Seed Selection for Successful Fuzzing

Adrian Herrera
ANU & DST
Australia

Hendra Gunadi
ANU
Australia

Shane Magrath
DST
Australia

Michael Norrish
CSIRO's Data61 & ANU
Australia

Mathias Payer
EPFL
Switzerland

Antony L. Hosking
ANU & CSIRO's Data61
Australia

ABSTRACT

Mutation-based greybox fuzzing—unquestionably the most widely-used fuzzing technique—relies on a set of non-crashing seed inputs (a corpus) to bootstrap the bug-finding process. When evaluating a fuzzer, common approaches for constructing this corpus include: (i) using an empty file; (ii) using a single seed representative of the target's input format; or (iii) collecting a large number of seeds (e.g., by crawling the Internet). Little thought is given to *how* this seed choice affects the fuzzing process, and there is no consensus on which approach is best (or even if a best approach exists).

To address this gap in knowledge, we systematically investigate and evaluate how seed selection affects a fuzzer's ability to *find bugs in real-world software*. This includes a systematic review of seed selection practices used in both evaluation and deployment contexts, and a large-scale empirical evaluation (over 33 CPU-years) of six seed selection approaches. These six seed selection approaches include three *corpus minimization* techniques (which select the smallest subset of seeds that trigger the same range of instrumentation data points as a full corpus).

Our results demonstrate that fuzzing outcomes vary significantly depending on the initial seeds used to bootstrap the fuzzer, with minimized corpora outperforming singleton, empty, and large (in the order of thousands of files) seed sets. Consequently, we encourage seed selection to be foremost in mind when evaluating/deploying fuzzers, and recommend that (a) seed choice be carefully considered and explicitly documented, and (b) never to evaluate fuzzers with only a single seed.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

KEYWORDS

fuzzing, corpus minimization, software testing

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464795

ACM Reference Format:
Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464795

1 INTRODUCTION

Fuzzing is a dynamic analysis technique for finding bugs and vulnerabilities in software, triggering crashes in a target program by subjecting it to a large number of (possibly malformed) inputs. *Mutation-based* fuzzing typically uses an initial set of valid seed inputs from which to generate new seeds by random mutation. Due to their simplicity and ease-of-use, mutation-based greybox fuzzers such as AFL [74], honggfuzz [64], and libFuzzer [61] are widely deployed, and have been highly successful in uncovering thousands of bugs across a large number of popular programs [6, 16]. This success has prompted much research into improving various aspects of the fuzzing process, including mutation strategies [39, 42], energy assignment policies [15, 25], and path exploration algorithms [14, 73]. However, while researchers often note the importance of high-quality input seeds and their impact on fuzzer performance [37, 56, 58, 67], few studies address the problem of *optimal design and construction of corpora* for mutation-based fuzzers [56, 58], and none assess the precise impact of these corpora in coverage-guided mutation-based greybox fuzzing.

Intuitively, the collection of seeds that form the initial corpus should generate a broad range of observable behaviors in the target. Similarly, candidate seeds that are behaviorally similar to one another should be represented in the corpus by a single seed. Finally, both the total size of the corpus and the size of individual seeds should be minimized. This is because previous work has demonstrated the impact that file system contention has on industrial-scale fuzzing. In particular, Xu et al. [71] showed that the overhead from opening/closing test-cases and synchronization between workers each introduced a 2× overhead. Time spent opening/closing test-cases and synchronization is time diverted from mutating inputs and expanding code coverage. Minimizing the total corpus size and the size of individual test-cases reduces this wastage and enables time to be (better) spent on finding bugs.

Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the "empty corpus" (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we survey here, the majority of published research uses either (a) the "singleton corpus" (e.g., a single seed representative of the target program's input format),

or (b) the empty corpus. In contrast, industrial-scale fuzzing (e.g., Google's OSSFuzz [6]) typically uses large corpora containing hundreds of inputs. These inputs may generate similar behavior in the target, potentially leading to wasted fuzzing effort in exhaustively exploring mutation from all available seeds. The practice is to use *corpus minimization* tools that eliminate seeds that display duplicate behavior. However, these tools are based on heuristic algorithms and generate corpora that vary wildly in size. It is unclear which of these approaches (fuzzing with an empty, singleton, or minimized corpus) is best, or even if a best approach exists.

Thus, we undertake a systematic study to better understand the impact that seed selection has on a fuzzer's ultimate goal: finding bugs in real-world software. We make the following contributions:

- A systematic review of seed selection practices and corpus minimization techniques used across fuzzer evaluations and deployments. Our review finds that seed selection practices vary wildly, and that there is no consensus on the best way to select the seeds that bootstrap the fuzzing process (Section 3).
- A new corpus minimization tool, OptiMin, which produces an *optimal minimum* corpus (Section 4).
- A quantitative evaluation and comparison of various seed-selection practices. This evaluation covers three corpus minimization tools (including OptiMin), and finds that corpora produced with these tools perform better than singleton and empty seeds with respect to bug-finding ability (Section 5).

2 FUZZING

Fuzzing is the most popular technique for automatically finding bugs and vulnerabilities in software. This popularity can be attributed to its simplicity and success in finding bugs in "real-world" software [16, 61, 64, 74].

How a fuzzer generates test-cases depends on whether it is *generation*-based or *mutation*-based. Generation-based fuzzers (e.g., QuickFuzz [29], Dharma [47], and CodeAlchemist [30]) require a specification/model of the input format. They use this specification to synthesize test-cases. In contrast, mutation-based fuzzers (such as AFL [74], honggfuzz [64], and libFuzzer [61]) require an initial corpus of seed inputs (e.g., files, network packets, and environment variables) to bootstrap test-case generation. New test-cases are then generated by mutating inputs in this corpus.

Perhaps the most popular mutation-based fuzzer is American Fuzzy Lop (AFL) [74]. AFL is a *greybox* fuzzer, meaning that it uses lightweight code instrumentation to gather *code coverage* information during the fuzzing process. This code coverage information acts as an approximation of program behavior. AFL instruments edge transitions between basic blocks and uses this information as code coverage. By feeding the code coverage information back into the test-case mutation algorithm, the fuzzer can be driven to explore new code paths (and hence behaviors) in the target. Despite its popularity, AFL is no longer actively improved and is being phased out in favor of AFL++.[1] AFL++ [23] builds on AFL by incorporating state-of-the-art fuzzing research, including Redqueen's lightweight taint-tracking approximation [9], AFLFast's Markov chain model for seed scheduling [15], and MOpt's particle swarm optimization for selecting mutation operators [42].

[1]For example, AFL++ has replaced AFL in Google's OSSFuzz: https://github.com/google/oss-fuzz/commit/665e4898215c25a47dd29139f46c4f47f8139417.

3 SEED SELECTION PRACTICES

The process for selecting the initial corpus of seeds varies wildly across fuzzer evaluations and deployments. We systematically review this variation in the following sections, first considering experimental evaluations, followed by industrial-scale deployments.

3.1 In Experimental Evaluation

Remarkably, Klees et al. [37] found that "*most papers treated the choice of seed casually, apparently assuming that any seed would work equally well, without providing particulars*". In particular, of the 32 papers that they surveyed (authored between 2012 and 2018): ten used non-empty seed(s), but it was not clear whether these seeds were valid inputs; nine assumed the existence of valid seed(s), but did not report how these were obtained; five used a random sampling of seed(s); four used manually constructed seed(s); and two used empty seed(s). Additionally, six studies used a combination of seed selection techniques. Klees et al. [37] find that "*it is clear that a fuzzer's performance on the same program can be very different depending on what seed is used*" and recommend that "*papers should be specific about how seeds are collected*".

We examine an additional 28 papers published since 2018, to see if these recommendations have been adopted. Table 1 summarizes our findings.

Unreported seeds. Three studies make no mention of their seed selection procedure. One, FuzzGen, explicitly mentions that "*standardized common seeds*" [33] are key to a valid comparison, yet does not mention the seeds used.

Benchmark and fuzzer-provided seeds. Three studies (Hawkeye, FuzzFactory, and Entropic) evaluate fuzzers on the Google Fuzzer Test Suite (FTS) [26], which provides seeds for 14 of its 24 targets (FTS commit `5135606`). Of these seed sets, eight contain only one or two seeds. When no seed is provided, it is unclear which seeds these three studies used. Similarly, four papers (AFL-Sensitive, PTrix, Savior, and EcoFuzz) used the singleton seed sets provided by AFL.

Manually-constructed seeds. Two papers (Redqueen and Grimoire) use "*an uninformed, generic seed consisting of different characters from the printable ASCII set*" [9]. However, the authors do not justify (a) why this specific singleton corpus was chosen, and (b) what impact this choice has on the authors' real-world results, particularly when fuzzing `binutils`, where most of the targets accept non-ASCII, binary file formats (e.g., `readelf`, `objdump`).

Random seeds. Five papers (MOpt, Superion, FuZZan, GreyOne, and TortoiseFuzz) randomly select seeds from either (a) a larger corpus of seeds provided by developers of a particular target, or (b) by crawling the Internet. Of these studies, two (Superion and GreyOne) specifically mention using `afl-cmin`, AFL's corpus minimization tool (discussed further in Section 4), to remove duplicate seeds from the random seed set.

Empty seeds. Eight papers use an empty seed to bootstrap the fuzzing process. Interestingly, both Böhme et al. [13] and Böhme and Falk [12] explicitly removed the corpora provided by OSSFuzz

(discussed further in Section 3.2) because they found that "*initial seed corpora… are often for saturation: feature discovery has effectively stopped shortly after the beginning of the campaign*".

Table 1: Summary of past fuzzing evaluation, focusing on seed selection. We adopt the categories and notation used by Klees et al. [37]: R means randomly sampled seeds; M means manually constructed seeds; G means automatically generated seed; N means non-empty seed(s) with unknown validity; V means the paper assumes the existence of valid seed(s), but with unknown provenance; E means empty seeds; / means different seeds were used in different programs, but only one kind of seeds in one program; and a blank cell means that the paper's evaluation did not mention seed selection. We introduce an additional category: V* means valid seed(s) with known provenance. We also indicate whether the evaluation is reproducible ("Rep.") with the same seeds.

