Magma: A Ground-Truth Fuzzing Benchmark

AHMAD HAZIMEH, EPFL, Switzerland
ADRIAN HERRERA, ANU & DST, Australia
MATHIAS PAYER, EPFL, Switzerland

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 seven widely-used mutation-based fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, honggfuzz, and SymCC-AFL) 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 fuzzer performance evaluation, highlighting the importance of ground truth in performing more accurate and meaningful evaluations.

CCS Concepts: • **General and reference** → **Metrics**; **Evaluation**; • **Software and its engineering** → **Software defect analysis**; • **Security and privacy** → *Software and application security*;

Keywords: fuzzing; benchmark; software security; performance evaluation

ACM Reference Format:
Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. In *Proc. ACM Meas. Anal. Comput. Syst.*, Vol. 4, 3, Article 49 (December 2020). ACM, New York, NY. 29 pages. https://doi.org/10.1145/3428334

1 INTRODUCTION

Fuzz testing ("fuzzing") is a widely-used dynamic bug discovery technique. A fuzzer procedurally generates inputs and subjects the target program (the "target") 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). 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

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868).
Authors' addresses: Ahmad Hazimeh, EPFL, Switzerland, ahmad.hazimeh@epfl.ch; Adrian Herrera, ANU & DST, Australia, adrian.herrera@anu.edu.au; Mathias Payer, EPFL, Switzerland, mathias.payer@nebelwelt.net.

2476-1249/2020/12-ART49 $15.00
https://doi.org/10.1145/3428334

crash does not necessarily mean that the program is bug-free (incompleteness). Fuzzing is wildly successful in finding bugs in open-source [2] and commercial off-the-shelf [4, 5, 51] software.

The success of fuzzing has resulted in an explosion of new techniques claiming to improve bug-finding performance [38]. In order to highlight improvements, these techniques are typically evaluated across a range of metrics, including: (i) crash counts; (ii) ground-truth bug counts; and/or (iii) code-coverage profiles. 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 targets that these metrics are evaluated on can vary wildly across papers, making cross-fuzzer comparisons impossible. Each of these metrics has particular deficiencies.

Crash counts. The simplest fuzzer evaluation method is to count the number of crashes triggered by a fuzzer, and compare this crash count with that achieved by another fuzzer (on the same target). Unfortunately, crash counts often inflate the number of actual bugs in the target [30]. Moreover, deduplication techniques (e.g., coverage profiles, stack hashes) fail to accurately identify the root cause of these crashes [8, 30].

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 [41].

Code-coverage profiles. 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 [30] 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 rethinking 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 targets 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 fuzzer 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 programs (in total, 118 bugs were analyzed and reinserted). Additionally, each reinserted bug is accompanied by a light-weight *oracle* that detects and reports if the bug is *reached* or *triggered*. This distinction between reaching and triggering a bug—in addition to a fuzzer's ability to *detect* 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 for a fuzzer benchmark that allow for a fair and accurate evaluation and comparison of fuzzers.
- A quantitative comparison of existing fuzzer benchmarks.
- The design and implementation of Magma, a ground-truth fuzzing benchmark based on real programs with real bugs.
- An evaluation of Magma against seven widely-used fuzzers.

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 Fuzz testing (fuzzing)

A fuzzer is a dynamic testing tool that discovers software flaws by running a target program (the "target") with a large number of automatically-generated inputs. Importantly, these inputs are generated with the intention of triggering a crash in the target. This input generation process is dependent on the fuzzer's knowledge of the target's *input format* and *program structure*. For example, *grammar-based* fuzzers (e.g., Superion [63], Peachfuzz [42], and QuickFuzz [22]) leverage the target'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 [66], 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 classified by their knowledge of the target's program structure. For example, *whitebox* fuzzers [17, 18, 47] leverage program analysis to infer knowledge about the program structure. In comparison, *blackbox* fuzzers [3, 64] blindly generate inputs in the hope of discovering a crash. Finally, *greybox* fuzzers [12, 34, 66] 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 randomness is independent 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's internals (blackbox fuzzing), or uses sophisticated program analyses to understand the target (whitebox fuzzing). The stochastic nature of fuzzing 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 [38] 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 [30]. 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. This software should be both diverse and realistic so that a practitioner has confidence that the fuzzer will perform similarly in real-world environments.

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.

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 targets.

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

The LAVA-M [14] test suite (built on top of `coreutils-8.24`) 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. We quantify these observations in Section 6.3.6.

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 [35] and the Google Fuzzer Test Suite (FTS) [20] both contain real programs with real bugs. However, each target 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 [19]—the successor to the Google FTS—is a fuzzer evaluation platform that relies solely on coverage profiles as a performance metric. As previously discussed, this metric has limited utility when evaluating fuzzers on their bug-finding capability. UniFuzz [33]—which was developed concurrently but independently from Magma—is similarly built on real programs containing real bugs. However, it lacks ground-truth knowledge and it is unclear how many bugs each target contains. Not knowing how many bugs exist in a benchmark makes fuzzer comparisons challenging.

Finally, popular open-source software (OSS) is often used to evaluate fuzzers [10, 30, 31, 37, 44, 62]. 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. First, it is

Table 1. Summary of existing fuzzer benchmarks and our benchmark, Magma. We characterize benchmarks across two dimensions: the targets 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 **R**eal or **S**ynthetic. Bug density is the mean number of bugs per workload. Finally, ground truth may be available (✓), available but not easily accessible (◗), or unavailable (✗).

Benchmark	Workloads #	Workloads Real/Synthetic	Bugs #	Bugs Real/Synthetic	Bug Density	Ground truth
BugBench [35]	17	R	19	R	1.12	◗
CGC [11]	131	S	590	S	4.50	◗
Google FTS [20]	24	R	47	R	1.96	◗
Google FuzzBench [19]	21	R	–	–	–	–
LAVA-M [14]	4	R	2265	S	566.25	✓
UniFuzz [33]	20	R	?	R	?	✗
Open-source software	–	R	?	R	?	✗
Magma	7	R	118	R	16.86	✓

often unclear which software version is used, making fair cross-paper comparisons impossible. Second, multiple software versions introduce *version divergence*, a subtle evaluation flaw shared by both crash and bug count metrics. After running for an extended period, a fuzzer's ability to discover new bugs diminishes over time [9]. If a second fuzzer later fuzzes a new version of the same program—with the bugs found by the first fuzzer appropriately patched—then the first fuzzer will find fewer bugs in this newer version. Version divergence is also inherent in UniFuzz, which builds on top of older versions of OSS.

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 [55]).

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) [8, 30]. Instead, the number of bugs found by each fuzzer should 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 bug classes 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 [7]. Several factors must be taken into account when designing a benchmark, including: relevance; reproducibility; fairness; verifiability; and usability [1, 60]. While building benchmarks around these properties is well studied [1, 7, 24, 29, 35, 50, 52, 57, 60], 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*" [60]. However, individual fuzzing trials vary wildly in performance, requiring a large number of repeated trials for a particular test environment [30]. While performance variance exists in most benchmarks (e.g., the SPEC CPU benchmark [57] uses the median of three repeated trials to account for small variations across environments), this variance is more pronounced in fuzzing. Furthermore, a fuzzer may actively modify the test environment (e.g., T-Fuzz [44] and FuzzGen [26] transform the target, while Skyfire [62] generates new seed inputs for the target). This is very different to traditional performance benchmarks (e.g., SPEC CPU [57], DaCapo [7]), where the workloads and their inputs remain fixed across all systems-under-test. 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 remainder of this section, 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 [2, 4, 5, 51]. 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) [40] 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, targets (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 the diversity 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 consider bugs as an evaluation metric. BugBench primarily focuses on memory corruption vulnerabilities, but also contains uninitialized read, memory leak, data race, atomicity, and semantic bugs (totalling nine bug classes). Conversely, Google FTS and FuzzBench satisfy the target diversity requirement: both contain workloads from a wide variety of application domains (e.g., cryptography, image parsing, text processing, and compilers).

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, a 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 fuzzer-generated 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 control- and data-flow analysis tools. Each fuzzer may introduce its own instrumentation into a target (e.g., AFL [66]), run the target in a specific execution engine (e.g., QSYM [65], Driller [58]), or provide inputs through a specific channel (e.g., libFuzzer [34]). 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**).

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* (see Section 4.2). 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 (see Section 4.3). 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 4.4).

