchmark canaries to determine which bugs were triggered and hence detected. Importantly, it is possible that the fuzzer produces crashing inputs that do not correspond to any injected bug. If this occurs, the new bug is triaged and added to the benchmark for other fuzzers to discover.

5 DESIGN AND IMPLEMENTATION DECISIONS

Magma’s unapologetic focus on fuzzing (as opposed to being a general bug-detection benchmark) necessitates a number of key design and implementation choices, which we discuss here.

5.1 Forward-Porting

5.1.1 Forward-Porting vs. Back-Porting. In contrast to back-porting bugs to previous versions, forward-porting ensures that all known bugs are fixed, and that the reintroduced bugs will have ground-truth oracles. While it is possible that the new fixes and features in newer codebases may (re)introduce unknown bugs, forward-porting allows Magma to evolve with each published bug fix. Additionally, future code changes may render a forward-ported bug
obsolescent, or make its trigger conditions unsatisfiable. Without verification, forward-porting may inject bugs which cannot be triggered. We use fuzzing to reduce this possibility, reducing the cost of manually verifying injected bugs. A fuzzer-generated PoV demonstrates that the bug is triggerable. Bugs that are discovered this way are added to the list of verified bugs, helping the evaluation of other fuzzers. While this approach may skew Magma towards fuzzier-discoverable bugs, we argue that this is a nonissue: any newly-discovered PoV will update the benchmark, thus ensuring a fair and balanced bug distribution.

5.1.2 Manual Forward-Porting. All Magma bugs are manually introduced through human effort. This process involves: (i) searching for bug reports; (ii) identifying bugs that affect the core codebase; (iii) finding the relevant fix commits; (iv) recognizing the bug conditions from the fix commits; (v) collecting these conditions as a set of path constraints; (vi) modeling these path constraints as a boolean expression (the bug canary); and (vii) injecting these canaries to flag bugs at runtime. The complexity of this process led us to reject a wholly-automated approach; automating bug injection would likely result in an incomplete and error-prone technique, ultimately yielding fewer bugs of lower quality. Moreover, an automated approach still requires manual verification of the results. Dedicating human resources to the forward-porting process maximizes the correctness of Magma’s bugs.

To justify a manual approach, we enumerate the scopes (i.e., code blocks, functions, modules) spanned by each bug fix and use these scopes as a simplified measure of bug-porting complexity (scope measures for all bugs are given in Table A1). While a simple bug-supporting technique works well for fixes with a scope of one, the bug-porting technique must become more advanced as the number of scopes increases (e.g., it must handle interprocedural constraints). Of the 118 Magma bugs, 34% had a scope measure greater than one.

Finally, our manual porting process was heavily reliant on prose; in particular, by the comments and discussions contained within a bug report. These discussions provide valuable insight into (a) developers’ intent and (b) the construction of precise trigger conditions. Additionally, function names (particularly those from the standard library) provide key insight into the code’s objective, without requiring in-depth analysis into what each function does. An automated technique would require either: (i) an in-depth analysis of such functions, likely resulting in path explosion; or (ii) inference of bug conditions and function utilities via natural language processing (NLP). Both of these approaches are too complex to be included in the scope of Magma’s development and would likely require several years of research to be effective.

5.2 Weird States

When a fuzzer generates an input that triggers an undetected bug, and execution continues past this bug, the program transitions into an undefined state: a weird state [15]. Any information collected after transitioning to a weird state is unreliable. To address this issue, we allow the fuzzer to continue the execution trace, but only collect bug oracle data before and until the first bug is triggered (i.e., transition to a weird state). Oracles do not signify that a bug has been executed; they only indicate whether the conditions required to execute a bug are satisfied.

Listing 1 shows an example of the interplay between weird states. This example contains two bugs: an out-of-bounds write (deonated as bug 17) and a division-by-zero (bug 6). When tmp.len == 0, the condition for bug 17 on line 6 is not satisfied, and bug 9 is captured on line 8 and triggered on line 9 (resulting in a divide-by-zero error). When tmp.len > 16, bug 17 is captured on line 5 and triggered on line 6. Furthermore, tmp.len is overwritten in the struct by a non-zero value, and bug 9 is not triggered. However, bug 17 is triggered when tmp.len == 16, overwriting tmp.len in the struct with the NULL terminator and setting its value to 0 (on a Little-Endian system). This triggers bug 9, despite the input not explicitly specifying a zero-length str.

5.3 A Static Benchmark

Much like other widely-used performance benchmarks—e.g., SPEC CPU [50] and DaCapo [8]—Magma is a static benchmark that contains realistic workflows. These benchmarks assume that if the system-under-test (SUT) performs well on the benchmark’s workloads, then it will perform similarly on real workflows. While realistic, static benchmarks are susceptible to overfitting. Overfitting can occur if developers tweak the SUT to perform better on a benchmark, rather than focusing on real workflows.