Paper	Seed	Rep.
CollAFL [25]		✗
Hawkeye [18]	E/V*	✗
Qsym [73]	M, V*	✓
AFL-Sensitive [69]	E/V*	✓
Cerebro [40]		✗
Redqueen [9]	M	✓
Grimoire [11]	M	✓
MOpt [42]	R	✓
Nautilus [8]	G, M	✓
pFuzzer [45]	E	✓
PTrix [20]	V*	✓
Superion [68]	R/V	✗
FuzzFactory [55]	V*, M	✓
Zest [54]	V	✓

Paper	Seed	Rep.
UnTracer [51]	V	✓
Ankou [43]	V*	✓
Entropic [13]	E/V*	✓
[12]	E	✓
Savior [19]	V*	✓
FuZZan [34]	E, V*, R	✓
EcoFuzz [72]	V*	✓
GreyOne [24]	R	✗
FuzzGen [33]		✗
FuzzGuard [75]	V/E	✗
Magma [31]	V*	✓
Muzz [17]	V	✗
ParmeSan [53]	E	✗
TortoiseFuzz [70]	R	✗

A Reproduction Experiment: Redqueen. To demonstrate the importance of seed selection, we reproduce an experiment from the Redqueen evaluation. Aschermann et al. [9] fuzz a number of programs from `binutils`, bootstrapping each trial with an "uninformed, generic seed" (discussed previously). Their `readelf` results are particularly striking: AFLFast and honggfuzz cover little code. We repeat this experiment, but use a variety of initial seeds, including: (i) the original, uninformed seed; (ii) a single, valid ELF file (from AFL's seed set); and (iii) a collection of ELF files sourced from the ALLSTAR [63] and Malpedia [57] datasets (reduced from 104,737 to 366 seeds using `afl-cmin`). In place of the original Redqueen (which we were unable to build and reproduce) we use AFL++ [23] with "CmpLog" instrumentation enabled; this reimplements Redqueen's "input-to-state correspondence".

Our results appear in Fig. 1, and clearly show the impact that seed choice has on code coverage. Similarly to the results of Aschermann et al. [9], AFLFast bootstrapped with the uninformed seed explores very little of `readelf`'s code: less than 1 %. However, this increases to about 38 % for AFLFast bootstrapped with the valid ELF file, making it much more competitive against both honggfuzz and AFL++ (although AFL++ still outperforms them both by around 15 %). Finally, while the `afl-cmin` corpus has a negligible impact on AFLFast and honggfuzz, it results in a significant improvement when fuzzing with AFL++, increasing coverage to about 60 %.

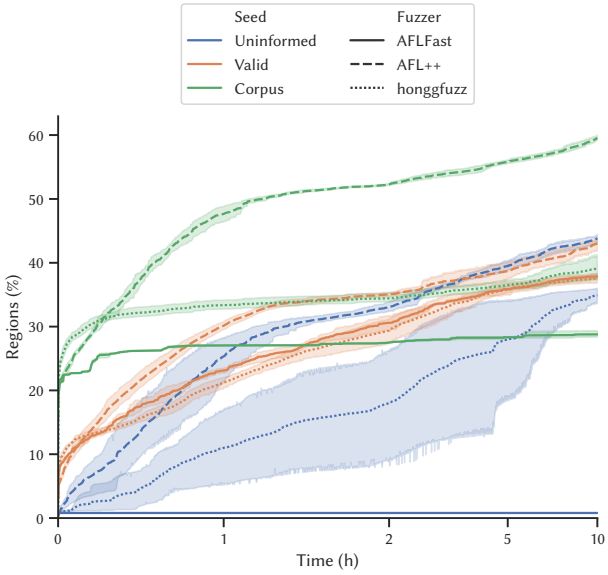


Figure 1: Code coverage of `readelf` with different initial seed sets. The mean coverage (using `llvm-cov`'s region coverage metric) and 95 % bootstrap confidence interval over five repeated 10 h trials is shown. The x-axis uses a log scale.

> At minimum, fuzzer evaluations must report the seed set used to bootstrap the fuzzing process. To ensure reproducibility, artifacts must provide the initial seed set (since results can vary wildly depending on the seeds used). Ideally, fuzzer evaluations should experiment with different initial seed corpora to see how varying initial seeds affects fuzzing outcomes.

3.2 In Deployment

In addition to being an active research topic, fuzzers are frequently deployed to find bugs in real-world software [6, 16, 48, 52]. Notably, practitioners also recognize the importance of seed selection. For example, the developers of the Mozilla Firefox browser remark [49]:

> *Mutation-based strategies are typically superior to others if the original samples* [i.e., seeds] *are of good quality because the originals carry a lot of semantics that the fuzzer does not have to know about or implement. However, success here really stands and falls with the quality of the samples. If the originals don't cover certain parts of the implementation, then the fuzzer will also have to do more work to get there.*

In contrast to the evaluation practices described in Section 3.1, industrial fuzzing eschews small seed sets and the empty seed in favor of large corpora. For example, seed corpora in Google's continuous fuzzing service for open-source software, OSSFuzz (commit `0deeef6`), range in size from a single seed (e.g., OpenThread, ICU) to 62,726 seeds (Suricata). Of the 363 OSSFuzz projects, 135 projects supply an initial corpus (~37 %) for 706 fuzzable targets. The mean corpus size is 1,083 seeds and the median is 36 seeds. More than half of the 135 projects include more than 100 seeds.

We examined the Suricata corpus in more detail, because it provided the largest number of seeds (62,726). We found large redundancy in these 62,726 seeds: we reduced the corpus to 31,234 seeds (~50 % decrease) by discarding seeds with an identical MD5 hash. We were able to reduce the size of the corpus further (down to 145 seeds, a 99 % reduction) by again applying `afl-cmin` (similarly to the `readelf` reproduction experiment described in Section 3.1). This redundancy is wasteful, as it leads to seeds clogging the fuzzing queue, which hinders and delays mutation of more promising seeds. We discuss corpus minimization in the following section.

> When deploying fuzzers at an industrial scale it is imperative that seeds exhibiting redundant behavior be removed from the fuzzing queue, as they will lead to wasted cycles.

4 CORPUS MINIMIZATION

Orthogonal to the seed selection practices discussed in Section 3, many popular fuzzers (e.g., AFL [74], libFuzzer [61], honggfuzz [64]) provide *corpus minimization* (sometimes called *distillation*) tools. Corpus minimization assumes that a large corpus of seeds already exists, and thus a corpus minimization tool *reduces* this large corpus to a subset of seeds that is then used to bootstrap the fuzzing process. When performing corpus minimization, the primary question that needs answering (as posed by Rebert et al. [58]) is:

> *Given a large collection of inputs for a particular target (the* collection corpus*), how do we select a subset of inputs that will form the initial* fuzzing corpus*?*

Abdelnur et al. [5] first formalized this problem as an instance of the *minimum set cover problem* (MSCP). The MSCP states that given a set of elements U (the universe) and a collection of N sets $S = s_1, s_2, \ldots, s_N$ whose union equals U, what is the *smallest* subset of S whose union still equals U. This smallest subset $C \subseteq S$ is known as the *minimum set cover*. Moreover, each $s_i \in S$ may be associated with a weight w_i. In this case, the *weighted* MSCP (WMSCP) attempts to minimize the total cost of elements in C.

[W]MSCP is NP-complete [36], so Abdelnur et al. [5] used a *greedy* algorithm to solve the unweighted MSCP. U consisted of code coverage information for the set of seeds in the original collection corpus. Subsequently, code coverage has continued to be used to characterize seeds in a fuzzing corpus due to the strong positive correlation between code coverage and bugs found while fuzzing [28, 38, 46, 50]. Finding C is therefore equivalent to finding the minimum set of seeds that still maintains the code coverage observed in the collection corpus.

A number of corpus minimization techniques have been proposed since the work of Abdelnur et al. [5]. We focus on file-format fuzzing and summarize the techniques relevant to our evaluation.

Minset. Rebert et al. [58] extended the work of Abdelnur et al. [5] by also computing C weighted by execution time or file size. They designed six corpus minimization techniques and both simulated and empirically evaluated these techniques over a number of fuzzing campaigns (using the BFF blackbox fuzzer). Rebert et al. [58] found that Unweighted Minset—an unweighted greedy-reduced minimization—performed best in terms of minimization ability, and that the Peach Set algorithm (based on the Peach fuzzer's `peachminset` tool [21]) found the highest number of bugs. Curiously, Rebert et al. [58] also found that `peachminset` does not in fact calculate C, nor a proven competitive approximation thereof. Our work extends Rebert et al. [58] with a more extensive evaluation based on modern *coverage-guided greybox fuzzing*.

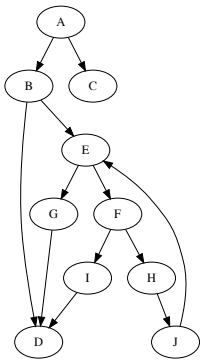
`afl-cmin`. Due to AFL's popularity, `afl-cmin` [74] is perhaps the most widely-used corpus minimization tool. It implements a greedy minimization algorithm, but has a unique approach to coverage. In particular, `afl-cmin` reuses AFL's own notion of edge coverage to categorize seeds at minimization time, recording an approximation of edge *frequency count*, not just whether the edge has been taken. Moreover, `afl-cmin` *bins* edge counts such that changes in edge frequency counts within a single bin are ignored, while transitions from one bin to another are "*flagged as an interesting change in program control flow*" [74]. When minimizing, `afl-cmin` chooses the smallest seed in the collection corpus that covers a given edge count, and then performs a greedy, weighted minimization. We consider `afl-cmin` and Rebert's Minset as representatives of the state-of-the-art in corpus minimization tools, and include both in our evaluation.