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, allowing us to determine whether a bug is *reached*. The fuzzer-generated input *triggers* a bug when the input's dataflow satisfies the bug's trigger condition(s). 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 4.3.

Finally, Magma provides a *fatal canaries* mode. In fatal canaries mode, the program is terminated if a canary's condition is satisfied (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.

4.1 Target Selection

Magma contains seven targets, which we summarize in Table 2. In addition to these seven *targets* (i.e., the codebases into which bugs are injected), Magma also includes 25 *drivers* (i.e., executable programs that provide a command-line interface to the target) that exercise different functionality within the target. Inspired by Google OSS-Fuzz [2], 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 qualitatively in Table 2). Inspired by benchmarks in other fields [7, 27, 48, 50], we apply *Principal Component Analysis* (PCA) to quantify this diversity. PCA is a statistical analysis technique that transforms an N-dimensional space into a lower-dimensional space while preserving variance as much as possible [43]. Reducing high-dimensional data into a set of *principal components* allows for the application of visualization and/or clustering techniques to compare and discriminate benchmark workloads.

We apply PCA as follows. First, we use an Intel Pin [36] tool to record instruction traces for $K = 284$ *subjects* (i.e., a library wrapped with a particular driver program [34, 39]): four from

Table 2. The targets, driver programs, bug counts, and evaluated features incorporated into Magma. The versions used are the latest at the time of writing.

Target	Drivers	Version	File type	Bugs	Magic values	Recursive parsing	Compression	Checksums	Global state
libpng	`read_fuzzer`, `readpng`	1.6.38	PNG	7	✓	✗	✓	✓	✗
libtiff	`read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF	14	✓	✗	✓	✗	✗
libxml2	`read_memory_fuzzer`, `xml_reader_for_file_fuzz` `xmllint`	2.9.10	XML	18	✓	✓	✗	✗	✗
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF	22	✓	✓	✓	✓	✗
openssl	`asn1`, `asn1parse`, `bignum`, `bndiv`, `client`, `cms`, `conf`, `crl`, `ct`, `server`, `x509`	3.0.0	*Binary blobs*	21	✓	✗	✓	✓	✓
sqlite3	`sqlite3_fuzz`	3.32.0	SQL queries	20	✓	✓	✗	✗	✓
php	`exif`, `json`, `parser`, `unserialize`	8.0.0-dev	*Various*	16	✓	✓	✗	✗	✗

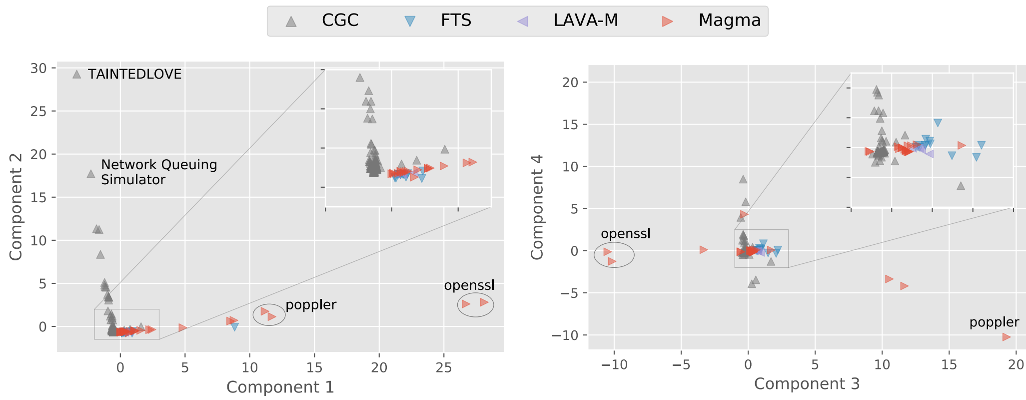


Fig. 1. Scatter plots of benchmark scores over the first four principal components (which account for ~60 % of the variance in the benchmark workloads). Each point corresponds to a particular subject in a benchmark.

LAVA-M, 14 from the FTS, 25 from Magma, and 241 from the CGC [59]. Each trace is driven by seeds provided by the benchmark (exercising functionality—and hence code—that would be explored by a fuzzer) and contains instructions executed by both the subject and any linked libraries. Second, instructions are categorized according to Intel XED, a disassembler built into Pin. A XED instruction category is "*a higher level semantic description of an instruction than its opcodes*" [25]. XED contains $N = 94$ instruction categories, spanning logical, floating point, syscall, and SIMD operations (amongst others). We use these categories as an approximation of the subject's functionality. Third, we create a matrix X, where $x_{ij} \in X$ ($i \in [1, N]$ and $j \in [1, K]$) is the mean number of instructions executed in a particular category for a given subject (over all seeds supplied with that subject). Finally, PCA is performed on a normalized version of X. The first four principal components, which in our case account for 60 % of the variance between benchmarks, are plotted in a two-dimensional space in Figure 1.

Figure 1 shows that the four LAVA-M workloads are tightly clustered over the first four principal components. This is unsurprising, given that the LAVA-M workloads are all sourced from coreutils and hence share the same codebase. In contrast, both the CGC and Magma provide a wide-variety of workloads. For example, *openssl*—which contains a large amount of cryptographic and networking code—appears distinct from the main clusters in Figure 1. The CGC's *TAINTEDLOVE* workload is similarly distinct, due to the relatively large number of floating point operations performed.

4.2 Bug Selection and Insertion

Magma contains 118 bugs, spanning 11 CWEs (summarized in Figure 2; the complete list of bugs is given in Table A1). Compared to existing benchmarks, Magma has both the second-largest variety of bugs (by CWE) and second-largest "bug density" (the ratio of the number of bugs to the number of targets) after the CGC and LAVA-M, respectively. While the CGC has a wider variety of bugs, its workloads are not indicative of real-world software (in terms of both size and complexity). Similarly, while LAVA-M's bug density (566.25 bugs per target) is an order-of-magnitude larger than Magma's (16.86 bugs per target), LAVA-M is restricted to a single, synthetic bug type.

Importantly, 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

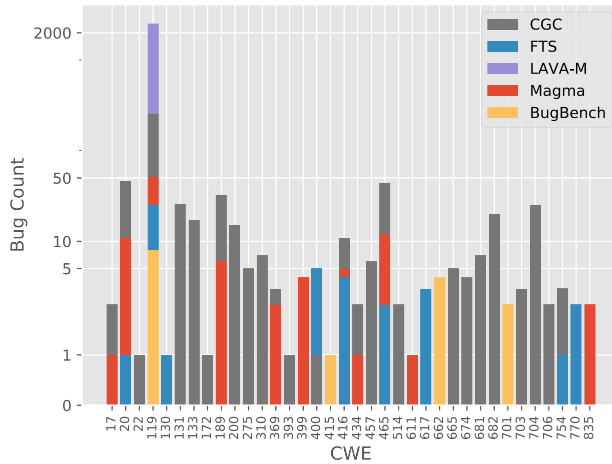


Fig. 2. Comparison of benchmark bug classes. The y-axis uses a log scale. A complete list of Magma bugs is presented in Table A1.

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 a light-weight 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.

4.3 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 (i.e., the canary) 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. 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 system-under-test.

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 [46] and NEZHA [45]). Such advancements in fuzzer techniques may benefit from an evaluation which includes the bug *detection* rate as another dimension for comparison.

4.4 Runtime Monitoring

Magma provides a runtime monitor that collects real-time statistics from the instrumented target. 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 4.3). 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. We discuss these choices 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 obsolete, 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 fuzzer-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. 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 measure of bug-porting complexity (scope measures for all bugs are given in Table A1). While a simple bug-porting 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 bug reports. 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 (bug 1) and a division-by-zero (bug 2). When `tmp.len == 0`, the condition for bug 1 (line 6) remains unsatisfied, logging and triggering bug 2 instead (lines 8 and 9, respectively). However, when `tmp.len > 16`, bug 1 is logged and triggered (lines 5 and 6,

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

Listing 1. Weird states can result in execution traces which do not exist in the context of normal program behavior.

respectively). Furthermore, `tmp.len` is overwritten by a non-zero value, leaving bug 2 untriggered. In contrast, bug 1 is triggered when `tmp.len == 16`, overwriting `tmp.len` with the `NULL` terminator and setting its value to 0 (on a Little-Endian system). This also triggers bug 2, 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 [57] and DaCapo [7]—Magma is a *static* benchmark that contains realistic workloads. These benchmarks assume that if the system-under-test performs well on the benchmark's workloads, then it will perform similarly on real workloads. While realistic, static benchmarks are susceptible to *overfitting*. Overfitting can occur if developers tweak the system-under-test to perform better on a benchmark, rather than focusing on real workloads.