Overfitting could be overcome by dynamically synthesizing a benchmark (and ensuring that the SUT is unaware of the synthesis parameters). However, this approach risks generating workflows different from real-world scenarios, rendering the evaluation biased and/or incomplete. While program synthesis is a long-studied topic [7, 22, 26], it remains difficult to generate large programs that remain faithful to real development patterns and styles.

To prevent overfitting, Magma’s forward-porting process allows targets to be updated as they evolve in the real-world. Each forward-ported bug requires minimal code changes: the addition of Magma’s instrumentation and the faulty code itself. This makes it relatively straightforward to update targets, including introducing new bugs and new features. For example, two undergraduate students without software security experience added over 60 bugs in three new targets over a single semester. These measures ensure that Magma remains representative of real, complex targets and suitable for fuzzer evaluation.
5.4 Leaky Oracles

Introducing oracles into the benchmark may leak information that interferes with a fuzzer’s exploration capability, potentially leading to overfitting (as discussed in Section 5.3). For example, if oracles were implemented as if statements, fuzzers that maximize branch coverage could detect the oracle’s branch and find an input that satisfies the branch condition.

One possible solution to this leaky oracle problem is to produce both instrumented and uninstrumented target binaries (with respect to Magma’s instrumentation, not any instrumentation that the fuzzer injects). The fuzzer’s input would be fed into both binaries, but the fuzzer would only collect the data it needs (e.g., coverage feedback) from the uninstrumented binary. The instrumented binary would then collect canary data and report it to the runtime monitor. This approach, however, introduces other challenges associated with duplicating the execution trace between two binaries (e.g., replicating the environment, maintaining synchronization between executions), which greatly complicates Magma’s implementation and introduces runtime overheads.

Instead, we use always-evaluate memory writes. First, an injected bug oracle evaluates a boolean expression representing the bug’s trigger condition. This typically involves a binary comparison operator, which most compilers (e.g., gcc, clang) translate into a pair of cmp and set instructions embedded into the execution path. The results of this evaluation are then shared with the runtime monitor (Section 4.5). This process is demonstrated in Listings 2 and 3.

Listing 2 shows Magma’s canary implementation. The always-evaluate memory accesses are shown in lines 4 and 5. The faulty flag addresses the problem of weird states (Section 5.2), and disables future canaries after the first bug is encountered.

Listing 3 shows an example program instrumented with a canary. A call to magma_log is inserted (line 3) prior to the execution of the faulty code (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 avoid leaking information through coverage, we provide custom x86-64 assembly blocks to evaluate these logical operators in a single basic block (without short-circuit behavior). We revert to C’s bitwise operators (& and |) which are more brittle and susceptible to safety-agnostic compiler passes [49]—when the compilation target is not x86-64.

Although this approach may introduce memory access patterns that are detectable by taint tracking and other data-flow analysis techniques, statistical tests can be used to infer whether the fuzzer overfits to these access patterns. By repeating the fuzzing campaign with the uninstrumented binary, we can verify if the results vary significantly.

5.5 Proofs of Vulnerability

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

When available, we extract PoVs from public bug reports. When no PoV is available, we launch multiple fuzzing campaigns against these targets in an attempt to trigger each injected bug. Inputs that trigger a bug are saved as a PoV. Bugs which are not triggered, even after multiple campaigns, are manually inspected to verify path reachability and satisfiability of trigger conditions.

5.6 Unknown Bugs

Because Magma uses real-world programs, it is possible that bugs exist for which no ground-truth is available (i.e., an oracle does not exist). A fuzzer might inadvertently trigger these bugs and (correctly) detect a fault. Due to the imperfections in automated deduplication techniques, these crashes are not included in Magma’s metrics. Instead, such crashes are used to improve Magma itself. The bug’s root cause can be determined by manually studying the execution trace, after which the bug can be added to the benchmark.

5.7 Fuzzer Compatibility

Fuzzers are not limited to a specific execution engine under which they analyze and explore a program. For example, some fuzzers (e.g., Driller [51], T-Fuzz [38]) leverage symbolic execution (using an engine such as angr [47]) to explore the target program. This can introduce (a) incompatibilities with Magma’s instrumentation, and (b) inconsistencies in the runtime environment (depending on how the symbolic execution engine models the environment).

However, the defining trait of most fuzzers, in contrast to other types of bug-finding tools, is that they concretely execute the target on the host system. Unlike benchmarks such as the CGC and BugBench—which aim to evaluate all bug-finding tools—Magma is unapologetically a fuzzing benchmark. This includes whitebox fuzzers that use symbolic execution to guide input generation, provided that the target is executed on the host system.