OptiMin. The previously-described corpus minimization techniques all employ heuristic algorithms to approximate C. This is because the underlying problem, the [W]MSCP, is NP-complete. However, in the case of corpus minimization, we found that exact solutions were nonetheless computable in reasonable time by encoding the problem as a maximum satisfiability problem (*MaxSAT*) and using an off-the-shelf MaxSAT backend. Thus, we implement OptiMin, an optimal corpus minimization tool for AFL.

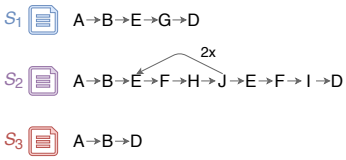
OptiMin[2] uses the EvalMaxSAT solver [10] to pose and solve corpus minimization as a MaxSAT problem. Unlike the *Boolean satisfiability problem* (SAT)—which determines whether the variables in a given Boolean formula can be assigned values to make the formula evaluate to true—the *maximum satisfiability problem* (MaxSAT) divides constraints into hard and soft constraints, and aims to satisfy all hard constraints and maximize the total number (or weighted total) of satisfied soft constraints. Here, OptiMin treats edge coverage as a hard constraint, while not including a particular seed in the solution is treated as a soft constraint. This approach ensures that the solution covers all edges with the minimal number of seeds, and is optimal in the sense that the solution C is guaranteed to be exact (rather than an approximation). This process is illustrated in Fig. 2: the program in Fig. 2a is executed with three seeds, producing the traces in Fig. 2b. These traces are translated into a set of (weighted) constraints (Fig. 2c) which are solved by EvalMaxSAT.

OptiMin is not the first tool to generate optimal solutions to minimization problems in software testing. For example, MINTS [32] and Nemo [41] both use integer linear programming (ILP) solvers to perform *test-suite minimization*; i.e., eliminate redundant test cases from a test suite "*based on any number of criteria [e.g., statement coverage, time-to-run, setup effort]*" [32]. When developing OptiMin, we explored the use of mixed integer programming solvers (which are more general than ILP solvers), but found that these solvers were *orders-of-magnitudes slower* than EvalMaxSAT.

[2] Available at https://github.com/HexHive/fuzzing-seed-selection.



(a) Program control-flow graph.



(b) Seed traces.

WOPT

Edges:

(A,B) :	$S_1 \vee S_2 \vee S_3$
(B,E) :	$S_1 \vee S_2$
(B,D) :	S_3
(E,F), (F,H), (F,I), (H,J), (I,D), and (J,E) :	S_2
(E,G), and (G,D) :	S_1

Hard constraint:

$(S_1 \vee S_2 \vee S_3) \wedge (S_1 \vee S_2) \wedge S_3 \wedge S_2 \wedge S_1$	weight = $\top$

Soft constraints:

$\neg S_1$	weight = W_1
$\neg S_2$	weight = W_2
$\neg S_3$	weight = W_3

WMOPT

Edges:

(B,E) :	$S_1 \vee S_2$
(A,B) :	$S_1 \vee S_2 \vee S_3$
(B,D) :	S_3
(E,F), (F,H), (F,I), (H,J), (I,D), $(J,E)_1$, and $(J,E)_2$:	S_2
(E,G), and (G,D) :	S_1

Hard constraint:

$(S_1 \vee S_2 \vee S_3) \wedge (S_1 \vee S_2) \wedge S_3 \wedge S_2 \wedge S_1$	weight = $\top$

Soft constraints:

$\neg S_1$	weight = W_1
$\neg S_2$	weight = W_2
$\neg S_3$	weight = W_3

(c) Weighted CNF constraints. Each edge is encoded as a disjunction of seeds that cover that edge. The final constraints are a conjunction of seed disjunctions (the hard constraint, assigned the maximum weight $\top$) and a set of soft constraints (the negation indicates a minimization), each with a weight W_i (for $i \in \{1,2,3\}$). For an unweighted minimization, $W_i = 1$, while for a weighted minimization W_i = the file size of S_i.
The left constraints are for a weighted minimization (WOPT in Section 5.1.5), while the right constraints are for a weighted minimization that takes into account edge frequencies (WMOPT in Section 5.1.5). In the latter, the loop backedge (J,E) has been split into two edges (the number of times that backedge was taken).

Figure 2: Corpus minimization with OptiMin. The program in Fig. 2a is executed with seeds S_1, S_2, and S_3, producing the traces in Fig. 2b. These traces are then translated into one of the constraint sets in Fig. 2c, which are solved by a MaxSAT solver. If the solver finds a solution where soft constraints $\neg S_i$, $\neg S_j \ldots$ are part of a satisfying assignment, the seeds S_i, $S_j \ldots$ can be omitted from the corpus.

5 EVALUATION

We perform a large-scale evaluation to understand the impact of seed selection on fuzzing's ultimate goal: finding bugs in real-world software. In particular, we aim to answer the following questions:

RQ1 How effective are corpus *minimization* tools at producing a minimal corpus? (Section 5.2)

RQ2 What effect does seed selection have on a fuzzer's *bug finding* ability? Do fuzzers perform better when bootstrapped with (a) a small seed set (e.g., empty or singleton set), or (b) a large corpus of seeds, derived from an even larger collection corpus after applying a corpus minimization tool? (Section 5.3)

RQ3 How does seed selection affect *code coverage*? Does starting from a corpus that executes more instrumentation data points result in greater code coverage, or does a fuzzer's mutation engine naturally achieve the same coverage (e.g., when starting from an empty seed)? (Section 5.4)

Notably, we find that while corpus minimization has a big impact on fuzzing campaigns, the underlying minimization tool is itself less important, as long as *some* form of minimization occurs. All experimental data is available at https://osf.io/hz8em/.

5.1 Methodology

5.1.1 Target Selection. We use targets from Magma [31], the Google Fuzzer Test Suite (FTS) [26], and six popular open-source programs (spanning 14 different file formats) to test different seed selection approaches. Table 2 details these targets.

We exclude some FTS targets, because: (i) they contain only memory leaks (e.g., proj4-2017-08-14), which are not detected by AFL by default, or (ii) we were unable to find a suitably-large collection corpus for a particular file type (e.g., ICC files for lcms-2017-03-21). This left us with 10 of the original 24 targets. Similarly, two Magma targets were excluded (openssl and sqlite3) because we were unable to find a suitably large corpus, leaving us with five targets.

We selected the six real-world targets to be representative of popular programs that are commonly fuzzed and that operate on a diverse range of file formats (e.g., images, audio, and documents).

5.1.2 Sample Collection. For each file type in Section 5.1.1, we built a Web crawler using Scrapy [60] to crawl the Internet for 72 h to create the collection corpus. For image files, crawling started with Google search results and the Wikimedia Commons repository. For media and document files, crawling started from the Internet Archive and Creative Commons collections. We used the regular expressions from regexlib [4], and sourced OGG files from old video games [1–3] (in addition to the Internet Archive). We sourced PHP files from test suites for popular PHP interpreters (e.g., Facebook's HipHop Virtual Machine) and from popular GitHub repositories (e.g., WordPress). Finally, we found TIFF files to be relatively rare, so we generated 40 % of the TIFF seeds by converting other image types such as JPEG and BMP using ImageMagick (v6.9.7).

We preprocessed each collection corpus to remove duplicates identified by MD5 checksum, and files larger than 300 KiB. The

cutoff file size of 300 KiB is our best effort to conform to the AFL authors' suggestions regarding seed size, while still having enough eligible seeds in the preprocessed corpora. We split audio files larger than 1 MiB into smaller files using FFmpeg (v3.4.8). In total, we collected 2,899,208 seeds across 14 different file formats. After preprocessing our collection corpus we were left with a total of 1,019,688 seeds. Our collection corpus is available at https://osf.io/hz8em/.

5.1.3 Experimental Setup. We run the Magma experiments on a cluster of AWS EC2 machines with 36-core Intel® Xeon® E5-2666 v3 2.9 GHz CPUs and 60 GiB of RAM. We conduct the FTS and real-world experiments on a pair of identically configured Dell PowerEdge servers with 48-core Intel® Xeon® Gold 5118 2.30 GHz CPUs and 512 GiB of RAM. All machines run Ubuntu 18.04.

5.1.4 Fuzzer Setup. We run one fuzzing *campaign* per target/filetype per initial corpus. Each fuzzing campaign consists of thirty independent 18 h *trials*. We emphasize the large number of repeated trials here because we found (consistent with Klees et al. [37]) that individual fuzzing trials vary wildly in performance. Therefore, reaching statistically meaningful conclusions requires many trials (and many fields of science use thirty trials [7]). The length of each trial and the number of repeated trials satisfy the recommendations of Klees et al. [37].

Our real-world fuzzing campaigns use AFL (v2.52b) in greybox mode, while our Magma and FTS campaigns also include AFL++ with (a) CmpLog instrumentation enabled and (b) a 250 KiB coverage map. We configure both fuzzers for single-system parallel execution with one main and one secondary node; the main node focuses on deterministic checks while the secondary node proceeds straight to havoc mode.

For FTS and the real-world targets we compile using AFL's LLVM (v8) instrumentation for 32-bit x86 and Address Sanitizer (ASan) [62]. We chose LLVM instrumentation over AFL's assembler-based instrumentation because LLVM's offers the best level of interoperability with ASan. We compile Magma targets using their default build configuration (i.e., for `x64` without ASan).

We tune AFL's timeout and memory parameters for each target to enable effective fuzzing.[3] When fuzzing the FTS we configure the target process to respawn after *every* iteration (due to stability issues that we encountered when fuzzing in parallel execution mode). All other parameters are left at their default values.

5.1.5 Experiment. We evaluate the following six seed selection approaches against the previously-described targets and fuzzers:

FULL The collection corpus without minimization, preprocessed to remove duplicates and filtering for size (as per Section 5.1.2).