Overfitting could be overcome by *dynamically synthesizing* a benchmark (and ensuring that the system-under-test is unaware of the synthesis parameters). However, this approach risks generating workloads different from real-world scenarios, rendering the evaluation biased and/or incomplete. While program synthesis is a well-studied topic [6, 23, 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 hence generate an input that satisifies 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 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), greatly complicating Magma's implementation and introducing runtime overheads.

Instead, we use *always-evaluate memory writes*, whereby 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 4.4). This process is demonstrated in Listings 2 and 3.

Listing 2 shows Magma's canary implementation. The always-evaluated memory accesses are shown on 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

```c
void magma_log(int id, bool condition) {
  extern struct magma_bug *bugs; // = mmap(...)
  extern bool faulty; // = false initially
  bugs[id].reached   += 1         & (faulty ^ 1);
  bugs[id].triggered += condition & (faulty ^ 1);
  faulty = faulty | condition;
}
```

Listing 2. Magma instrumentation.

```c
void libfoo_bar() {
  // uint32_t a, b, c;
  magma_log(42, (a == 0) | (b == 0));
  // possible divide-by-zero
  uint32_t x = c / (a * b);
}
```

Listing 3. Instrumented example.

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 [56]—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 inadvertantly 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 [58], T-Fuzz [44]) leverage symbolic execution (using an engine such as angr [54]) to explore the target. 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 (SymCC [49] is one such fuzzer that we include in our evaluation).

We therefore impose the following restriction on the fuzzers evaluated by Magma: the fuzzer must execute the target 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 seven *mutational fuzzers* whose source code was available at the time of writing: AFL [66], AFLFast [10], AFL++ [16], FairFuzz [31], MOpt-AFL [37], honggfuzz [21], and SymCC-AFL [49]. These seven fuzzers were evaluated over ten identical 24 h and 7 d fuzzing campaigns for each fuzzer/target combination. This amounts to 200,000 CPU-hours of fuzzing.

To ensure fairness, benchmark parameters were identical across all fuzzing campaigns. Each fuzzer was bootstrapped with the same set of seed files (sourced from the original target codebase) and configured with the same timeout and memory limits. Magma's monitoring utility was configured to poll canary information every five seconds, and *fatal canaries* mode (Section 4) was used to evaluate a fuzzer's ability to *reach* and *trigger* bugs. 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.

AddressSanitizer (ASan) [53] was used to evaluate *detected* bugs. Crashing inputs (generated by fatal canaries) were validated by replaying them through the ASan-instrumented target. 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 required to find a bug as a measure of fuzzer performance. As discussed in Section 4.3, 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 and 7 d, 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-bug can vary wildly between identical trials. To account for this variation, we repeat each trial ten times. Despite this repetition, a fuzzer may still fail to find a bug within the alloted 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 [61] and use the Kaplan-Meier estimator [28] 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, 24 h and 7 d trials). A smaller survival time indicates better fuzzer performance.

6.3 Experimental Results

Figure 3, Figure 4, Table A2, and Table A3 present the results of our fuzzing campaigns.

6.3.1 Bug Count and Statistical Significance. Figure 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

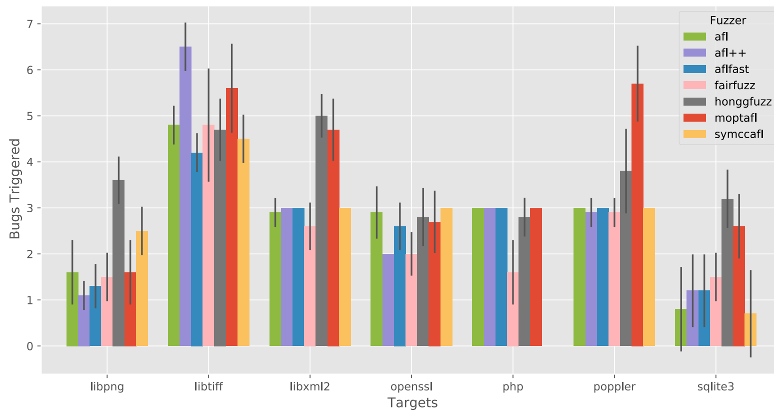


Fig. 3. The mean number of bugs (and standard deviation) found by each fuzzer across ten 24 h campaigns.

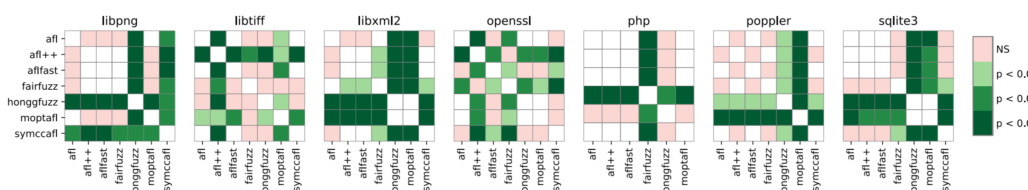


Fig. 4. Significance of evaluations of fuzzer pairs using p-values from the Mann-Whitney U-Test. We use $p < 0.05$ as a threshold for significance. Values greater than 0.05 are shaded red. Darker shading indicates a lower p-value, or higher statistical significance. White cells indicate that the pair of sample sets are identical.

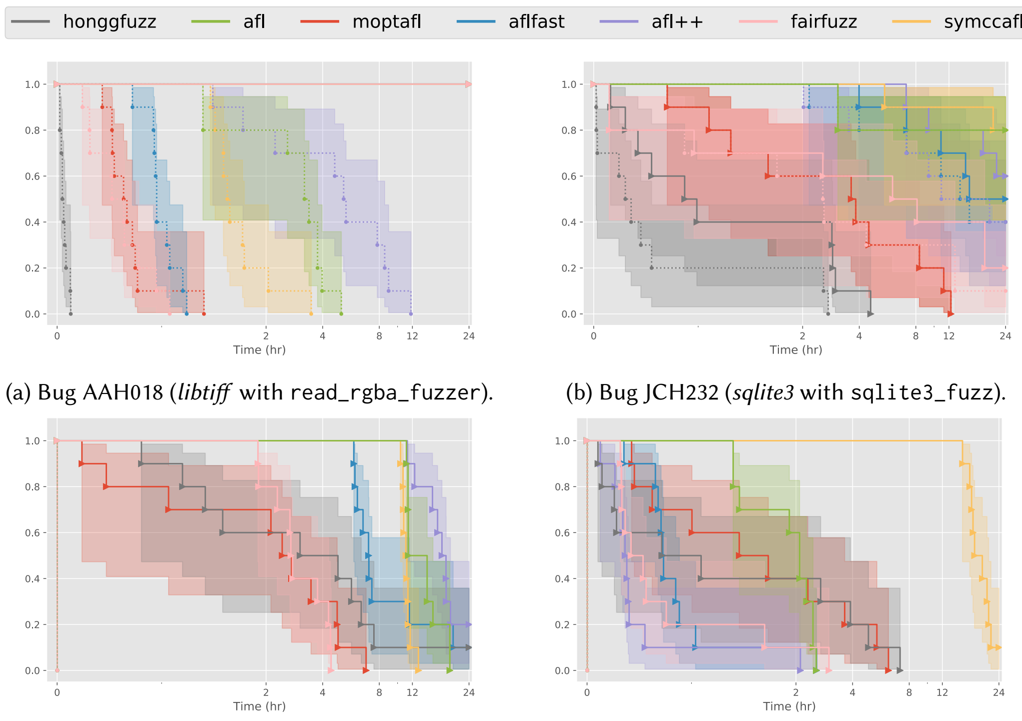
fuzzers), we make no assumptions about their distributions. Hence, we apply the Mann-Whitney U-test to measure statistical significance. Figure 4 shows the results of this analysis.