We therefore impose the following restriction on the fuzzers evaluated by Magma: the fuzzer must execute the target program in the context of an OS process, with unrestricted access to OS facilities (e.g., system calls, libraries, file system). This allows Magma’s runtime monitor to extract canary statistics using the operating system’s services at relatively low overhead/complexity.
6 EVALUATION

6.1 Methodology

We evaluated several fuzzers in order to establish the versatility of our metrics and benchmark suite. We chose a set of six mutational greybox fuzzers whose source code was available at the time of writing: AFL [58], AFLFast [10], AFL++ [25], FairFuzz [30], Moya-AFL [33], and honggfuzz [20]. These six fuzzers were evaluated over ten identical 24 h fuzzing campaigns for each fuzzer/target combination. This amounted to 200 000 CPU-hours of fuzzing.

Benchmark parameters were identical across all fuzzing campaigns to ensure fairness. Each fuzzer was bootstrapped with the same set of seed files sourced from the original target codebase and configured with a 50 ms timeout and 50 MiB memory limit. The Magma monitoring utility was configured to poll canary information every five seconds. All experiments were run on one of 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.

Notably, all six fuzzers implement fault detection as a simple crash/hang listener (Section 2.2.2). This makes it possible to leverage Magma’s fatal canaries mode (Section 4.1) to evaluate a fuzzer’s program exploration and fault detection capabilities. 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 such as NEZHA [39]), then their method must be used when evaluating the fuzzer’s 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 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.

AddressSanitizer (ASan) [46] is used to evaluate detected bugs. Crashing inputs (generated by fatal canaries) are replayed through the ASan-instrumented targets: the fuzzer can correctly detect the bug if the target hangs, exceeds its memory limit, or reports a crash. While ASan introduces 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 that the fuzzer is configured with strict fault-detection parameters and can thus detect more bugs. Although this may introduce false-positives during post-processing (e.g., a bug may be labeled as “detected”, whereas in fact the fuzzer may not have detected it during a campaign), it also skews results in favor of the evaluated fuzzers (because the fuzzers report more bugs). Although this evaluation method measures ASan’s fault-detection capabilities, it still highlights the bugs that fuzzers can realistically detect when fuzzing without ground truth.

6.2 Time to Bug

We use the time to find a bug as a measure of fuzzer performance. As discussed in Section 4.4, Magma records the time taken to both reach and trigger a bug, allowing us to compare fuzzer performance across multiple dimensions. Fuzzing campaigns are typically limited to a finite duration (we limit our campaigns to 24 h, repeated ten times), so it is important that the time-to-bug discovery is low.

The highly-stochastic nature of fuzzing means that the time to find a bug can vary wildly between identical trials. To account for this variation, we repeat each trial ten times. Despite this repition, a fuzzer may still fail to find a bug within the allotted time, leading to missing measurements. We therefore apply survival analysis to account for this missing data and high variation in bug discovery times. Specifically, we adopt Wagner’s approach [53] and use the Kaplan-Meier estimator [27] to model a bug’s survival function. This survival function describes the probability that a bug remains undiscovered (i.e., “survives”) within a given time (here, a 24 h trial). A smaller survival time indicates better fuzzer performance.

6.3 Experimental Results

Table 3, Figure 2, and Table A2 present the results of our fuzzing campaigns.
6.3.1 Bug Count and Statistical Significance. Table 3 shows the mean number of bugs found per fuzzer (across ten 24 h 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 2 shows the results of this analysis.

The Mann-Whitney U-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, the small fluctuations in mean bug 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 2 shows that this difference is statistically significant. While Table A2 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 [30] also displayed significant performance regression against libxml2, openssl, and php. While the original evaluation of FairFuzz claims that it achieved the highest coverage against xmlint, that improvement was not reflected in our results. Finally, honggfuzz performed significantly better than all other fuzzers in four out of seven targets, possibly because of its wrapping of memory-comparison functions.

6.3.2 Time to Bug. In total, 72 of the 118 Magma bugs (61 %) were reached. Additionally, 38 of the 46 verified bugs (83 %)—i.e., those with PoVs—were triggered. Notably, no single fuzzer triggered more than 33 bugs (72 % of the verified bugs). These results are presented in Table A2. Here, bugs are sorted by the mean trigger time, which we use to approximate “difficulty”.

The long bug discovery times (18 of the 38 triggered bugs—47 %—took on average more than 20 h to trigger) suggests that the evaluated fuzzers still have a long way to go in improving program exploration. However, while many of the Magma bugs are difficult to discover, Table A2 highlights a set of 17 “simple” bugs that all fuzzers find consistently within 24 h. These bugs provide a baseline for detecting performance regression: if a new fuzzer fails to discover these bugs, then its policy or implementation should be revisited.