EMPTY A per-target corpus comprising just an "empty" seed. For six filetypes (JSON, MP3, REGEX, TIFF, TTF, and XML) this seed is an empty file. For the remaining eight filetypes, the seed is not merely a zero-length input, but rather a small file handcrafted to contain the bytes necessary to satisfy file header checks (the `readelf` experiments in Section 3.1 demonstrate how poorly AFL performs when these header checks are not satisfied by the initial corpus). These files range in size from 11 B (SVG) to 13 KiB (OGG), with a median size of 51 B. We follow Klees et al. [37], who reported that "*despite its use contravening conventional wisdom*", the empty seed outperformed (in terms of bug yield) a set of valid non-empty seeds for some targets [37].

PROV The corpus provided with the benchmark (if any). This approach is only applicable for the two fuzzer benchmarks (Magma and FTS).

MSET The corpus obtained using the Unweighted Minset tool. We present Unweighted Minset (as opposed to Time or Size Minset) because it finds more bugs than other Minset configurations [58].

CMIN The corpus produced using AFL's `afl-cmin` tool.

WOPT The *optimal minimum* corpus weighted by *file size*.

WMOPT The *weighted optimal minimum* corpus that takes into account edge frequencies. WMOPT attempts to *minimize* file sizes while *maximizing* an edge's frequency count. We originally tried to implement an "optimal `afl-cmin`" (i.e., minimizing file size while treating the same edge with different hit counts as distinct constraints), but EvalMaxSAT was unable to find a solution (after 6 h) for many targets. Maximizing the total hit count for a given edge is a compromise, and one that we hypothesize results in *deeper* program exploration.

We excluded Redqueen's uninformed, generic seed due to its poor performance in Section 3.1. We also explored an unweighted optimal minimum corpus, but found that MaxEvalSAT produced the same corpora as WOPT for all but three targets (libpng from Magma, libarchive from FTS, and the real-world poppler target). Thus, we exclude unweighted minimal corpora from our results.

We compare the performance of each seed selection approach across three measures:

Bug count: The ultimate aim of fuzzing is to find bugs in software. Thus, we use a direct bug count for comparing fuzzer effectiveness (as recommended in previous work [31, 37]). To this end, we perform manual triage for *all* crashes in the real-world targets, isolating the bugs that cause those crashes. This is in contrast to much of the existing literature [37, 40, 58, 69], which uses stack-hash deduplication to determine unique bugs from crashes—a technique known to both over- *and* under-count bugs [31, 37].

Bug survival time: As previously discussed, fuzzing is a highly stochastic process, and individual trials vary wildly in bug-finding performance. Following recommendations of Böhme and Falk [12], we statistically analyze and compare time-to-bug; or how long a bug "survives" in a fuzzing campaign. Following previous work [7, 31, 66], applying *survival analysis* to time-to-bug events allows us to handle *censoring*: individual trials where a given bug is not found. For each fuzzing campaign (i.e., set of 30 repeated 18 h trials with a particular fuzzer/corpus combination on a given target) we model a bug's *survival function*—the probability of a bug being found over time—using the Kaplan-Meier estimator [35]. Integrating this survival function with an upper-bound of 18 h gives a bug's *restricted mean survival time* (RMST) for a particular fuzzing campaign. Smaller bug survival times

[3] Per-target settings are available at https://osf.io/hz8em/.

indicate better performance. We report the RMST and 95 % confidence interval (CI) across each campaign. We also use the *log-rank test* [44] to statistically compare bug survival times. The log-rank test is computed under the null hypothesis that two corpora share the same survival function. Thus, we consider two corpora to have statistically equivalent bug survival times if the log-rank test's p-value > 0.05.

Code coverage: Coverage is often used to measure fuzzing effectiveness, as "*covering more code intuitively correlates with finding more bugs*" [37]. We use *region* coverage as reported by `llvm-cov`, generated by replaying each trial's fuzzing queue through a CoverageSanitizer-instrumented [65] target. Region coverage is the most granular of CoverageSanitizer's coverage metrics, is accurate (compared to AFL/AFL++'s notion of edge coverage, which is prone to hash collisions [25]), and allows us to normalize coverage across the two fuzzers.[4] We report both the mean and 95 % bootstrap CI of the percentage of code regions executed across each campaign. Region coverage is compared across corpora using the Mann-Whitney U-test; a p-value > 0.05 means that one fuzzer/corpus combination yields a statistically equivalent result compared to another. We use the Mann-Whitney U-test for the same reasons given by Klees et al. [37]; specifically, that it is *non-parametric* and makes no assumption on the underlying distribution.

5.2 Minimization (RQ1)

Table 2 shows the sizes of 14 collection corpora minimized across 21 target programs based on the code coverage measured by AFL. We reapplied the four corpus minimizers when fuzzing Magma with AFL++, as AFL++'s larger coverage map (250 KiB, compared to AFL's 64 KiB) theoretically results in a more fine-grained coverage view. Indeed, we saw small variations (up to 10 %) between the AFL and AFL++ minimized corpora, due to both the different-sized coverage maps and hash collisions that are inherent to AFL's method for computing edges.

Across both fuzzers, corpora produced by CMIN are significantly larger than that produced by MSET (mean 8× larger), WOPT (mean 9× larger), and WMOPT (mean 4× larger). This can be attributed to CMIN distinguishing seeds with different edge frequency counts: MSET and WOPT only look at edges executed, ignoring the number of times these edges are executed, while WMOPT maximizes an edge's frequency count. In comparison, MSET was only at most 37 seeds larger than WOPT (php-parser), and on average only five seeds larger than WOPT. WMOPT corpora were (on average) twice as large as WOPT corpora. Finally, the minimized php-exif corpora are notable because they discard 99 % of the full JPEG corpus, due to a lack of diverse EXIF data. The small minimized php-exif corpora demonstrate the importance of selecting a diverse range of initial inputs and minimizing large corpora.

In addition to generating smaller corpora, W[M]OPT incur lower runtime costs during minimization (compared to CMIN). We exclude the time required to trace the target and collect coverage data for each seed. WOPT's minimization times range from 12 ms (freetype2) to 23 min (libjpeg-turbo), with a mean minimization time of 141 s. WMOPT takes a similar amount of time: between 31 ms (freetype2) and 24 min (php-exif), with a mean minimization time of 143 s. In comparison, CMIN's minimization times range from 12 s to 130 min, with a mean time of 25 min. Despite CMIN's significantly slower minimization times, it is important to remember that (a) `afl-cmin` is a BASH script, while our optimal solver is written in C++, and (b) corpus minimization is a one-time upfront cost.

What ultimately matters is if the minimized corpora lead to better fuzzing outcomes. To this end, the following section discusses the bug-finding ability of the different corpus minimization techniques across our three benchmark suites (Magma, FTS, and a set of real-world targets) and two fuzzers (AFL and AFL++). We analyze these results with respect to the performance measures outlined in Section 5.1.5.

> OptiMin produces significantly smaller corpora compared to existing state-of-the-art corpus minimization tools, while also incurring lower runtime costs.

5.3 Bug Finding (RQ2)

Table 3 summarizes the bugs found in our Magma and FTS campaigns. Space constraints prevent us from providing the same level of detail for our real-world campaigns, so we instead summarize the seven CVEs assigned to us (for bugs found in these campaigns) in Table 4. In total, 78 (26 Magma, 15 FTS, and 33 real-world) bugs were found.

5.3.1 FTS Coverage Benchmarks. While the remainder of this section focuses on the bug-finding ability of each corpus, we first discuss six FTS "bugs" that are not actual bugs, but are instead code locations that the fuzzer must reach (marked with † in Table 3b). Notably, two of these locations are reached instantaneously (i.e., seeds in the fuzzing corpora reach the particular line of code without requiring any fuzzing) by most corpora *except* EMPTY. Naturally, EMPTY takes some time to reach the target locations, as AFL must construct valid inputs from "nothing". Nevertheless, EMPTY reaches four of the six target locations within two hours (on average). A libpng location and the freetype2 locations are never reached by EMPTY, because: (i) freetype2 requires a valid composite glyph, which EMPTY never synthesizes in the given timeframe, and (ii) libpng requires a specific chunk type (sRGB), which is difficult to synthesize without any knowledge of the PNG file format.

The only coverage benchmark that is not reached within minutes—libjpeg-turbo—is reliably reached by all corpora *except* FULL within the first five hours (on average) of each trial. The FULL corpus is highly unreliable on this target: it only reaches the target location in 10 % of trials, and when it does, it takes double the time of the other corpora. This results in a high survival time of 17.19 h.

5.3.2 The EMPTY Seed. Of the 41 (14 Magma, 8 FTS, and 19 real-world) bugs that EMPTY was able to find, it was the (equal) fastest to do so for 21 of these (5 Magma, 3 FTS, and 13 real-world). This result is particularly striking on SoX (both MP3 and WAV), where EMPTY found the most bugs with the lowest RMSTs (including three of the CVEs in Table 4). However, EMPTY also suffers from

[4] Google FuzzBench [27] also uses region coverage to normalize coverage across several fuzzers.

Table 2: Targets and corpora. Each corpus is summarized by the number of files contained within ("#") and total size ("S", in MiB unless stated otherwise). The smallest *minimized* corpus for both "#" and "S" are highlighted in green and blue, respectively.