The Mann-Whitney U-test shows that AFL, AFLFast, AFL++, and SymCC-AFL performed similarly against most targets (signified by the large number of red and white cells in Figure 4), despite some minor differences in mean bug counts (shown in Figure 3). Figure 4 shows 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*. Figure 3 reveals that AFL++ scored the highest mean bug count compared to all other fuzzers, and Figure 4 shows that this difference is statistically significant.

On the other hand, FairFuzz [31] displayed significant performance regression against *libxml2*, *openssl*, and *php*. While the original evaluation of FairFuzz claims that it achieved the highest coverage against `xmllint`, that improvement was not reflected in our results.

Finally, honggfuzz and MOpt-AFL performed significantly better than all other fuzzers in three out of seven targets. Additionally, honggfuzz was the best fuzzer for *libpng* as well. We attribute honggfuzz's performance to its wrapping of memory-comparison functions, which provides comparison progress information to the fuzzer (similar to Steelix [32]).

6.3.2 Time to Bug. In total, during the 24 h campaigns, 74 of the 118 Magma bugs (62 %) were reached. Additionally, 43 of the 54 *verified* bugs (79 %)—i.e., those with PoVs—were triggered. Notably, no single fuzzer triggered more than 37 bugs (68 % 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".



(a) Bug AAH018 (*libtiff* with `read_rgba_fuzzer`). (b) Bug JCH232 (*sqlite3* with `sqlite3_fuzz`).

(c) Bug AAH020 (*libtiff* with `tiffcp`). (d) Bug AAH020 (*libtiff* with `read_rgba_fuzzer`).

Fig. 5. Survival functions for a subset of Magma bugs. The y-axis is the *survival probability* for the given bug. Dotted lines represent survival functions for *reached* bugs, while solid lines represent survival functions for *triggered* bugs. Confidence intervals are shown as shaded regions.

The long bug discovery times (19 of the 43 triggered bugs—44 %—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 program exploration strategy should be revisited.

Most of the bugs in Table A2 were reached by all fuzzers. SymCC-AFL was the worst performing fuzzer in this regard, failing to reach nine bugs (the highest amongst the seven 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.

The longer, 7 d campaigns in Table A3 reveal a peculiar result: while honggfuzz was faster to trigger bugs during the 24 h campaigns, MOpt-AFL was faster to trigger 11 additional bugs after 24 h, making it the most successful fuzzer over the 7 d campaigns. Notably, honggfuzz failed to trigger any of these 11 bugs. This highlights the importance of long fuzzing campaigns and the utility of Magma’s survival time analysis for comparing fuzzer performance.