Most of the bugs in Table A2 were reached by all fuzzers. AFL++ was the worst performing fuzzer in this regard, failing to reach five bugs (the highest amongst the six evaluated fuzzers). Interestingly, most bugs show a large difference between reach and trigger times. For example, only the first three bugs listed in Table A2 were triggered when first reached. In contrast, bugs such as MAE115 (from openssl) take 10 s to reach (by all fuzzers), but up to 20 h (on average) to trigger. This difference between time-to-reach and time-to-trigger a bug provides another feature for determining bug “difficulty”: while control flow may be trivially satisfied (as evidence by the time to reach a bug), bugs such as MAE115 may require complex, stateful data-flow constraints.

Figure 3 plots four survival functions for three Magma bugs (AAH018, JCH232, and AAH020). These plots illustrate the probability of a bug surviving a 24 h fuzzing trial, and are generated by applying the Kaplan-Meier estimate to the results of ten repeated fuzzing trials. Dotted lines represent survival functions for reached bugs, while solid lines represent survival functions for triggered bugs.

Figure 3a shows the time to reach bug AAH018 (libtiff). Notably, this bug was not triggered by any of the six evaluated fuzzers. Thus, the probability of bug AAH018 “surviving” 24 h (i.e., not being triggered) remains at one.

In comparison, Figure 3b shows the differences in the time taken to reach and trigger bug JCH232 (sqlite3). Here, honggfuzz is the best performer, because the bug’s probability of survival approaches zero the fastest. Notably, the variance is much higher compared to bug AAH018 (as evident by the larger confidence intervals).

Finally, Figure 3d and Figure 3c compare the probability of survival for bug AAH020 (libtiff) across two driver programs: tiffcp and read_rgba_fuzzer. The former is a general-purpose application, while the latter is a driver specifically designed as a fuzzer harness. While the bug is reached relatively quickly by both drivers, the fuzzer harness is clearly superior at triggering the bug, as it is faster across all fuzzers. This result supports our claim in ?? that domain experts are most suitable for selecting and developing fuzzing drivers.

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

6.3.3 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 specialities and begin to show performance improvements brought upon by the evaluated fuzzers.

6.3.4 Achilles’ Heel of Mutational Fuzzing. AAH001 (CVE-2018-13785, shown in Listing 4), 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).

6.3.5 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.

6.3.6 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.
6.4 Discussion

6.4.1 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. Inferring 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 FuzzBench only measures edges covered as a performance metric, dismissing fault-based metrics such as crash counts or bug counts. Edge coverage, however, is only an approximation of control-flow path coverage; relying on it as the only performance metric may result in biased evaluations, as our FairFuzz results highlight.

6.4.2 Ground Truth and Confidence. Ground truth enables us to determine a crash’s root cause. Unlike many existing benchmarks, Magma provides easy access to ground truth information. In particular, ground-truth knowledge is provided for all 118 injected bugs. Orthogonally, a bug’s PoV serves as a witness that demonstrates that a bug is triggerable. If fuzzer F does not trigger bug B with an existing PoV, then we can state that fuzzer F fails to trigger bug B; however, we cannot make this conclusion for bugs where no PoV exists. Importantly, only bugs with PoVs can be used to confidently measure a fuzzer’s performance. Currently, 39% of Magma’s bugs have a PoV. Regardless, bugs without a PoV remain useful; any fuzzer evaluated against Magma can produce a PoV, increasing the benchmark’s utility in the process. Widespread adoption of Magma will increase the percentage of bugs with PoVs.

6.4.3 Beyond Crashes. Although Magma’s instrumentation does not collect information about detected bugs—because detection is a characteristic of the fuzzer and not the bug itself—Magma enables the measurement of this metric through a post-processing step (supported by fatal canaries, discussed in Section 4.1). In particular, bugs are not restricted to crash-triggering faults. Some bugs result in resource starvation (e.g., unbounded loops or mallocs), 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 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 bug classes continue to evade state-of-the-art sanitization techniques (and to what degree).

6.4.4 Magma as a Lasting Benchmark. Magma leverages software with a long history of security bugs to build an extensible framework with ground truth knowledge. Like most benchmarks, the widespread 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, despite the fact that most of its bugs have been found, and that these bugs are of a single, synthetic type. Magma aims to provide an evaluation benchmark that incorporates realistic bugs in real software.

7 CONCLUSIONS

Magma is an open ground-truth fuzzing benchmark that enables accurate and consistent fuzzer evaluation and performance comparison. We designed and implemented Magma to provide researchers with a benchmark containing real targets with real bugs. Additionally, Magma contains ground-truth bug instrumentation which allows for real-time measurements of a fuzzer’s performance. After carefully selecting targets with a wide variety of applications, and a known history of security-critical bugs, we forward-ported 118 reported bugs and injected instrumentation that serves as ground-truth knowledge.