				FULL		PROV		MSET		CMIN		WOPT		WMOPT	
	Target (driver)	Version	File type	#	S	#	S	#	S	#	S	#	S	#	S
Magma	libpng (libpng_read_fuzzer)		PNG	66,512	7,773.60	4	1.22 KiB	36	2.69	172	9.61	33	2.01	51	5.37
	libtiff (tiff_read_rgba_fuzzer)		TIFF	99,955	446.63	21	0.20	35	0.14	115	0.39	33	0.13	55	0.23
	libxml2 (libxml2_xml_reader_for_file_fuzzer)		XML	79,032	205.64	1,268	3.90	42	0.38	132	0.82	42	0.40	56	0.51
	php-exif (exif)		JPEG	120,000	222.86	60	0.26	2	0.01	2	0.01	2	0.01	2	0.01
	php-json (json)		JSON	19,978	76.46	55	4.23 KiB	17	0.72	212	4.29	17	0.85	30	1.91
	php-parser (parser)		PHP	75,777	224.51	2,934	0.92	606	1.94	2,229	11.43	569	1.71	1,187	7.86
	poppler (pdf_fuzzer)		PDF	99,986	6,085.07	392	18.39	237	28.11	2,273	119.27	222	26.70	510	66.89
Google FTS	freetype2	2017	TTF	466	35.50	2	2.83 KiB	43	5.40	246	20.92	37	5.04	53	6.95
	guetzli	2017-3-30	JPEG	120,000	222.86	2	544 B	17	0.04	463	0.60	13	0.03	51	0.10
	json	2017-02-12	JSON	19,978	76.46	1	14 B	17	0.95	149	2.56	16	1.21	27	1.77
	libarchive	2017-01-04	GZIP	108,558	850.64	1	500 B	41	1.05	180	2.80	40	0.94	57	1.52
	libjpeg-turbo	07-2017	JPEG	120,000	222.86	1	413 B	3	0.01	93	0.11	3	0.01	13	0.02
	libpng	1.2.56	PNG	66,512	7,773.60	1	1.23 KiB	22	1.91	107	4.05	19	1.71	28	2.85
	libxml2	2.9.2	XML	79,032	205.64	0	–	97	2.23	440	7.71	89	1.40	175	4.50
	pcre2	10.00	Regex	4,520	0.46	0	–	183	0.04	691	0.13	175	0.03	321	0.09
	re2	2014-12-09	Regex	4,520	0.46	0	–	56	0.01	155	0.01	55	0.01	84	0.01
	vorbis	2017-12-11	OGG	99,450	8,902.70	1	2.54 KiB	8	0.33	237	12.06	8	0.27	20	1.88
Real-world	freetype2 (char2svg)	2.5.3	TTF	466	35.50	–	–	23	3.04	73	8.68	23	3.02	33	4.70
	librsvg (rsvg-convert)	2.40.20	SVG	71,763	744.59	–	–	173	4.34	881	17.05	159	3.80	333	10.47
	libtiff (tiff2pdf)	4.0.9	TIFF	99,955	446.63	–	–	23	0.10	67	0.27	23	0.10	33	0.14
	libxml2 (xmllint)	2.9.0	XML	79,032	205.64	–	–	103	1.67	505	9.04	95	1.60	196	6.70
	poppler (pdftotext)	0.64.0	PDF	99,986	6,085.07	–	–	189	22.70	1,318	121.90	177	22.04	381	50.30
	sox-mp3 (sox)	14.4.2	MP3	99,691	4,094.22	–	–	9	0.17	137	3.75	6	0.30	15	0.64
	sox-wav (sox)	14.4.2	WAV	74,000	2,490.61	–	–	10	0.39	68	1.65	9	0.27	14	0.49

the highest "false negative" rate: it is the most likely corpus to miss a bug when one exists (as evident from the number of ⊤ entries in Table 3, the most of any corpus).

We hypothesize that the low RMST is due to the reduced search space when mutating the empty seed, but that the mutation engine is less likely to "get lucky" in generating a bug-inducing input when starting from nothing. Indeed, for the three FTS bugs where EMPTY statistically outperforms the other corpora (libjpeg-turbo, libpng's bug C, and libxml2's bug B), EMPTY finds the bug with the lowest number of mutations (on average, half the number of mutations compared to the other corpora on these three bugs) while also achieving a comparable (and sometimes, slightly slower) iteration rate than the other corpora (in particular, WOPT achieves a higher iteration rate than EMPTY on these three targets).

5.3.3 PROVided Seeds. The FTS PROV seeds are selected (by the FTS developers) based on their ability to trigger the target bug(s) within a few hours. For example, the json bug is "*usually found in about 5 minutes using the provided seed*" [26] (which our results confirm). Moreover, half of the FTS PROV seeds are *singleton* seeds. However, this is not indicative of fuzzing in practice, as (a) the location of bugs is unknown *a priori*, and (b) large seed sets are used in practice (per Section 3.2). Given the former, it is notable that the minimized corpora (CMIN, MSET, WOPT, and WMOPT) also successfully found the same FTS bugs that PROV found, and even outperform the PROV corpus in half of these targets (freetype2, libarchive, and libpng).

The PROV corpus is the best performer at finding bugs in Magma: it triggers the most bugs—21 of the 25 bugs found by all fuzzers—and achieves the (equal) lowest RMST for 15 of these bugs. Similarly to freetype2 and libpng in FTS, all three php-exif bugs were found without any mutation of the PROV seeds. Closer inspection of this corpus reveals why: PROV contains images that serve as regression tests for each of these three bugs (`bug77753.tiff`, `bug77563.jpg`, `bug77950.jpg`, corresponding to bugs MAE008, MAE014, and MAE016, respectively). These regression tests immediately trigger their respective bugs, but with Magma's *ideal sanitization*—"*in which triggering a bug immediately results in a crash*" [31]—disabled by default, these seeds do not cause a crash and hence are not excluded by AFL.[5]

5.3.4 Iteration Rates. Low iteration rates (i.e., the number of test-case executions per second) coupled with large corpora have a detrimental effect on a fuzzer's ability to find bugs. For example, with FULL, guetzli achieves mean iteration rates of 0.84 execs/s and 0.74 execs/s for AFL and AFL++, respectively. At the other end of the spectrum, EMPTY achieves mean iteration rates of 229.23 execs/s and 167 execs/s (for AFL and AFL++, respectively), while the minimized corpora achieve iteration rates between 2 and 5 execs/s. FULL's low iteration rate has a severe impact: both AFL and AFL++ fail to complete an initial pass over the 120,000 seeds in this corpus (in an 18 h trial), let alone perform any mutations and discover the bug. In comparison, the guetzli bug is found by all minimized corpora (CMIN, MSET, WOPT, and WMOPT) and PROV. We encounter similar results with poppler, where again neither AFL nor AFL++ can complete a full pass over the collection corpus (resulting in no bugs triggered).

We find that iteration rates vary significantly between fuzzers. For example, fuzzing Magma's libpng with AFL and EMPTY achieves a mean iteration rate of 693 execs/s, while AFL++ achieves a mean iteration rate of 2,508 execs/s. Conversely, fuzzing the same target with AFL and WOPT achieves a mean iteration rate of 4,575 execs/s, compared to AFL++ at 351 execs/s. These results correlate with the bug survival times in Table 3a (where AFL++ outperforms

[5] At the time of writing, this is a known issue flagged by the Magma developers, per https://github.com/HexHive/magma/issues/54.

Table 3: Bug-finding results, presented as the RMST with 95 % CI (in hours). Bugs that are never found have an RMST of ⊤ (to distinguish bugs with an RMST of 18 h). The best performing corpus (corpora if the bug survival times are statistically equivalent per the log-rank test) for each target (smaller is better) is highlighted in green.

(a) Magma bugs found by AFL and AFL++. We only report the RMST for bugs *triggered*. Bugs that are not triggered by any corpus are omitted (irrespective of whether the bug was reached or not). The php-json and php-parser targets are omitted because no bugs were found.

		FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	Bug	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
libpng	AAH001	⊤	⊤	16.83 ±4.30	⊤	16.96 ±3.82	⊤	17.95 ±0.33	⊤	⊤	⊤	17.54 ±2.83	⊤	17.47 ±3.25	⊤
	AAH003	1.93 ±0.07	8.67 ±3.45	⊤	0.12 ±0.05	0.0042 ±0.01	0.0050 ±0.01	0.01 ±0.01	0.01 ±0.01	0.02 ±0.01	0.10 ±0.08	0.01 ±0.01	0.04 ±0.01	0.01 ±0.01	0.34 ±0.13
	AAH007	⊤	⊤	⊤	17.96 ±0.26	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	AAH008	17.04 ±3.53	⊤	⊤	17.41 ±3.61	6.85 ±2.53	17.56 ±2.69	11.03 ±3.46	⊤	12.26 ±3.68	⊤	10.65 ±3.26	⊤	10.04 ±3.54	⊤
libtiff	AAH009	17.77 ±0.96	⊤	⊤	⊤	⊤	⊤	15.98 ±3.23	⊤	15.81 ±2.85	⊤	13.93 ±3.64	⊤	15.20 ±2.55	⊤
	AAH010	16.85 ±2.97	16.48 ±4.03	17.65 ±1.17	⊤	15.29 ±2.77	17.21 ±2.93	16.39 ±2.73	17.20 ±3.10	12.81 ±2.73	17.46 ±3.27	14.09 ±3.32	16.40 ±4.25	14.50 ±2.84	⊤
	AAH015	0.34 ±0.07	7.33 ±4.14	1.27 ±0.66	0.42 ±0.25	0.05 ±0.02	0.05 ±0.03	0.03 ±0.01	0.01 ±0.01	0.03 ±0.01	0.13 ±0.01	0.03 ±0.01	0.01 ±0.01	0.04 ±0.01	0.02 ±0.01
	AAH016	16.96 ±1.07	⊤	17.02 ±2.94	⊤	16.23 ±2.81	⊤	12.53 ±2.23	13.36 ±3.85	7.91 ±2.24	17.43 ±3.50	8.68 ±2.29	⊤	9.39 ±2.18	⊤
	AAH020	3.64 ±1.81	12.24 ±3.66	2.45 ±1.24	9.04 ±3.93	0.47 ±0.17	7.65 ±2.79	0.65 ±0.21	7.84 ±3.03	0.53 ±0.12	9.85 ±3.31	0.68 ±0.18	9.33 ±3.04	0.70 ±0.14	9.83 ±3.11
	AAH022	0.76 ±0.15	11.34 ±3.97	1.87 ±1.19	0.73 ±0.55	0.88 ±0.36	6.46 ±3.23	1.38 ±0.78	3.08 ±1.68	1.56 ±0.72	5.16 ±2.31	1.45 ±0.35	2.60 ±1.16	1.66 ±0.51	1.39 ±1.13
libxml2	AAH024	17.87 ±0.82	⊤	⊤	⊤	17.42 ±2.49	⊤	⊤	⊤	17.25 ±2.10	⊤	⊤	⊤	⊤	⊤
	AAH026	8.69 ±2.00	14.94 ±2.89	⊤	⊤	14.93 ±4.32	14.74 ±3.69	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	AAH027	0.69 ±0.12	0.85 ±0.08	0.01 ±0.01	0.02 ±0.01	0.05 ±0.01	0.02 ±0.01	0.77 ±0.30	0.08 ±0.05	0.61 ±0.29	0.15 ±0.08	1.21 ±0.41	0.77 ±0.26	0.63 ±0.27	0.80 ±0.34
	AAH032	15.99 ±3.58	16.42 ±3.26	17.94 ±0.36	15.00 ±2.75	16.44 ±3.64	17.09 ±2.93	17.55 ±1.69	17.50 ±2.19	15.33 ±3.22	17.07 ±2.58	15.51 ±3.55	16.91 ±2.39	14.31 ±3.16	16.89 ±2.01
	AAH037	5.96 ±2.38	7.77 ±1.73	⊤	⊤	3.07 ±1.26	11.22 ±2.82	9.37 ±1.96	9.98 ±1.92	7.76 ±1.91	10.29 ±2.42	6.64 ±1.66	11.49 ±2.66	5.50 ±1.16	9.53 ±2.01
	AAH041	0.63 ±0.12	1.32 ±0.11	16.88 ±2.66	2.60 ±1.22	0.06 ±0.02	0.38 ±0.15	0.09 ±0.01	0.09 ±0.01	0.11 ±0.03	0.07 ±0.01	0.11 ±0.01	0.07 ±0.01	0.09 ±0.01	0.08 ±0.01
php-exif	MAE008	⊤	⊤	⊤	⊤	0.00 ±0.01	16.20 ±5.28	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	MAE014	⊤	⊤	⊤	⊤	0.00 ±0.01	0.00 ±0.01	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	MAE016	11.79 ±2.09	7.76 ±3.78	8.68 ±2.32	0.05 ±0.01	0.00 ±0.01	13.74 ±2.40	9.66 ±2.67	0.05 ±0.01	7.40 ±2.16	0.06 ±0.01	8.43 ±1.97	0.06 ±0.02	9.76 ±2.77	0.05 ±0.01
poppler	AAH043	⊤	⊤	⊤	⊤	17.43 ±3.45	⊤	17.86 ±0.82	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	AAH047	⊤	⊤	⊤	⊤	⊤	⊤	17.46 ±3.28	⊤	17.18 ±3.01	⊤	16.93 ±2.92	⊤	15.40 ±3.38	⊤
	AAH052	⊤	⊤	⊤	⊤	0.83 ±0.64	7.66 ±4.15	4.06 ±0.74	17.42 ±3.54	5.54 ±0.62	17.65 ±1.13	4.51 ±0.48	⊤	8.07 ±1.49	⊤
	JCH207	⊤	⊤	0.0042 ±0.01	0.0066 ±0.01	0.08 ±0.01	0.09 ±0.02	0.14 ±0.01	2.19 ±0.14	1.11 ±0.04	3.13 ±0.17	0.14 ±0.01	2.48 ±0.17	0.30 ±0.01	2.51 ±0.16
	JCH209	⊤	⊤	⊤	⊤	6.17 ±2.62	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	JCH212	⊤	⊤	⊤	⊤	⊤	⊤	17.61 ±2.37	⊤	17.67 ±2.00	⊤	⊤	⊤	⊤	⊤
	JCH214	⊤	⊤	⊤	⊤	⊤	⊤	17.79 ±0.57	⊤	⊤	⊤	17.90 ±0.48	⊤	⊤	⊤

AFL on the EMPTY seed, while AFL outperforms AFL++ with the WOPT corpus), suggesting that higher iteration rates contribute to a fuzzer's bug-finding ability (and at the very least, allow a fuzzer more quickly to discard inputs that are not worth exploring).

When fuzzing with non-empty corpora, AFL achieves a higher iteration rate than AFL++. This is likely due to a combination of (a) more complex target instrumentation (where more of this instrumentation is being exercised with valid inputs), and (b) a coverage map that is ∼ 4× larger than AFL's (which has L2 cache implications). These results further reinforce the need for corpus minimization when starting with a large collection corpus, particularly as fuzzer complexity increases.

(b) Bug-finding results (cont.). FTS bugs found by AFL and AFL++. IDs are derived from the order in which the bugs are presented in the target's README (from the FTS repo). Bugs marked with † denote benchmarks that attempt to verify that the fuzzer can reach a known location. Results with – indicate that the FTS does not contain seeds for that target (see Table 2), and so we ignore it. The vorbis target is omitted because none of its three bugs were found.

		FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	Bug	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
freetype2	A†	0.00 ±0.00	0.00 ±0.00	⊤	⊤	6.78 ±2.12	6.10 ±2.73	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
guetzli	A	⊤	⊤	⊤	⊤	10.25 ±2.33	17.48 ±1.38	15.88 ±2.09	17.45 ±3.37	17.78 ±1.07	⊤	15.45 ±2.82	⊤	16.92 ±2.83	⊤
json	A	⊤	⊤	⊤	16.41 ±4.24	0.09 ±0.08	0.28 ±0.27	17.64 ±2.17	⊤	16.93 ±3.92	17.57 ±2.60	17.68 ±1.43	⊤	17.90 ±0.62	⊤
libarchive	A	⊤	⊤	⊤	15.31 ±1.76	⊤	9.08 ±0.90	4.44 ±0.21	11.86 ±1.74	9.39 ±1.42	11.97 ±1.90	12.50 ±0.50	6.78 ±0.44	7.38 ±0.05	13.50 ±1.75
libjpeg-turbo	A†	17.19 ±2.37	⊤	1.92 ±0.45	10.43 ±2.73	3.00 ±0.95	14.14 ±2.47	3.36 ±0.98	16.57 ±2.70	3.82 ±1.19	15.62 ±2.75	4.71 ±1.52	16.91 ±2.70	3.68 ±0.93	15.70 ±2.11
libpng	A†	2.41 ±0.01	0.0043 ±0.0020	0.03 ±0.01	0.11 ±0.02	0.08 ±0.08	0.21 ±0.03	0.0051 ±0.0029	0.09 ±0.01	0.0072 ±0.0036	0.0041 ±0.0027	0.0025 ±0.0017	0.0049 ±0.0032	0.0038 ±0.0025	0.08 ±0.01
	B†	0.00 ±0.00	0.00 ±0.00	⊤	0.33 ±0.04	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	C†	2.42 ±0.02	0.0089 ±0.0092	0.0003 ±0.0004	0.07 ±0.14	0.0008 ±0.0007	2.47 ±0.85	0.0046 ±0.0029	0.0049 ±0.0025	0.0070 ±0.0035	0.0054 ±0.0033	0.0023 ±0.0017	0.0062 ±0.0034	0.0036 ±0.0022	0.0040 ±0.0032
libxml2	A	4.46 ±0.32	17.84 ±0.58	⊤	16.46 ±2.48	–	–	0.54 ±0.06	3.10 ±0.09	0.79 ±0.12	4.72 ±0.96	0.66 ±0.07	4.70 ±0.51	0.85 ±0.23	7.81 ±2.23
	B	16.46 ±1.27	⊤	5.32 ±1.22	13.43 ±2.26	–	–	13.35 ±1.95	⊤	10.57 ±2.20	17.85 ±0.91	8.93 ±1.81	17.95 ±0.28	14.46 ±2.09	⊤
	C	⊤	⊤	⊤	⊤	–	–	⊤	⊤	17.53 ±2.89	⊤	⊤	⊤	⊤	⊤
pcre2	A	2.57 ±0.39	4.56 ±0.36	1.88 ±0.25	2.70 ±0.44	–	–	2.15 ±0.24	5.43 ±0.74	2.07 ±0.23	3.10 ±0.46	1.59 ±0.16	4.83 ±0.47	1.39 ±0.18	3.34 ±0.44
	B	2.04 ±0.73	5.34 ±1.58	3.25 ±0.70	5.71 ±0.88	–	–	2.01 ±0.68	3.83 ±0.73	2.29 ±0.65	3.98 ±1.36	1.42 ±0.42	3.47 ±0.96	1.61 ±0.72	3.39 ±1.22
re2	A†	0.82 ±0.16	2.91 ±0.52	1.72 ±0.31	9.72 ±1.61	–	–	0.74 ±0.52	3.30 ±0.53	0.52 ±0.12	3.85 ±0.70	0.50 ±0.16	7.52 ±2.88	0.42 ±0.09	3.35 ±1.14
	B	16.20 ±3.83	16.41 ±2.49	17.66 ±1.48	17.69 ±1.87	–	–	11.52 ±2.70	15.69 ±2.53	11.85 ±3.22	16.97 ±2.35	12.19 ±3.07	17.08 ±2.62	12.36 ±3.18	15.91 ±2.55

Table 4: New bugs assigned CVEs in the real-world targets.