Figure 5 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 estimator 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. Confidence intervals are shown as shaded regions. Figure 5a shows the time to reach bug AAH018 (*libtiff*). Notably, this bug was not triggered by any of the seven evaluated fuzzers. Thus, the probability of bug AAH018 "surviving" 24 h (i.e., not being triggered) remains at one. In comparison, Figure 5b 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 5d and Figure 5c 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 Section 4.1 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.

```
void png_check_chunk_length(png_ptr, 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 4. Divide-by-zero bug in *libpng*. Input undergoes non-trivial transformations to trigger the bug.

6.3.3 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 with a width of `0x55555555`. This "magic value" is not encoded anywhere in the target, and is easily calculated by solving the constraints for `row_factor == 0`. However, mutational fuzzers struggle to discover this bug type. This is because mutational fuzzers sample from an extremely large input space, making them unlikely to pick the exact byte sequence required to trigger the bug (here, `0x55555555`). Notably, only honggfuzz, AFL, and SymCC-AFL were able to trigger this bug. SymCC-AFL was the fastest to do so, likely due to its constraint-solving capabilities.

6.3.4 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 remains unmarked for release upon object destruction, leading to a 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

Table 3. Overheads introduced by LAVA-M compared to `coreutils-8.24`. These overheads denote increases in LLVM IR instruction counts, object file sizes, and average runtimes when processing seeds generated from a 24 h fuzzing campaign. The total number of unique bugs triggered across all 10 trials/fuzzer is also shown, with the best performing fuzzer highlighted in green.

Target	Bugs	Overheads (%)			Total bugs triggered (#)						
		LLVM IR	Size	Runtime	afl	aflfast	afl++	moptafl	fairfuzz	honggfuzz	symccafl
base64	44	107.9	57.2	9.7	1	0	48	0	3	33	0
md5sum	57	60.2	46.1	9.5	0	1	40	1	1	29	0
uniq	28	63.6	27.8	11.6	3	0	29	1	0	13	3
who	2136	1786.7	2409.1	42.9	1	1	819	1	1	750	1

satisfies string magic-value checks. SymCC-AFL also consistently triggers this bug, demonstrating how whitebox fuzzers can trivially solve constraints based on magic values.

6.3.5 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.

Semantic bugs are not always 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 [45]).

6.3.6 Comparison to LAVA-M. In addition to our Magma evaluation, we also evaluate the same seven fuzzers against LAVA-M, measuring (a) the overheads introduced by LAVA-M's bug oracles, and (b) the total number of bugs found by each fuzzer (across a 24 h campaign, repeated 10 times per fuzzer). These results—presented in Table 3—show that LAVA-M's most iconic target, *who*, accounts for 94.3 % of the benchmark's bugs. This high bug count reduces the amount of functional code (compared to benchmark instrumentation) in the *who* binary to 5.3 %, impeding a fuzzer's exploration capabilities. Notably, we found that the evaluated fuzzers spent (on average) 42.9 % of their time executing oracle code in *who* (this percentage is based on the final state of the fuzzing queue, and may not represent the runtime overhead of *all* code paths). Finally, the bug counts found by each fuzzer show a clear bias towards fuzzers with magic-value detection capabilities (due to LAVA-M's single, simple bug type, per Section 2.2.1).

6.4 Discussion

6.4.1 Ground Truth and Confidence. Ground truth enables us to determine a crash's root cause. Unlike many existing benchmarks, Magma provides straightforward access to ground truth. While ground truth is available for all 118 bugs, only 45 % of these bugs have a PoV that demonstrate triggerability. Importantly, only bugs with PoVs can be used to confidently measure a fuzzer's performance. Regardless, bugs without a PoV remain useful: any fuzzer evaluated against Magma can produce a PoV, increasing the benchmark's utility. Widespread adoption of Magma will increase the number of bugs with PoVs. Notably, Table A3 shows that running the benchmark for longer indeed yields more PoVs for previously-untriggered bugs. We leave it as an open challenge to generate PoVs for these bugs.

6.4.2 Beyond Crashes. While Magma's instrumentation does not collect information about *detected* bugs (detection is a characteristic of the fuzzer, not the bug itself), it does enable the evaluation of this metric through a post-processing step (supported by fatal canaries).

In particular, bugs should not be restricted to crash-triggering faults. For example, some bugs result in resource starvation (e.g., unbounded loops or `mallocs`), privilege escalation, or undesirable outputs. Importantly, fuzzer developers recognize the need for additional bug-detection mechanisms: AFL has a hang timeout, and SlowFuzz searches for inputs that trigger worst-case behavior. Excluding non-crashing bugs from an 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 for an evaluation of path exploration, while bugs triggered and detected allows for an evaluation of a fuzzer's constraint generation/solving capabilities). It also allows us to identify which bug classes continue to evade state-of-the-art sanitization techniques (and to what degree).

6.4.3 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 platform 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. We achieve this by forward-porting 118 bugs across seven diverse targets. However, this is only the beginning. 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. Increased adoption will only strengthen Magma's value, and thus we encourage fuzzer developers to incorporate their fuzzers into Magma.

We evaluated Magma against seven popular open-source mutation-based fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, honggfuzz, and SymCC-AFL). Our evaluation shows that ground truth enables systematic comparison of fuzzer performance. Our evaluation provides tangible insight into fuzzer performance, 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 used 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. Magma is open-source and available at https://hexhive.epfl.ch/magma/.

REFERENCES

[1] Kayla Afanador and Cynthia Irvine. 2020. Representativeness in the Benchmark for Vulnerability Analysis Tools (B-VAT). In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association. https://www.usenix.org/conference/cset20/presentation/afanador

[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] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488
[8] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko
[9] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, New York, NY, USA. https://doi.org/10.1145/3368089.3409729
[10] 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
[11] Brian Caswell. [n.d.]. Cyber Grand Challenge Corpus. http://www.lungetech.com/cgc-corpus/.
[12] 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
[13] 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
[14] 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
[15] T. Dullien. 2020. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing* 8, 2 (2020), 391–403.
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi Accessed: 2020-10-19.
[17] 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
[18] 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
[19] Google. [n.d.]. FuzzBench. https://google.github.io/fuzzbench/. Accessed: 2020-05-02.
[20] Google. [n.d.]. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite. Accessed: 2019-09-06.
[21] Google. [n.d.]. honggfuzz. http://honggfuzz.com/. Accessed: 2019-10-19.
[22] 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
[23] 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
[24] John L. Henning. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7 (July 2000), 28–35. https://doi.org/10.1109/2.869367

[25] Intel. [n.d.]. Intel Pin API Reference. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html.

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

[27] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L. K. John. 2006. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.* 55, 6 (2006), 769–782.

[28] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.

[29] V. Kashyap, J. Ruchti, L. Kot, E. Turetsky, R. Swords, S. A. Pan, J. Henry, D. Melski, and E. Schulte. 2019. Automated Customized Bug-Benchmark Generation. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 103–114.

[30] 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

[31] 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

[32] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[33] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

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

[35] 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*.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[37] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[38] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2946563

[39] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. https://doi.org/10.1145/3377811.3380421

[40] MITRE. 2007. Common Weakness Enumeration (CWE). https://cwe.mitre.org/.

[41] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 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

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

[43] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (1901), 559–572.

[44] 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

[45] 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

[46] 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

[47] 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

[48] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. 2005. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* 10–20.

[49] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[50] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[51] 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.

[52] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

[53] 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

[54] 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.*

[55] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1275–1295. https://doi.org/10.1109/SP.2019.00010

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

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

[58] 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

[59] Trail of Bits. [n.d.]. DARPA Challenge Binaries on Linux, OS X, and Windows. https://github.com/trailofbits/cb-multios/. Accessed: 2020-10-04.

[60] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[61] Jonas Benedict Wagner. 2017. Elastic Program Transformations Automatically Optimizing the Reliability/Performance Trade-off in Systems Software. (2017), 149. https://doi.org/10.5075/epfl-thesis-7745

[62] 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

[63] 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

[64] 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

[65] 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

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

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. PoVs of (∗)-marked bugs are sourced from bug reports.

Bug ID	Report	Class	PoV	Scopes
AAH001	CVE-2018-13785	Integer overflow, divide by zero	✓	1
AAH002*	CVE-2019-7317	Use-after-free	✓	4
AAH003	CVE-2015-8472	API inconsistency	✓	2
AAH004	CVE-2015-0973	Integer overflow	✗	1
AAH005*	CVE-2014-9495	Integer overflow, Buffer overflow	✓	1
AAH007	(Unspecified)	Memory leak	✓	2
AAH008	CVE-2013-6954	0-pointer dereference	✓	2
AAH009	CVE-2016-9535	Heap buffer overflow	✓	1
AAH010	CVE-2016-5314	Heap buffer overflow	✓	1
AAH011	CVE-2016-10266	Divide by zero	✗	2
AAH012	CVE-2016-10267	Divide by zero	✗	1
AAH013	CVE-2016-10269	OOB read	✓	1
AAH014	CVE-2016-10269	OOB read	✓	1
AAH015	CVE-2016-10270	OOB read	✓	4
AAH016	CVE-2015-8784	Heap buffer overflow	✓	1
AAH017	CVE-2019-7663	0-pointer dereference	✓	1
AAH018*	CVE-2018-8905	Heap buffer underflow	✓	1
AAH019	CVE-2018-7456	OOB read	✗	1
AAH020	CVE-2016-3658	Heap buffer overflow	✓	2
AAH021	CVE-2018-18557	OOB write	✗	2
AAH022	CVE-2017-11613	Resource Exhaustion	✓	2
AAH024	CVE-2017-9047	Stack buffer overflow	✓	2
AAH025	CVE-2017-0663	Type confusion	✓	1
AAH026	CVE-2017-7375	XML external entity	✓	1
AAH027	CVE-2018-14567	Resource exhaustion	✗	1
AAH028	CVE-2017-5130	Integer overflow, heap corruption	✗	1
AAH029	CVE-2017-9048	Stack buffer overflow	✓	2
AAH030	CVE-2017-8872	OOB read	✗	2
AAH031	ISSUE #58 (gitlab)	OOB read	✗	1
AAH032	CVE-2015-8317	OOB read	✓	2
AAH033	CVE-2016-4449	XML external entity	✗	1
AAH034	CVE-2016-1834	Heap buffer overflow	✗	2
AAH035	CVE-2016-1836	Use-after-free	✓	2
AAH036	CVE-2016-1837	Use-after-free	✗	1
AAH037	CVE-2016-1838	Heap buffer overread	✓	2
AAH038	CVE-2016-1839	Heap buffer overread	✗	1
AAH039	BUG 758518	Heap buffer overread	✗	1
AAH040	CVE-2016-1840	Heap buffer overflow	✗	1
AAH041	CVE-2016-1762	Heap buffer overread	✓	1
AAH042	CVE-2019-14494	Divide-by-zero	✓	1
AAH043	CVE-2019-9959	Resource exhaustion (memory)	✓	1
AAH045	CVE-2017-9865	Stack buffer overflow	✓	4
AAH046	CVE-2019-10873	0-pointer dereference	✓	2
AAH047*	CVE-2019-12293	Heap buffer overread	✓	1
AAH048	CVE-2019-10872	Heap buffer overflow	✓	3
AAH049	CVE-2019-9200	Heap buffer underwrite	✓	1
AAH050	Bug #106061	Divide-by-zero	✓	1
AAH051*	ossfuzz/8499	Integer overflow	✓	1
AAH052	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	✗	2
JCH204	CVE-2018-20481	0-pointer dereference	✗	1
JCH206	CVE-2018-19058	Type confusion	✗	2
JCH207	CVE-2018-13988	OOB read	✓	1
JCH208	CVE-2019-12360	Stack buffer overflow	✗	1
JCH209	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-2019-12493	Stack buffer overread	✗	3

Bug ID	**Report**	**Class**	**PoV**	**Scopes**
AAH054	CVE-2016-2842	OOB write	✗	5
AAH055	CVE-2016-2108	OOB read	✓	5
AAH056	CVE-2016-6309	Use-after-free	✓	1
AAH057	CVE-2016-2109	Resource exhaustion (memory)	✗	2
AAH058	CVE-2016-2176	Stack buffer overread	✗	2
AAH059	CVE-2016-6304	Resource exhaustion (memory)	✗	3
MAE100	CVE-2016-2105	Integer overflow	✗	1
MAE102	CVE-2016-6303	Integer overflow	✗	1
MAE103	CVE-2017-3730	0-pointer dereference	✗	1
MAE104	CVE-2017-3735	OOB read	✓	1
MAE105	CVE-2016-0797	Integer overflow	✗	2
MAE106	CVE-2015-1790	0-pointer dereference	✗	2
MAE107	CVE-2015-0288	0-pointer dereference	✗	1
MAE108	CVE-2015-0208	0-pointer dereference	✗	1
MAE109	CVE-2015-0286	Type confusion	✗	1
MAE110	CVE-2015-0289	0-pointer dereference	✗	1
MAE111	CVE-2015-1788	Resource exhaustion (CPU)	✗	1
MAE112	CVE-2016-7052	0-pointer dereference	✗	1
MAE113	CVE-2016-6308	Resource exhaustion (memory)	✗	2
MAE114	CVE-2016-6305	Resource exhaustion (CPU)	✗	1
MAE115	CVE-2016-6302	OOB read	✓	1
JCH214	CVE-2019-9936	Heap buffer overflow	✗	1
JCH215	CVE-2019-20218	Stack buffer overread	✓	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-2019-19244	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
MAE002	CVE-2019-9021	Heap buffer overread	✗	1
MAE004	CVE-2019-9641	Uninitialized memory access	✗	1
MAE006	CVE-2019-11041	OOB read	✗	1
MAE008	CVE-2019-11034	OOB read	✓	1
MAE009	CVE-2019-11039	OOB read	✗	1
MAE010	CVE-2019-11040	Heap buffer overflow	✗	1
MAE011	CVE-2018-20783	OOB read	✗	3
MAE012	CVE-2019-9022	OOB read	✗	2
MAE014	CVE-2019-9638	Uninitialized memory access	✓	1
MAE015	CVE-2019-9640	OOB read	✗	2
MAE016	CVE-2018-14883	Heap buffer overread	✓	2
MAE017	CVE-2018-7584	Stack buffer underread	✗	1
MAE018	CVE-2017-11362	Stack buffer overflow	✗	1
MAE019	CVE-2014-9912	OOB write	✗	1
MAE020	CVE-2016-10159	Integer overflow	✗	2
MAE021	CVE-2016-7414	OOB read	✗	2

Received August 2020; revised September 2020; accepted October 2020

Table A2. Mean bug survival times—both **R**eached and **T**riggered—over a 24-hour period, in **s**econds, **m**inutes, and **h**ours. Bugs are sorted by "difficulty" (mean times). The best performing fuzzer is highlighted in green (ties are not included).

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH037	10.00s	20.00s	10.00s	10.00s	10.00s	45.50s	5.00s	15.00s	5.00s	15.00s	5.00s	15.00s	10.00s	25.50s	7.86s	20.86s
AAH041	15.00s	21.00s	10.00s	10.00s	15.00s	48.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	15.00s	30.00s	12.14s	22.00s
AAH003	10.00s	16.00s	10.00s	11.00s	10.00s	15.00s	5.00s	10.00s	5.00s	10.00s	5.00s	10.00s	10.00s	1.58m	7.86s	23.86s
JCH207	10.00s	1.12m	5.00s	1.57m	10.00s	1.94m	5.00s	2.05m	5.00s	1.60m	5.00s	1.42m	10.00s	1.62m	7.14s	1.62m
AAH056	15.00s	14.57m	10.00s	14.43m	15.00s	19.49m	10.00s	13.07m	10.00s	11.27m	10.00s	8.17m	15.00s	17.80m	12.14s	14.11m
AAH015	32.50s	1.57m	10.00s	13.50s	27.00s	17.50m	1.18m	34.59m	52.00s	10.84m	1.07m	10.86m	15.07m	1.02h	2.76m	19.55m
AAH055	15.00s	40.86m	10.00s	2.71m	15.00s	3.62h	10.00s	25.01m	10.00s	2.24h	10.00s	6.36h	15.00s	2.44h	12.14s	2.26h
AAH020	5.00s	2.32h	5.00s	2.12h	5.00s	31.62m	5.00s	2.01h	5.00s	55.17m	5.00s	49.92m	5.00s	11.22h	5.00s	2.85h
MAE016	10.00s	1.57m	5.00s	10.00s	10.00s	5.79m	5.00s	3.97m	5.00s	4.93m	5.00s	2.21h	24.00h	24.00h	3.43h	3.78h
AAH052	15.00s	3.17m	15.00s	14.10m	15.00s	45.03m	10.00s	3.94h	10.00s	10.56h	10.00s	12.02h	15.00s	5.28m	12.86s	3.95h
AAH032	15.00s	3.38m	5.00s	2.06m	15.00s	1.65h	10.00s	3.22h	10.00s	34.19m	10.00s	9.67h	15.00s	12.95h	11.43s	4.02h
MAE008	15.00s	1.42h	10.00s	9.73h	15.00s	1.44m	10.00s	1.14m	10.00s	1.54m	10.00s	12.08h	24.00h	24.00h	3.43h	6.76h
AAH022	32.50s	54.98m	10.00s	34.86m	27.00s	3.47h	1.18m	9.38h	52.00s	5.66h	1.07m	14.04h	15.07m	15.25h	2.76m	7.04h
MAE014	15.00s	1.11h	10.00s	4.11h	15.00s	14.52m	10.00s	5.58m	10.00s	8.28m	10.00s	21.83h	24.00h	24.00h	3.43h	7.36h
JCH215	2.14m	3.24h	15.00s	40.97m	22.30m	11.97h	2.37h	15.67h	48.87m	11.51h	3.23h	9.86h	1.85h	18.08h	1.24h	10.15h
AAH017	5.19h	5.20h	22.32h	22.32h	13.97h	13.97h	19.84h	19.84h	8.67h	9.20h	5.92h	5.92h	9.92h	9.92h	12.26h	12.34h
JCH232	4.87h	4.87h	43.86m	1.66h	14.87h	20.02h	19.82h	19.82h	14.93h	17.21h	6.23h	10.31h	21.81h	21.81h	11.89h	13.67h
AAH014	12.48h	12.48h	24.00h	24.00h	13.06h	13.06h	6.34h	6.34h	24.00h	24.00h	18.46h	18.46h	10.68h	10.68h	15.57h	15.57h
JCH201	15.00s	14.65h	10.00s	24.00h	15.00s	19.48h	10.00s	16.82h	10.00s	12.98h	10.00s	14.02h	15.00s	14.27h	12.14s	16.60h
AAH007	15.00s	24.00h	5.00s	57.00s	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	23.12m	11.43s	17.20h
AAH008	15.00s	16.51h	10.00s	3.65h	15.00s	23.40h	10.00s	19.44h	10.00s	19.66h	10.00s	15.28h	15.00s	23.43h	12.14s	17.34h
AAH045	20.00s	3.33h	13.50s	1.13h	20.00s	24.00h	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	20.00s	24.00h	16.93s	17.78h
AAH013	24.00h	24.00h	4.05h	4.05h	13.88h	13.88h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	19.70h	19.70h
AAH024	15.00s	9.05h	10.00s	9.27h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	19.76h
JCH209	14.40m	14.41m	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	20.61h	20.61h
MAE115	15.00s	22.64h	10.00s	20.96h	15.00s	24.00h	10.00s	21.32h	10.00s	23.33h	10.00s	21.97h	15.00s	10.13h	12.14s	20.62h
AAH026	15.00s	20.88h	10.00s	7.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	21.13h
AAH001	15.00s	22.57h	10.00s	17.70h	15.00s	24.00h	10.00s	22.60h	10.00s	24.00h	10.00s	24.00h	15.00s	14.58h	12.14s	21.35h
MAE104	15.00s	15.53h	10.00s	24.00h	15.00s	24.00h	10.00s	21.81h	10.00s	17.60h	10.00s	24.00h	24.00h	24.00h	3.43h	21.56h
AAH010	21.35h	21.97h	12.53h	16.40h	14.59m	20.34h	10.18m	24.00h	24.00h	24.00h	13.81h	21.79h	4.76h	24.00h	10.98h	21.79h
AAH016	18.68h	19.66h	24.00h	24.00h	22.59h	22.59h	24.00h	24.00h	17.61h	19.83h	19.89h	19.97h	24.00h	24.00h	21.54h	22.01h
JCH226	23.20h	23.72h	4.09h	10.93h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	21.04h	22.09h
JCH228	12.33h	18.10h	2.47h	20.05h	22.07h	24.00h	22.57h	22.60h	24.00h	24.00h	18.78h	24.00h	22.66h	23.80h	17.84h	22.36h
AAH035	15.00s	19.34h	10.00s	24.00h	21.50s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	19.00s	24.00h	13.64s	23.33h
JCH212	15.00s	24.00h	10.00s	20.42h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	23.49h
AAH025	22.22h	22.22h	22.48h	22.48h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.53h	23.53h
AAH053	24.00h	24.00h	35.00s	21.80h	24.00h	24.00h	30.00s	24.00h	29.50s	24.00h	26.00s	24.00h	24.00h	24.00h	10.29h	23.69h
AAH042	39.50s	21.93h	20.00s	24.00h	39.50s	24.00h	40.00s	24.00h	34.50s	24.00h	31.00s	24.00h	45.00s	24.00h	35.64s	23.70h
AAH048	15.00s	24.00h	10.00s	22.72h	16.50s	24.00h	15.00s	24.00h	10.50s	24.00h	10.00s	24.00h	20.00s	24.00h	13.86s	23.82h
AAH049	15.00s	22.82h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	23.83h
AAH043	25.00s	22.91h	16.80h	24.00h	2.41h	24.00h	25.00s	24.00h	20.00s	24.00h	20.00s	24.00h	25.00s	24.00h	2.75h	23.84h
JCH210	30.00s	23.07h	20.00s	24.00h	33.00s	24.00h	30.00s	24.00h	25.00s	24.00h	25.00s	24.00h	32.50s	24.00h	27.93s	23.87h
AAH050	25.00s	24.00h	16.80h	23.71h	29.00s	24.00h	24.00h	24.00h	20.00s	24.00h	20.00s	24.00h	29.00s	24.00h	5.83h	23.96h

Table A2. Mean bug survival times (cont.). None of these bugs were triggered by the seven evaluated fuzzers.

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH054	10.00s	24.00h	5.00s	24.00h	10.00s	24.00h	5.00s	24.00h	5.00s	24.00h	5.00s	24.00h	10.00s	24.00h	7.14s	24.00h
MAE105	10.00s	24.00h	5.00s	24.00h	10.00s	24.00h	5.00s	24.00h	5.00s	24.00h	5.00s	24.00h	10.00s	24.00h	7.14s	24.00h
AAH011	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
AAH005	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
JCH202	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
MAE114	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH029	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH034	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH004	16.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.29s	24.00h
MAE111	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	20.00s	24.00h	12.86s	24.00h
AAH059	20.00s	24.00h	10.00s	24.00h	17.00s	24.00h	15.00s	24.00h	15.00s	24.00h	10.00s	24.00h	20.00s	24.00h	15.29s	24.00h
JCH204	18.00s	24.00h	20.00s	24.00h	15.50s	24.00h	15.00s	24.00h	15.00s	24.00h	10.00s	24.00h	20.00s	24.00h	16.21s	24.00h
AAH031	20.00s	24.00h	15.00s	24.00h	42.00s	24.00h	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	25.00s	24.00h	21.00s	24.00h
AAH051	25.00s	24.00h	10.00s	24.00h	42.50s	24.00h	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h	30.00s	24.00h	23.93s	24.00h
MAE103	33.00s	24.00h	28.00s	24.00h	33.00s	24.00h	27.50s	24.00h	25.00s	24.00h	20.00s	24.00h	31.00s	24.00h	28.21s	24.00h
JCH214	33.50s	24.00h	45.00s	24.00h	36.00s	24.00h	31.00s	24.00h	26.50s	24.00h	25.00s	24.00h	35.00s	24.00h	33.14s	24.00h
JCH220	4.38m	24.00h	11.50s	24.00h	22.04m	24.00h	2.09h	24.00h	54.77m	24.00h	3.12h	24.00h	2.28h	24.00h	1.26h	24.00h
JCH229	4.53m	24.00h	16.00s	24.00h	24.62m	24.00h	2.80h	24.00h	1.07h	24.00h	3.23h	24.00h	2.32h	24.00h	1.42h	24.00h
AAH018	41.88m	24.00h	4.00m	24.00h	5.77h	24.00h	3.17h	24.00h	59.96m	24.00h	36.01m	24.00h	1.85h	24.00h	1.88h	24.00h
JCH230	4.02m	24.00h	22.50s	24.00h	1.07h	24.00h	3.31h	24.00h	1.36h	24.00h	3.56h	24.00h	5.57h	24.00h	2.13h	24.00h
AAH047	25.00s	24.00h	16.80h	24.00h	2.41h	24.00h	25.00s	24.00h	20.00s	24.00h	20.00s	24.00h	25.00s	24.00h	2.75h	24.00h
JCH233	8.31m	24.00h	12.02m	24.00h	6.16h	24.00h	3.87h	24.00h	1.98h	24.00h	3.59h	24.00h	5.17h	24.00h	3.02h	24.00h
JCH223	16.59m	24.00h	30.50s	24.00h	1.19h	24.00h	3.89h	24.00h	1.33h	24.00h	4.03h	24.00h	10.60h	24.00h	3.05h	24.00h
JCH231	21.88m	24.00h	36.00s	24.00h	2.44h	24.00h	3.96h	24.00h	1.41h	24.00h	4.05h	24.00h	10.62h	24.00h	3.27h	24.00h
MAE006	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	24.00h	24.00h	3.43h	24.00h
MAE004	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	24.00h	24.00h	3.43h	24.00h
JCH222	1.75h	24.00h	21.97m	24.00h	18.91h	24.00h	15.17h	24.00h	13.39h	24.00h	18.87h	24.00h	20.82h	24.00h	12.75h	24.00h
AAH009	14.61h	24.00h	20.62h	24.00h	24.00h	24.00h	5.67h	24.00h	19.45h	24.00h	17.62h	24.00h	23.42h	24.00h	17.91h	24.00h
JCH227	24.00h	24.00h	20.58h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.51h	24.00h
JCH219	23.41h	24.00h	23.22h	24.00h	24.00h	24.00h	24.00h	24.00h	23.79h	24.00h	24.00h	24.00h	24.00h	24.00h	23.77h	24.00h
JCH216	23.48h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.93h	24.00h

Table A3. Mean bug survival times over a 7-day period.

Bug ID	moptafl R	moptafl T	honggfuzz R	honggfuzz T	afl++ R	afl++ T	afl R	afl T	aflfast R	aflfast T	fairfuzz R	fairfuzz T	symccafl R	symccafl T	**Mean** R	**Mean** T
AAH037	10.00s	20.00s	15.00s	15.00s	10.00s	45.50s	10.00s	20.00s	10.00s	20.00s	10.00s	21.00s	10.00s	25.50s	10.71s	23.86s
AAH041	15.00s	21.00s	15.00s	15.00s	15.00s	48.00s	15.00s	20.50s	15.00s	20.00s	15.00s	21.00s	15.00s	30.00s	15.00s	25.07s
AAH003	10.00s	16.00s	15.00s	17.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	10.00s	1.58m	10.71s	26.86s
JCH207	10.00s	1.12m	10.00s	2.16m	10.00s	1.94m	10.00s	2.02m	10.00s	3.73m	10.00s	2.96m	10.00s	1.62m	10.00s	2.22m
AAH056	15.00s	14.57m	15.00s	19.65m	15.00s	19.49m	15.00s	16.75m	15.00s	14.69m	15.00s	11.16m	15.00s	17.80m	15.00s	16.30m
AAH015	32.50s	1.57m	15.00s	21.50s	27.00s	17.50m	1.40m	59.27m	1.12m	8.00m	1.23m	13.34m	15.07m	1.02h	2.87m	23.04m
AAH020	5.00s	2.32h	5.00s	2.37h	5.00s	31.62m	5.00s	2.40h	5.00s	49.68m	5.00s	49.06m	5.00s	11.22h	5.00s	2.93h
AAH052	15.00s	3.17m	18.00s	15.09m	15.00s	45.03m	15.00s	3.83h	15.00s	6.17h	15.00s	13.78h	15.00s	5.28m	15.43s	3.56h
AAH022	32.50s	54.98m	15.00s	19.83m	27.00s	3.47h	1.40m	19.31h	1.12m	3.54h	1.23m	11.50h	15.07m	15.63h	2.87m	7.81h
AAH055	15.00s	40.86m	15.00s	4.07m	15.00s	3.62h	15.00s	4.17h	15.00s	1.74h	15.00s	71.84h	15.00s	2.44h	15.00s	12.08h
AAH017	13.22h	13.23h	66.80h	66.84h	13.97h	13.97h	13.88h	14.38h	6.78h	6.78h	3.50h	3.53h	9.92h	9.92h	18.30h	18.38h
AAH032	15.00s	3.38m	10.00s	2.70m	15.00s	1.65h	15.00s	51.23h	15.00s	15.81m	15.00s	67.23h	15.00s	36.05h	14.29s	22.36h
MAE016	10.00s	1.57m	10.00s	15.00s	10.00s	5.79m	10.00s	2.38m	10.00s	6.25m	10.00s	3.13h	168.00h	168.00h	24.00h	24.49h
JCH215	2.14m	3.24h	23.50s	2.42h	22.30m	13.00h	1.87h	45.83h	21.33m	15.91h	48.42m	38.01h	1.85h	85.15h	45.45m	29.08h
MAE008	15.00s	1.42h	15.00s	14.11h	15.00s	1.44m	15.00s	3.87m	15.00s	2.25m	15.00s	33.70h	168.00h	168.00h	24.00h	31.05h
JCH201	15.00s	17.54h	15.00s	140.14h	15.00s	20.53h	15.00s	11.25h	15.00s	13.41h	15.00s	13.95h	15.00s	14.27h	15.00s	33.01h
MAE014	15.00s	1.11h	15.00s	55.08m	15.00s	14.52m	15.00s	4.37m	15.00s	10.03m	15.00s	154.13h	168.00h	168.00h	24.00h	46.38h
AAH014	14.52h	14.52h	122.24h	122.24h	13.06h	13.06h	18.54h	18.54h	143.74h	143.74h	75.78h	75.78h	38.20h	38.20h	60.87h	60.87h
JCH232	4.87h	4.87h	44.67m	2.35h	26.08h	48.03h	83.73h	117.35h	31.74h	50.30h	34.90h	101.98h	105.04h	117.65h	41.01h	63.22h
MAE115	15.00s	61.48h	15.00s	109.18h	15.00s	133.93h	15.00s	47.46h	15.00s	32.83h	15.00s	94.71h	15.00s	10.13h	15.00s	69.96h
AAH008	15.00s	32.45h	15.00s	3.39h	15.00s	141.24h	15.00s	25.27h	15.00s	126.02h	15.00s	117.94h	15.00s	55.21h	15.00s	71.65h
JCH209	14.40m	14.41m	168.00h	168.00h	63.14h	63.14h	62.32h	62.33h	44.40h	44.41h	154.76h	154.76h	49.70h	49.72h	77.51h	77.51h
MAE104	15.00s	57.64h	15.00s	168.00h	15.00s	109.74h	15.00s	114.85h	15.00s	51.48h	15.00s	40.27h	168.00h	168.00h	24.00h	101.43h
AAH010	41.60h	44.03h	22.39h	42.87h	14.59m	121.14h	14.80m	168.00h	139.00h	150.52h	39.15h	136.06h	19.16h	65.62h	37.40h	104.04h
JCH228	16.37h	35.61h	6.97h	60.72h	94.73h	117.15h	128.84h	153.74h	58.00h	111.25h	104.22h	126.98h	129.87h	146.81h	77.00h	107.47h
AAH007	15.00s	168.00h	10.00s	1.56m	15.00s	167.02h	15.00s	163.55h	15.00s	168.00h	15.00s	168.00h	15.00s	23.12m	14.29s	119.28h
AAH045	20.00s	3.33h	20.00s	21.44m	20.00s	168.00h	20.00s	168.00h	20.00s	164.38h	20.00s	168.00h	20.00s	164.86h	20.00s	119.56h
AAH013	168.00h	168.00h	2.40h	2.40h	13.88h	13.88h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	122.33h	122.33h
AAH016	49.04h	50.76h	140.25h	141.75h	44.30h	137.79h	81.63h	146.06h	110.77h	125.20h	120.45h	120.48h	154.66h	155.00h	100.16h	125.29h
AAH001	15.00s	152.17h	15.00s	12.17h	15.00s	168.00h	15.00s	144.94h	15.00s	168.00h	15.00s	168.00h	15.00s	76.79h	15.00s	127.15h
AAH024	15.00s	9.05h	15.00s	52.37h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	128.77h
AAH026	15.00s	77.04h	15.00s	15.91h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	133.28h
JCH226	54.49h	87.37h	3.58h	19.05h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	152.61h	168.00h	168.00h	168.00h	126.10h	135.20h
AAH049	15.00s	45.24h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	150.46h
JCH210	30.00s	63.56h	25.00s	155.54h	33.00s	168.00h	35.00s	168.00h	30.00s	168.00h	33.50s	168.00h	32.50s	168.00h	31.29s	151.30h
AAH035	15.00s	83.58h	15.00s	168.00h	21.50s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	19.00s	168.00h	16.50s	155.94h
JCH212	15.00s	168.00h	15.00s	94.40h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	157.49h
JCH227	68.10h	121.28h	110.51h	158.09h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	145.52h	159.91h
AAH025	139.13h	139.13h	155.04h	155.04h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	162.02h	162.02h
AAH043	25.00s	126.83h	168.00h	168.00h	16.81h	168.00h	26.50s	168.00h	25.00s	168.00h	25.00s	168.00h	25.00s	168.00h	26.41h	162.12h
AAH048	15.00s	168.00h	15.00s	128.38h	16.50s	168.00h	20.00s	168.00h	15.00s	168.00h	15.00s	168.00h	20.00s	168.00h	16.64s	162.34h
AAH050	25.00s	143.70h	151.20h	159.73h	29.00s	168.00h	33.61h	168.00h	28.00s	168.00h	29.00s	168.00h	29.00s	168.00h	26.41h	163.35h
JCH216	69.76h	142.55h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	153.97h	164.36h
AAH046	75.34h	149.74h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	154.76h	165.39h
AAH042	39.50s	151.53h	24.00s	168.00h	39.50s	168.00h	45.00s	168.00h	40.00s	168.00h	39.50s	168.00h	45.00s	168.00h	38.93s	165.65h
AAH009	21.86h	157.44h	104.71h	168.00h	125.76h	168.00h	6.48h	168.00h	56.27h	168.00h	29.35h	168.00h	24.07h	168.00h	52.64h	166.49h
JCH223	16.59m	158.18h	31.50s	168.00h	1.19h	168.00h	5.90h	168.00h	2.57h	168.00h	1.47h	168.00h	11.94h	168.00h	3.34h	166.60h
AAH029	15.00s	166.95h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	167.85h
JCH229	4.53m	167.67h	24.00s	168.00h	24.62m	168.00h	2.09h	168.00h	49.24m	168.00h	49.67m	168.00h	2.32h	168.00h	56.19m	167.95h

Table A3. Mean bug survival times (cont.).

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH054	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h
AAH011	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h
MAE105	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	15.00s	168.00h	10.00s	168.00h	10.71s	168.00h
AAH005	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
JCH202	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
MAE114	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
AAH034	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
AAH004	16.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.14s	168.00h
MAE111	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	20.00s	168.00h	15.71s	168.00h
AAH059	20.00s	168.00h	15.00s	168.00h	17.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	18.86s	168.00h
JCH204	18.00s	168.00h	24.00s	168.00h	15.50s	168.00h	20.00s	168.00h	20.00s	168.00h	19.00s	168.00h	20.00s	168.00h	19.50s	168.00h
AAH031	20.00s	168.00h	22.50s	168.00h	42.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	25.00s	168.00h	24.21s	168.00h
AAH051	25.00s	168.00h	15.00s	168.00h	42.50s	168.00h	25.00s	168.00h	25.00s	168.00h	15.00s	168.00h	30.00s	168.00h	25.36s	168.00h
JCH214	33.50s	168.00h	53.50s	168.00h	36.00s	168.00h	35.00s	168.00h	31.00s	168.00h	30.00s	168.00h	35.00s	168.00h	36.29s	168.00h
MAE103	33.00s	168.00h	1.05m	168.00h	33.00s	168.00h	40.00s	168.00h	33.50s	168.00h	30.00s	168.00h	31.00s	168.00h	37.64s	168.00h
JCH220	4.38m	168.00h	19.50s	168.00h	22.04m	168.00h	1.78h	168.00h	46.76m	168.00h	49.27m	168.00h	2.28h	168.00h	52.36m	168.00h
AAH018	41.88m	168.00h	5.81m	168.00h	5.77h	168.00h	1.72h	168.00h	1.60h	168.00h	1.40h	168.00h	1.85h	168.00h	1.88h	168.00h
JCH230	4.02m	168.00h	41.00s	168.00h	1.07h	168.00h	5.67h	168.00h	1.16h	168.00h	1.14h	168.00h	5.57h	168.00h	2.10h	168.00h
JCH233	8.31m	168.00h	23.55m	168.00h	6.16h	168.00h	11.84h	168.00h	1.44h	168.00h	2.41h	168.00h	5.17h	168.00h	3.93h	168.00h
JCH231	21.88m	168.00h	35.00s	168.00h	2.44h	168.00h	5.99h	168.00h	5.22h	168.00h	1.69h	168.00h	11.96h	168.00h	3.95h	168.00h
MAE006	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	168.00h	168.00h	24.00h	168.00h
MAE004	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	168.00h	168.00h	24.00h	168.00h
AAH047	25.00s	168.00h	168.00h	168.00h	16.81h	168.00h	26.50s	168.00h	25.00s	168.00h	25.00s	168.00h	25.00s	168.00h	26.41h	168.00h
JCH222	1.75h	168.00h	39.17m	168.00h	113.15h	168.00h	151.50h	168.00h	57.91h	168.00h	71.58h	168.00h	136.02h	168.00h	76.08h	168.00h
JCH219	72.02h	168.00h	168.00h	168.00h	162.32h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	153.48h	168.00h