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 that are evaluated—ideally by the developers of those 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 open-source mutation-based greybox fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, and honggfuzz). Our evaluation shows that ground-truth enables accurate measurements of fuzzer performance. Our evaluation provides tangible insight on comparing 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] 2020. The Industrial Age of Hacking. In 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, 1129–1146. https://www.usenix.org/conference/usenixsecurity20/presentation/nosco

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer
## A Bugs and Reports

Table A1: The bugs injected into Magma, and the original bug reports. Of the 118 bugs, 78 bugs (66 %) have a scope measure of one. Although most single-scope bugs can be ported with an automatic technique, relying on such a technique would produce fewer and lower-quality canaries.

| Bug ID | Report | Class | PoV | Scopes |
|--------|--------|-------|-----|--------|
| AAH081 | CVE-2018-13785 | Integer overflow, divide by zero | ✓ | 1 |
| AAH082 | CVE-2019-7317 | Use-after-free | ✓ | 4 |
| AAH083 | CVE-2015-8472 | API inconsistency | 2 |
| AAH084 | CVE-2015-0973 | Integer overflow | ✗ | 1 |
| AAH085 | CVE-2014-9495 | Integer overflow, Buffer overflow | ✓ | 1 |
| AAH087 | Unspecified | Memory leak | 2 |
| AAH088 | CVE-2013-6994 | 0-pointer dereference | ✗ | 2 |
| AAH089 | CVE-2016-9535 | Heap buffer overflow | ✓ | 1 |
| AAH090 | CVE-2016-5314 | Heap buffer overflow | ✓ | 1 |
| AAH091 | CVE-2016-10266 | Divide by zero | ✗ | 2 |
| AAH092 | CVE-2016-10267 | Divide by zero | ✗ | 1 |
| AAH093 | CVE-2016-10269 | OOB read | ✓ | 1 |
| AAH094 | CVE-2016-10269 | OOB read | ✓ | 1 |
| AAH095 | CVE-2016-10270 | OOB read | ✓ | 4 |
| AAH096 | CVE-2015-8784 | Heap buffer overflow | ✓ | 1 |
| AAH097 | CVE-2017-7664 | 0-pointer dereference | 1 |
| AAH098 | CVE-2018-8950 | Heap buffer underflow | ✓ | 1 |
| AAH099 | CVE-2018-7456 | OOB read | ✓ | 1 |
| AAH100 | CVE-2018-3658 | Heap buffer overflow | ✓ | 2 |
| AAH101 | CVE-2018-15537 | OOB write | 2 |
| AAH102 | CVE-2017-11613 | Heap buffer underflow | ✗ | 2 |
| AAH104 | CVE-2017-9047 | Stack buffer overflow | ✓ | 2 |
| AAH105 | CVE-2017-6636 | Type confusion | ✓ | 1 |
| AAH106 | CVE-2017-7373 | XML external entity | ✓ | 1 |
| AAH107 | CVE-2018-14567 | Resource exhaustion | ✓ | 1 |
| AAH108 | CVE-2017-5130 | Integer overflow, heap corruption | ✗ | 1 |
| AAH109 | CVE-2017-9048 | Stack buffer overflow | ✗ | 2 |
| AAH110 | CVE-2017-8872 | OOB read | ✗ | 1 |
| AAH111 | ISSUE #58 (gitlab) | OOB read | ✗ | 1 |
| AAH112 | CVE-2015-8317 | OOB read | ✓ | 2 |
| AAH113 | CVE-2016-4449 | XML external entity | ✗ | 1 |
| AAH114 | CVE-2016-1834 | Heap buffer overflow | ✓ | 2 |
| AAH115 | CVE-2016-1836 | Use-after-free | ✗ | 2 |
| AAH116 | CVE-2018-1387 | Use-after-free | ✓ | 1 |
| AAH117 | CVE-2018-1838 | Heap buffer overread | ✓ | 2 |
| AAH118 | CVE-2018-1839 | Heap buffer overread | ✓ | 1 |
| AAH119 | CVE-2016-1840 | Heap buffer overflow | ✓ | 1 |
| AAH120 | CVE-2016-1762 | Heap buffer overread | ✓ | 1 |
| AAH121 | CVE-2018-14949 | Divide-by-zero | ✓ | 1 |
| AAH122 | CVE-2019-9959 | Resource exhaustion (memory) | ✓ | 1 |
| AAH123 | CVE-2017-8661 | Stack buffer overflow | ✓ | 1 |