Target	CVE	Description
libtiff	2019-14973	Elision of integer overflow check by compiler
poppler	2019-12293	Heap buffer overread
sox	2019-8354	Integer overflow causes improper heap allocation
	2019-8355	Integer overflow causes improper heap allocation
	2019-8356	Stack buffer bounds violation
	2019-8357	Integer overflow causes failed memory allocation
	2019-13590	Integer overflow causes failed memory allocation

5.3.5 Comparison to Previous Magma Evaluations. Our results improve on those originally presented by Hazimeh et al. [31]. Nine of the bugs triggered during our 18 h trials (AAH007, AAH009, AAH024, AAH026, AAH043, AAH047, JCH209, JCH212, and JCH214) were *never* triggered by AFL/AFL++ in the original 24 h campaigns. Furthermore, two of these bugs (poppler's AAH047 and JCH214) were never found by *any* of the seven fuzzers originally evaluated by Hazimeh et al. [31]. These two bugs were only found by the distilled corpora fuzzed with AFL and never with FULL, EMPTY, PROV, or AFL++. This is significant because over 200,000 CPU-hours were spent fuzzing Magma targets (across many 24 h and 7 d trials).

5.3.6 CVEs. Our real-world fuzzing campaigns led to the assignment of seven CVEs across three targets, summarized in Table 4 (our campaigns uncovered another 26 bugs, but these were already under investigation). Of these bugs, libtiff's CVE-2019-14973 is particularly interesting: discovered by all corpora, but found the fastest and most reliably by EMPTY (with an RMST of 6.02 h), this bug is only evident because we build our real-world targets for 32-bit x86. The libtiff maintainers report that the undefined behavior at the root of this bug does not manifest on 64-bit targets.

Our campaigns also uncovered an already-known uncontrolled resource consumption bug in libtiff (CVE-2018-5784). This bug is caused by an infinite loop in the TIFF image directory linked list and is again found most frequently by EMPTY (11 out of 30 trials, resulting in an RMST of 12.37 h). In comparison, this bug is never found by FULL and MSET and only once by the other corpora. Notably, the initial EMPTY seed does not contain any image file directories, while all of the TIFF files in our minimized corpora do. We suspect that AFL's mutations break existing directory structures (leading to parser failures), whereas EMPTY is able to construct a (malformed) directory list from scratch. These mutations eventually lead to a loop in the list, causing the uncontrolled resource consumption.

Seed selection has a significant impact on a fuzzer's ability to find bugs. Both AFL and AFL++ perform better when bootstrapped with a minimized corpus, although the exact minimization tool is inconsequential. While both AFL and AFL++ find a similar number of bugs, AFL is generally faster to do so (and with less variance in bug-finding times).

5.4 Code Coverage (RQ3)

Table 5 summarizes the coverage achieved over all Magma and FTS campaigns. For the majority of targets, EMPTY explores less code than the other corpora: on average, half as much code (although this difference decreases slightly when fuzzing with AFL++). In particular, on targets that accept highly-structured input formats (e.g., libxml2 and poppler), EMPTY explores less than half as much of the program's code compared to the four minimized corpora. EMPTY's results improve slightly when fuzzing with AFL++, likely due to the additional CmpLog instrumentation (reflecting our `readelf` results in Section 3.1). However, even with this improvement, EMPTY's performance remains inferior to any of the other seed selection approaches.

After an 18 h trial, little distinguishes the code coverage achieved by the non-EMPTY corpora, and once again the four minimized corpora with AFL produced the best results. Curiously, the coverage gains that AFL++ saw when fuzzing `readelf` with CMIN (Section 3.1) did not manifest in any of the five Magma targets: both AFL and AFL++ achieved similar levels of code coverage.

Seed selection has a significant impact on a fuzzer's ability to expand code coverage. When fuzzing with the empty seed, more-advanced fuzzers (such as AFL++) are able to cover more code. However, this advantage all but disappears when bootstrapping the fuzzer with a minimized corpus, as faster iteration rates become more critical. The exact minimzation tool remains inconsequential.

5.5 Discussion

Selecting a corpus minimization tool. We evaluated three corpus minimization tools: MINSET, `afl-cmin`, and our own OPTIMIN. Our results do not reveal a "best" minimization tool; while minimized corpora sizes varied markedly between tools, stochastic fuzzing variability means that this ultimately has no statistically-significant impact on fuzzing outcomes (with respect to both bug finding and code coverage). However, a minimized corpus is always better due to the faster iteration rate, and while our results show that this may not necessarily find more bugs in a given trial, it still means that the fuzzer is able to more quickly discard inputs that are not worth exploring. We therefore recommend the adoption of OPTIMIN, given the considerably smaller corpora that it produces.

When to use the empty seed. While our results demonstrate that corpus minimization achieves best results, there were nine occasions (three in each of the Magma, FTS, and real-world benchmarks) where EMPTY performed as well as or better than the minimized corpora. These occasions correspond to when coverage is at its lowest, suggesting that these are shallow bugs. Thus, where possible, we recommend that an additional campaign with the empty seed be conducted to quickly weed out shallow bugs. However, when conducting industrial-scale fuzzing campaigns, the empty seed should *never* be used.

Corpus minimization as lossy compression. Previous work [59, 69] demonstrates that different coverage metrics can affect fuzzing results in practice. Similarly, corpus minimization can also make use of coverage metrics that are not solely based on code coverage (or, in AFL's case, approximate edge coverage). Corpus minimization based solely on code coverage is effectively a form of *lossy compression* [22]: program states may be discarded if they do not expand code coverage. Indeed, we saw this in Section 5.2, where the AFL/AFL++ corpus sizes differed due to different-sized coverage maps. We leave it to future work to explore how corpus minimization generalizes to different coverage metrics.

Generalizing to other fuzzers. We limit our experiments to two coverage-guided, mutational greybox fuzzers: AFL and AFL++. We selected AFL because it is widely evaluated and deployed, while AFL++ is an updated and maintained version of AFL that incorporates broad improvements from recent fuzzing research and regularly outperforms *all* other fuzzers on Google's FuzzBench [27]. While it is unclear how our results might generalize to other fuzzers (e.g., honggfuzz [64] and libFuzzer [61], both of which provide corpus minimization capabilities), we believe that our AFL++ results—which demonstrate how seed selection practices impact a range of recent advances in fuzzing research—are generalizable to other mutation-based greybox fuzzers. We leave it to future work to confirm this.

6 CONCLUSIONS

We present here, to the best of our knowledge, the first in-depth analysis of the impact that seed selection has on mutation-based greybox fuzzing. Our premise is that the choice of fuzzing corpus is a critical decision—often overlooked—made before a fuzzing campaign begins. Our results provide ample confirmation of this criticality, and demonstrate that fuzzing outcomes can vary significantly depending on the initial seeds used to bootstrap the fuzzer.

Intuitively, bootstrapping a fuzzing campaign with a single, small, *representative* seed would seem to be a fair baseline for comparison of fuzzers. After all, fuzzing is already a highly-stochastic process, so simplifying the initial seed choice seems uncontroversial. However, we argue—and our results show—that this can lead to unfair performance comparisons and high variance in results. Furthermore, it is not representative of how fuzzing is performed in the "real world", where large seed corpora are typically used. We therefore recommend that large, diverse corpora be collected and minimized (e.g., with OPTIMIN, which produces significantly smaller corpora than the current state-of-the-art) to maximize fuzzing yield. Seed selection is a critical step that must be considered prior to launching any fuzzing campaign.

Table 5: Code coverage, expressed as the mean region coverage with 95 % bootstrap CI. The best performing corpus (corpora if the code coverages are statistically equivalent per the Mann-Whitney U-test) for each target (larger is better) is highlighted in green.

(a) Magma code coverage with AFL and AFL++.

	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*
libpng	28.71 ±0.09	25.56 ±1.51	15.26 ±0.06	19.10 ±0.97	25.19 ±0.29	26.55 ±0.74	30.00 ±0.05	29.53 ±0.06	29.98 ±0.05	29.60 ±0.06	30.03 ±0.04	29.39 ±0.12	29.08 ±0.05	28.81 ±0.14
libtiff	44.37 ±0.39	44.76 ±0.30	35.10 ±2.23	27.41 ±2.05	44.77 ±0.25	41.72 ±0.43	45.40 ±0.23	44.45 ±0.22	46.09 ±0.23	44.38 ±0.39	45.70 ±0.26	44.38 ±0.26	45.86 ±0.29	43.90 ±0.32
libxml2	21.10 ±0.29	20.39 ±0.31	10.60 ±0.29	14.99 ±0.44	22.47 ±0.22	22.74 ±0.37	19.47 ±0.14	19.84 ±0.20	20.75 ±0.21	20.64 ±0.32	19.53 ±0.22	19.14 ±0.10	20.06 ±0.14	19.07 ±0.07
php-exif	2.24 ±0.04	2.34 ±0.01	2.28 ±0.04	2.36 ±0.00	2.37 ±0.00	2.37 ±0.00	2.25 ±0.04	2.36 ±0.00	2.30 ±0.03	2.36 ±0.00	2.29 ±0.03	2.36 ±0.00	2.26 ±0.04	2.36 ±0.00
poppler	35.96 ±0.00	35.95 ±0.00	1.49 ±0.00	1.76 ±0.01	41.12 ±0.05	38.35 ±0.06	41.40 ±0.06	36.74 ±0.30	41.13 ±0.06	37.92 ±0.32	41.30 ±0.07	36.69 ±0.27	41.44 ±0.07	36.85 ±0.33

(b) FTS code coverage with AFL and AFL++.