| AAH124 | CVE-2019-10873 | 0-pointer dereference | ✗ | 2 |
| AAH125 | CVE-2019-12293 | Heap buffer underflow | ✗ | 1 |
| AAH126 | CVE-2019-10872 | Heap buffer overflow | ❌ | 3 |
| AAH127 | CVE-2019-9200 | Heap underwrite | ❌ | 1 |
| AAH128 | Bug #100861 | Divide-by-zero | ✗ | 1 |
| AAH129 | osafuzz-8499 | Integer overflow | ✓ | 1 |
| AAH130 | Bug #101366 | 0-pointer dereference | ✗ | 1 |
| JCH201 | CVE-2019-7310 | Heap buffer overflow | ✓ | 1 |
| JCH202 | CVE-2018-21009 | Integer overflow | ✗ | 1 |
| JCH203 | CVE-2018-20650 | Type confusion | ✗ | 1 |
| JCH204 | CVE-2018-20481 | 0-pointer dereference | ✗ | 1 |
| JCH205 | CVE-2018-19058 | Type confusion | ✗ | 2 |
| JCH206 | CVE-2018-13988 | OOB read | ✗ | 1 |
| JCH207 | CVE-2019-12360 | Stack buffer overflow | ✗ | 1 |
| JCH208 | CVE-2018-10768 | 0-pointer dereference | ✗ | 1 |
| JCH210 | CVE-2017-9776 | Integer overflow | ✗ | 1 |
| JCH211 | CVE-2017-18267 | Resource exhaustion (CPU) | ✗ | 1 |
| JCH212 | CVE-2017-14617 | Divide-by-zero | ✓ | 1 |
| JCH214 | CVE-2019-9936 | Heap buffer overflow | ✗ | 1 |
| JCH215 | CVE-2019-20218 | Stack buffer overflow | ✓ | 1 |
| JCH216 | CVE-2019-19923 | 0-pointer dereference | ✗ | 1 |
| JCH217 | CVE-2019-19959 | OOB read | ❌ | 1 |
| JCH218 | CVE-2019-19925 | 0-pointer dereference | ✗ | 1 |
| JCH219 | CVE-2019-19948 | OOB read | ✗ | 2 |
| JCH220 | CVE-2018-8740 | 0-pointer dereference | ✗ | 1 |
| JCH221 | CVE-2017-15286 | 0-pointer dereference | ✗ | 1 |
| JCH222 | CVE-2017-2520 | Heap buffer overflow | ✗ | 2 |
| JCH223 | CVE-2017-2518 | Use-after-free | ✗ | 1 |
| JCH225 | CVE-2017-10989 | Heap buffer overflow | ✗ | 1 |
| JCH226 | CVE-2019-19646 | Logical error | ✗ | 2 |
| JCH227 | CVE-2013-7443 | Heap buffer overflow | ✗ | 1 |
| JCH228 | CVE-2019-19926 | Logical error | ✗ | 1 |
| JCH229 | CVE-2019-19317 | Resource exhaustion (memory) | ✗ | 1 |
| JCH230 | CVE-2015-3415 | Double-free | ✗ | 1 |
| JCH231 | CVE-2020-9327 | 0-pointer dereference | ✗ | 3 |
| JCH232 | CVE-2015-3414 | Uninitialized memory access | ✗ | 1 |
| JCH233 | CVE-2015-3416 | Stack buffer overflow | ✗ | 1 |
| JCH234 | CVE-2019-19880 | 0-pointer dereference | ✗ | 1 |
| JCH235 | CVE-2019-9021 | Heap buffer overflow | ✗ | 1 |
| JCH236 | CVE-2019-9641 | Uninitialized memory access | ✗ | 1 |
| JCH237 | CVE-2019-11034 | OOB read | ✗ | 1 |
| JCH238 | CVE-2019-11039 | OOB read | ✗ | 1 |
| JCH239 | CVE-2019-11040 | Heap buffer overflow | ✗ | 1 |
| JCH240 | CVE-2018-20783 | OOB read | ✗ | 3 |
| JCH241 | CVE-2019-9022 | OOB read | ✗ | 2 |
| JCH242 | CVE-2019-9638 | Uninitialized memory access | ✗ | 1 |
| JCH243 | CVE-2019-9648 | OOB read | ✗ | 2 |
| JCH244 | CVE-2018-14883 | Heap buffer overflow | ✗ | 2 |
| JCH245 | CVE-2018-7584 | Stack buffer underread | ✗ | 1 |
| JCH246 | CVE-2017-11362 | Stack buffer overflow | ✗ | 1 |
| JCH247 | CVE-2019-9912 | Use-after-free | ✗ | 1 |
| JCH248 | CVE-2016-10159 | Integer overflow | ✗ | 2 |
| JCH249 | CVE-2016-7432 | OOB read | ✗ | 2 |
| Bug ID     | h fuzz R | all R | moptafl R | allfast R | all++ R | fairfuzz R | Mean R |
|------------|----------|-------|-----------|-----------|---------|-------------|--------|
| AAH003     | 10.00s   | 11.00s| 5.00s     | 10.00s    | 5.00s   | 10.00s      | 5.00s   |
| AAH037     | 10.00s   | 10.00s| 5.00s     | 15.00s    | 5.00s   | 16.00s      | 5.00s   |
| AAH041     | 10.00s   | 10.00s| 10.00s    | 15.00s    | 10.00s  | 16.00s      | 10.00s  |
| JCH207     | 5.00s    | 4.42m | 5.00s     | 2.05m     | 5.00s   | 2.93m       | 5.00s   |
| AAH056     | 10.00s   | 19.38m| 10.00s    | 17.17m    | 10.00s  | 14.58m      | 10.00s  |
| MAE016     | 5.00s    | 10.00s| 5.00s     | 3.97m     | 5.00s   | 4.42m       | 5.00s   |
| MAE026     | 10.00s   | 9.73m | 10.00s    | 1.14m     | 10.00s  | 1.55m       | 10.00s  |
| AAH015     | 25.50s   | 2.83m | 15.00m    | 6.01h     | 15.43m  | 5.28h       | 15.24m  |
| MAB014     | 10.00s   | 4.11h | 10.00s    | 5.58m     | 10.00s  | 8.18m       | 10.00s  |
| AAH052     | 15.00s   | 38.27m| 10.00s    | 3.94h     | 10.00s  | 2.18h       | 10.00s  |
| AAH032     | 5.00s    | 4.28m | 10.00s    | 14.45h    | 10.00s  | 24.00h      | 10.00s  |
| JCH215     | 15.00s   | 40.97m| 2.37h     | 15.67h    | 14.63m  | 10.74h      | 48.87m  |
| AAH020     | 5.00s    | 5.65h | 5.00s     | 13.89h    | 5.00s   | 14.88h      | 5.00s   |
| AAH022     | 26.00s   | 1.66h | 15.00m    | 15.70h    | 15.43m  | 18.12h      | 15.24m  |
| JCH232     | 43.86m   | 1.66h | 19.82h    | 19.82h    | 17.94h  | 17.94h      | 14.93h  |
| AAH055     | 10.00s   | 40.23m| 10.00s    | 13.61h    | 10.00s  | 23.66h      | 10.00s  |
| AAH017     | 22.32h   | 22.32h| 19.84h    | 19.84h    | 12.88h  | 12.88h      | 8.67h   |
| JCH201     | 10.00s   | 24.00h| 10.00s    | 16.82h    | 10.00s  | 12.54h      | 10.00s  |
| AAH008     | 10.00s   | 3.65h | 10.00s    | 19.44h    | 10.00s  | 24.00h      | 10.00s  |
| AAH014     | 24.00h   | 24.00h| 6.34h     | 6.34h     | 17.47h  | 17.47h      | 24.00h  |
| AAH007     | 5.00s    | 57.00s| 10.00s    | 24.00h    | 10.00s  | 24.00h      | 10.00s  |
| AAH045     | 13.50s   | 1.13h | 15.00s    | 24.00h    | 15.00s  | 24.00h      | 15.00s  |
| AAH013     | 4.05h    | 4.05h | 24.00h    | 24.00h    | 24.00h  | 24.00h      | 24.00h  |
| MAB115     | 10.00s   | 20.96h| 10.00s    | 21.32h    | 10.00s  | 18.11h      | 10.00s  |
| AAH026     | 10.00s   | 7.00h | 10.00s    | 24.00h    | 10.00s  | 24.00h      | 10.00s  |
| AAH024     | 15.00s   | 9.27h | 10.00s    | 24.00h    | 10.00s  | 24.00h      | 10.00s  |
| JCH226     | 4.09h    | 10.93h| 24.00h    | 24.00h    | 24.00h  | 24.00h      | 24.00h  |
| JCH228     | 2.47h    | 20.05h| 22.57h    | 22.60h    | 17.14h  | 19.87h      | 24.00h  |
| MAB104     | 10.00s   | 24.00h| 10.00s    | 21.81h    | 10.00s  | 24.00h      | 10.00s  |
| AAH001     | 10.00s   | 17.70h| 10.00s    | 22.60h    | 10.00s  | 24.00h      | 10.00s  |
| JCH212     | 15.00s   | 20.42h| 15.00s    | 24.00h    | 15.00s  | 24.00h      | 15.00s  |
| AAH053     | 35.00s   | 21.80h| 30.00s    | 24.00h    | 29.50s  | 24.00h      | 29.50s  |
| AAH025     | 22.48h   | 22.48h| 24.00h    | 24.00h    | 24.00h  | 24.00h      | 24.00h  |
| AAH048     | 10.00s   | 22.72h| 15.00s    | 24.00h    | 10.00s  | 24.00h      | 10.00s  |
| AAH050     | 16.80h   | 23.71h| 24.00h    | 24.00h    | 20.00s  | 24.00h      | 20.00s  |
| AAH016     | 24.00h   | 24.00h| 24.00h    | 24.00h    | 24.00h  | 24.00h      | 24.00h  |
| AAH054     | 5.00s    | 24.00h| 5.00s     | 24.00h    | 5.00s   | 24.00h      | 5.00s   |
| JCH202     | 10.00s   | 24.00h| 10.00s    | 24.00h    | 10.00s  | 24.00h      | 10.00s  |
### Table A2: Mean bug survival times (cont.). None of these bugs were triggered by the six evaluated fuzzers.

| Bug ID | hfuzz | all | moptafl | aflfast | afl++ | fairfuzz | Mean |
|--------|-------|-----|---------|---------|-------|----------|-------|
|        | R     | T   | R       | T       | R     | T        | R     | T   |
| AAH005 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH004 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH011 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| MAE006 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| MAE004 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| MAE114 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| MAE105 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| MAE111 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH034 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH029 | 15.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH035 | 10.00s| 24.00h| 10.00s  | 24.00h  | 10.00s| 24.00h   | 10.00s| 24.00h|
| AAH059 | 10.00s| 24.00h| 15.00s  | 24.00h  | 15.00s| 24.00h   | 10.00s| 24.00h|
| JCH204 | 23.50s| 24.00h| 15.00s  | 24.00h  | 15.00s| 24.00h   | 10.00s| 24.00h|
| AAH049 | 15.00s| 24.00h| 15.00s  | 24.00h  | 15.00s| 24.00h   | 10.00s| 24.00h|
| AAH031 | 15.00s| 24.00h| 15.00s  | 24.00h  | 15.00s| 24.00h   | 15.00s| 24.00h|
| MAE103 | 28.00s| 24.00h| 27.50s  | 24.00h  | 25.00s| 24.00h   | 20.00s| 24.00h|
| JCH210 | 24.50s| 24.00h| 30.00s  | 24.00h  | 25.00s| 24.00h   | 28.00s| 24.00h|
| JCH214 | 1.00m | 24.00h| 31.00s  | 24.00h  | 31.00s| 24.00h   | 25.00s| 24.00h|
| AAH042 | 25.00s| 24.00h| 43.50s  | 24.00h  | 43.50s| 24.00h   | 33.50s| 24.00h|
| AAH051 | 30.50s| 24.00h| 19.32m  | 24.00h  | 12.72m| 24.00h   | 14.16m| 24.00h|
| JCH220 | 11.50s| 24.00h| 2.09h   | 24.00h  | 11.50s| 24.00h   | 9.37m | 24.00h|
| JCH229 | 16.00s| 24.00h| 2.80h   | 24.00h  | 7.44s | 24.00h   | 1.07h | 24.00h|
| AAH047 | 16.80h| 24.00h| 25.00s  | 24.00h  | 20.00s| 24.00h   | 20.00s| 24.00h|
| AAH043 | 16.80h| 24.00h| 25.00s  | 24.00h  | 20.00s| 24.00h   | 20.00s| 24.00h|
| JCH230 | 22.50s| 24.00h| 3.31h   | 24.00h  | 0.88s | 24.00h   | 1.36h | 24.00h|
| JCH223 | 30.50s| 24.00h| 3.89h   | 24.00h  | 8.34s | 24.00h   | 1.33h | 24.00h|
| JCH231 | 36.00s| 24.00h| 3.96h   | 24.00h  | 8.35s | 24.00h   | 1.41h | 24.00h|
| JCH233 | 12.02m| 24.00h| 3.87h   | 24.00h  | 8.98s | 24.00h   | 1.98h | 24.00h|
| AAH018 | 1.12h | 24.00h| 12.45h  | 24.00h  | 12.68s| 24.00h   | 7.71h | 24.00h|
| JCH222 | 21.97m| 24.00h| 15.17h  | 24.00h  | 24.00s| 24.00h   | 13.39h| 24.00h|
| AAH010 | 24.00h| 24.00h| 6.34h   | 24.00h  | 17.47h| 24.00h   | 24.00h| 24.00h|
| AAH009 | 23.46h| 24.00h| 24.00h  | 24.00h  | 21.70s| 24.00h   | 24.00h| 24.00h|
| JCH227 | 20.58s| 24.00h| 24.00h  | 24.00h  | 24.00s| 24.00h   | 24.00h| 24.00h|
| JCH219 | 23.22h| 24.00h| 24.00h  | 24.00h  | 23.79h| 24.00h   | 24.00h| 24.00h|

Magma: A Ground-Truth Fuzzing Benchmark