	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*	*AFL*	*AFL++*
freetype2	48.01 ±0.12	44.65 ±0.41	17.31 ±0.16	33.85 ±0.78	34.99 ±0.17	45.41 ±0.59	47.58 ±0.14	44.84 ±0.34	47.84 ±0.15	44.11 ±0.42	47.82 ±0.20	43.24 ±0.24	47.73 ±0.16	45.12 ±0.21
guetzli	67.34 ±0.00	67.34 ±0.00	31.05 ±0.14	30.31 ±0.14	75.55 ±0.12	73.61 ±0.08	72.95 ±0.14	69.84 ±0.10	73.12 ±0.11	70.76 ±0.07	72.93 ±0.10	70.16 ±0.07	72.79 ±0.11	69.92 ±0.08
json	90.29 ±0.08	90.87 ±0.11	90.00 ±0.34	90.18 ±0.31	90.32 ±0.26	90.74 ±0.22	90.60 ±0.06	90.96 ±0.06	90.62 ±0.07	91.14 ±0.08	90.82 ±0.10	90.96 ±0.07	90.79 ±0.08	91.05 ±0.06
libarchive	17.14 ±0.12	16.67 ±0.47	17.84 ±0.46	23.09 ±0.77	18.04 ±0.63	24.63 ±0.68	18.51 ±0.19	20.14 ±0.39	18.54 ±0.22	22.18 ±0.24	18.65 ±0.10	21.55 ±0.21	18.36 ±0.20	21.61 ±0.27
libjeg-turbo	16.31 ±0.30	15.38 ±0.08	18.56 ±0.09	18.01 ±0.07	18.84 ±0.24	18.13 ±0.07	20.53 ±0.11	19.24 ±0.32	20.54 ±0.11	19.50 ±0.24	20.47 ±0.12	19.25 ±0.30	20.41 ±0.11	19.01 ±0.38
libpng	32.81 ±0.12	34.87 ±0.10	19.30 ±0.00	29.66 ±0.23	25.40 ±0.00	33.83 ±0.21	34.92 ±0.09	36.62 ±0.18	35.09 ±0.06	36.82 ±0.13	34.97 ±0.06	36.95 ±0.15	34.94 ±0.07	36.63 ±0.12
libxml2	14.90 ±0.09	14.49 ±0.03	6.77 ±0.09	8.12 ±0.49	–	–	15.81 ±0.15	15.08 ±0.12	15.91 ±0.13	14.95 ±0.03	16.01 ±0.02	15.06 ±0.15	15.76 ±0.15	15.29 ±0.18
pcre2	60.81 ±0.16	63.97 ±0.14	59.65 ±0.15	63.13 ±0.20	–	–	60.94 ±0.16	63.21 ±0.22	61.04 ±0.22	63.64 ±0.24	61.09 ±0.23	63.03 ±0.16	61.00 ±0.16	62.92 ±0.22
re2	59.00 ±0.07	58.96 ±0.06	59.26 ±0.16	58.13 ±0.53	–	–	59.05 ±0.06	59.03 ±0.05	59.08 ±0.04	59.00 ±0.06	59.10 ±0.05	57.40 ±0.83	59.05 ±0.08	59.00 ±0.07

ACKNOWLEDGMENTS

The authors would like to thank: Arlen Cox, for his initial ideas on applying SAT solvers to corpus minimization; Felix Friedlander and Maggi Sebastian, for building the collection corpora and triaging crashes; and Liam Hayes and Jonathan Milford, for their early work on MoonLight. This work was supported by the Defence Science and Technology Group Next Generation Technologies Fund (Cyber) program via the Data61 Collaborative Research Project *Advanced Program Analysis for Software Vulnerability Discovery and Mitigation*, and has also been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868).

REFERENCES

[1] 2020. Kenney. https://www.kenney.nl/
[2] 2020. The Motion Monkey. https://www.themotionmonkey.co.uk/
[3] 2020. Open Game Art. https://opengameart.org/
[4] 2020. Regular Expression Library. http://regexlib.com
[5] Humberto Abdelnur, Radu State, Obes Jorge Lucangeli, and Olivier Festor. 2010. *Spectral Fuzzing: Evaluation & Feedback*. Research Report RR-7193. INRIA. https://hal.inria.fr/inria-00452015
[6] 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
[7] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 1–10. https://doi.org/10.1145/1985793.1985795
[8] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars

[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[10] Florent Avellaneda. 2020. A short description of the solver EvalMaxSAT. In *MaxSAT Evaluations*. http://florent.avellaneda.free.fr/dl/EvalMaxSAT.pdf

[11] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *USENIX Security Symposium (SEC)*. 1985–2002. https://www.usenix.org/system/files/sec19-blazytko.pdf

[12] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 713–724. https://doi.org/10.1145/3368089.3409729

[13] Marcel Böhme, Valentin J.M. Manès, , and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 678–689. https://doi.org/10.1145/3368089.3409748

[14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043. https://doi.org/10.1145/2976749.2978428

[16] Oliver Chang, Abhishek Arya, Kostya Serebryany, and Josh Armour. 2017. OSS-Fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html

[17] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *USENIX Security Symposium (SEC)*. 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[18] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2095–2108. https://doi.org/10.1145/3243734.3243849

[19] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy (S&P)*. 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[20] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. 633–645. https://doi.org/10.1145/3321705.3329828

[21] Deja Vu Security. [n.d.]. PeachMinset. http://community.peachfuzzer.com/minset.html

[22] Brandon Falk. 2021. Fuzzing: Corpus Minimization. https://youtu.be/947b0lgyvJs

[23] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

[24] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium (SEC)*. 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[25] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[26] Google. 2016. Google Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite

[27] Google. 2020. FuzzBench. https://google.github.io/fuzzbench/

[28] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 72–82. https://doi.org/10.1145/2568225.2568278

[29] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. 2017. QuickFuzz testing for fun and profit. *Journal of Systems and Software* 134 (Dec. 2017), 340–354. https://doi.org/10.1016/j.jss.2017.09.018

[30] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*. https://www.ndss-symposium.org/ndss-paper/codealchemist-semantics-aware-code-generation-to-find-vulnerabilities-in-javascript-engines/

[31] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (March 2021). https://doi.org/10.1145/3428334

[32] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A General Framework and Tool for Supporting Test-Suite Minimization. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2009.5070541

[33] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security Symposium (SEC)*. 2271–2287. https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[34] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *USENIX Annual Technical Conference (ATC)*. 249–263. https://www.usenix.org/conference/atc20/presentation/jeon

[35] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *J. Amer. Statist. Assoc.* 53, 282 (June 1958). https://doi.org/10.2307/2281868

[36] Richard M. Karp. 2011. Computational Complexity of Combinatorial and Graph-Theoretic Problems. In *Theoretical Computer Science*, F. Preparata (Ed.). CIME Summer Schools, Vol. 68. Springer, 97–184. https://doi.org/10.1007/978-3-642-11120-4_3

[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[38] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 627–637. https://doi.org/10.1145/3106237.3106295

[40] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 533–544. https://doi.org/10.1145/3338906.3338975

[41] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-Criteria Test-Suite Minimization with Integer Nonlinear Programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 1039–1049. https://doi.org/10.1145/3180155.3180174

[42] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium (SEC)*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[43] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 1024–1036. https://doi.org/10.1145/3377811.3380421

[44] Nathan Mantel. 1966. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemotherapy Reports* 50, 3 (1966), 163–170.

[45] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 548–560. https://doi.org/10.1145/3314221.3314651

[46] Charlie Miller. 2008. Fuzz By Number: More Data About Fuzzing Than You Ever Wanted To Know. In *CanSecWest*. https://cansecwest.com/csw08/csw08-miller.pdf

[47] Mozilla. 2015. Dharma: A Generation-based, Context-Free Grammar Fuzzer. https://blog.mozilla.org/security/2015/06/29/dharma/

[48] Mozilla. 2018. Introducing the ASan Nightly Project. https://blog.mozilla.org/security/2018/07/19/introducing-the-asan-nightly-project/

[49] Mozilla. 2020. Fuzzing—Test Samples. https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html

[50] Ben Nagy. 2010. Prospecting for Rootite: More Code Coverage, More Bugs, Less Wasted Effort. In *Ruxcon*. https://2010.ruxcon.org.au/presentations/#pfr

[51] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (S&P)*. 787–802. https://doi.org/10.1109/ISTAS48451.2019.8937885

[52] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 2020. The Industrial Age of Hacking. In *USENIX Security Symposium (SEC)*. 1129–1146. https://www.usenix.org/conference/usenixsecurity20/presentation/nosco

[53] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium (SEC)*. 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[54] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *ACM SIGSOFT International*

Symposium on Software Testing and Analysis (ISSTA). 329–340. https://doi.org/10.1145/3293882.3330576

[55] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 174–1749. https://doi.org/10.1145/3360600

[56] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *USENIX Security Symposium (SEC)*. 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[57] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. 2018. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *Journal on Cybercrime & Digital Investigations* 3, 1 (2018). https://doi.org/10.18464/cybin.v3i1.17

[58] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium (SEC)*. 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[59] Christopher Salls, Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Exploring Abstraction Functions in Fuzzing. In *IEEE Conference on Communications and Network Security (CNS)*. 1–9. https://doi.org/10.1109/CNS48642.2020.9162273

[60] Scrapinghub. 2020. Scrapy. https://scrapy.org/

[61] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development (SecDev)*. 157. https://doi.org/10.1109/SecDev.2016.043

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*. 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[63] JHU/APL Staff. 2019. Assembled Labeled Library for Static Analysis Research (ALLSTAR) Dataset. https://allstar.jhuapl.edu/

[64] Robert Swiecki. 2016. honggfuzz. http://honggfuzz.com/

[65] The Clang Team. 2020. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[66] Jonas Benedict Wagner. 2017. *Elastic Program Transformations Automatically Optimizing the Reliability/Performance Trade-off in Systems Software*. Ph.D. Dissertation. EPFL. http://infoscience.epfl.ch/record/228899

[67] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*. 579–594. https://doi.org/10.1109/SP.2017.23

[68] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

[69] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[70] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/not-all-coverage-measurements-are-equal-fuzzing-by-coverage-accounting-for-input-prioritization/

[71] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2313–2328. https://doi.org/10.1145/3133956.3134046

[72] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security Symposium (SEC)*. 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[73] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium (SEC)*. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[74] Michał Zalewski. 2015. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/

[75] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *USENIX Security Symposium (SEC)*. 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong