SAST-Guided Grey-Box Fuzzing

Stephan Lipp
Technical University of Munich
Germany

Severin Kacianka
Technical University of Munich
Germany

Alexander Pretschner
Technical University of Munich
Germany

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

ABSTRACT

Fuzzing is an automated testing technique that generates random inputs to provoke program crashes, indicating security vulnerabilities. While coverage-based fuzzing is most widespread in the field, it becomes exponentially and thus prohibitively expensive when searching for new security bugs; valuable fuzzing resources are wasted on non-security-critical code regions as code coverage treats all regions as equally important/vulnerable. Moreover, existing defect-guided fuzzing approaches struggle to accurately pinpoint problematic regions, flagging too many locations to be considered during fuzzing and thus run into similar issues.

In this paper, we leverage static application security testing (SAST) tools to direct fuzzing toward potentially vulnerable regions, effectively reducing the code scope to be tested. Further, we introduce a target scheduling mechanism that dynamically disables likely false-positive and unreachable SAST findings/fuzzing targets to mitigate their impact on fuzzing performance. We implement our approach in the fuzzer SASTFuzz and evaluate it on 19 open-source programs against the coverage-based fuzzer AFL as well as two defect-guided fuzzers, TortoiseFuzz and ParmeSan (4.6 CPU-years worth of fuzzing). The results show that SASTFuzz outperforms the three fuzzers, finding three times more distinct bugs than AFL and TortoiseFuzz, the best-performing defect-guided fuzzer in our study. Thereby, SASTFuzz exclusively detects 10 of the 80 newly discovered bugs, twice as many as AFL and TortoiseFuzz. Moreover, during the 48-hour campaigns, SASTFuzz discovers bugs on average 2.2 to 5 hours faster than TortoiseFuzz, with slightly less time savings when compared to AFL.

1 INTRODUCTION

Context. Fuzz testing, *aka* fuzzing [42, 46], is an automated testing technique that has proven extremely effective in detecting software bugs [50], especially in C/C++ programs. Fuzzing is straightforward: Generate millions of random inputs (mechanized by fuzzers), feed them into the system under test, and then monitor the program execution for crashes. Crashes that enable malicious code execution, data leakage, or trigger denial-of-service are considered security vulnerabilities. For instance, Google uses fuzzing extensively, utilizing massive compute resources (over 100 k CPU cores) on its OSS-Fuzz platform [56] to enhance the security of major open-source projects. Over eight years, OSS-Fuzz has identified over 46 k bugs in more than 1,000 codebases [7].

State-of-Practice and Problem. The most widely used fuzzing approach is coverage-based fuzzing [31, 42, 45, 56], as implemented in the well-known fuzzer AFL [1]. There, the goal is to cover as much code as possible, as a faulty line must be reached in the first place (with the correct input data) to trigger the underlying (security) bug. However, regular code coverage considers all code regions as *equally* important/vulnerable [65], wasting valuable fuzzing resources on non-security-critical regions due to the sparsity of software bugs [40, 47, 75]. This is also shown by an empirical study by Böhme and Falk [12], highlighting a worrying trend: An *exponential* increase in resources (CPUs or time, respectively) is required by state-of-the-art (coverage-based) grey-box fuzzers to achieve only linear growth in vulnerability discovery. This makes coverage-based fuzzing prohibitively expensive to find more complex bugs, thus motivating the need for more effective fuzzing approaches that counteract this exponential trend.

Recent defect-guided approaches like in AFLChurn [73], TortoiseFuzz [65], and ParmeSan [49] focus—instead of all—only on specific, likely vulnerable code regions, such as regions recently or frequently changed, containing memory-modifying instructions, or instrumented by sanitizers [60] like AddressSanitizer (ASan) [57]. For AFLChurn, however, this means it is bound to incremental fuzzing of programs with *access* to the version control system history. TortoiseFuzz and ParmeSan, on the other hand, run into similar issues as coverage-based fuzzing because their *imprecise*[1] defect heuristics still require extensive code coverage. Therefore, more accurate identification of potentially vulnerable code is required for defect-guided fuzzing to work effectively.

Solution Approach. To address the above limitations, we *combine* static application security testing (SAST) [11] and directed grey-box fuzzing [13] by utilizing multiple static analyzers whose findings (*i.e.*, flagged lines) steer the fuzzing process towards the problematic code locations. In our experiments, we found that state-of-the-art SAST tools quickly (in under 10 minutes on the median) locate ~75% of the (potentially) vulnerable functions while flagging only 30% of the functions[2] (half the rate issued by ASan), thus *effectively reducing* the code scope to be targeted during fuzzing. Furthermore, our approach includes a *target scheduling mechanism* that dynamically disables likely false-positive and unreachable SAST findings/fuzzing targets from the distance computations[3] performed during directed fuzzing. This mitigates their impact on the fuzzing performance, as the fuzzer shifts execution more quickly to other, more likely

[1] In a case study with nine open-source programs [38], we found that ASan instruments/flags on average more than 75% of the functions (with only 1% of them vulnerable). This number is even larger when considering all memory-modifying instructions as in TortoiseFuzz.

[2] Our experiments confirm similar results in related studies [20, 21, 37]

[3] Most directed fuzzers use the distance (*i.e.*, the average number of control-flow edges) between the fuzz input's execution trace and the target code locations to evaluate its proximity, thereby keeping distance-reducing inputs in the queue for further mutation.

vulnerable/reachable target locations. Additionally, we *dynamically* switch between directed and coverage-based fuzzing depending on the reachability of the enabled targets, thereby allowing the fuzzer to discover vulnerable code overlooked by the analyzers (false negatives).

Empirical Evaluation. We implemented our SAST-guided fuzzing approach in a grey-box fuzzer called SASTFuzz. In a comprehensive empirical evaluation, we assess SASTFuzz's performance relative to that of the popular coverage-based fuzzer AFL and two state-of-the-art defect-guided fuzzers, TortoiseFuzz and ParmeSan. This evaluation comprises ~4.6 CPU-years worth of fuzzing across 19 open-source programs (in total 11.5 k+ functions) from various application domains, revealing *80 previously unknown (security) bugs* that we confidentially reported to corresponding developers. Our comparison thereby focuses on the total/distinct number of new bugs found and the time the fuzzers require to catch them. Furthermore, in a post-bug analysis, we investigate whether the bugs exclusively detected by SASTFuzz result from our SAST-based approach and are thus no mere random discoveries.

Contributions. This paper presents the following *contributions*:

- ★ *We propose and implement a methodology* for combining SAST and (directed) fuzzing, thereby compensating for false-positive and unreachable SAST findings via a novel dynamic target scheduling mechanism (see § 3).
- ★ *We perform an in-depth analysis*, showing that our scheduling mechanism leads to more target locations being reached than directed fuzzing with static targets (see § 4.2.2).
- ★ *We show in a large-scale empirical evaluation* that our SAST-based fuzzing approach detects more and different previously unknown (security) bugs than coverage-based fuzzing and existing defect-guided approaches (see §§ 5.2.1 and 5.2.2). Also, compared to the latter, our approach finds the bugs several hours faster (see § 5.2.3).
- ★ *We will release our SAST-guided grey-box fuzzer* SASTFuzz as open-source software (OSS), as well as all evaluation data and analysis scripts upon acceptance to foster open science.
- ★ *We found and reported 80 new (security) bugs*, helping to improve the state of OSS security.

2 MOTIVATION

Unlike coverage-based fuzzing, current defect-guided approaches focus on certain code locations considered more bug-prone than others. However, they rely on imprecise defect heuristics that often produce many fuzzing targets, potentially resulting in ineffective campaigns. Therefore, we advocate using SAST tools to accurately identify targets, presenting hereafter insights into their runtime overhead and bug-finding effectiveness.

Runtime Overhead. Figure 1 shows the SAST durations on 24 subject programs when running the five static analyzers from Table 2 in parallel. The analysis takes less than 15 minutes on most subjects, with somewhat longer durations of about 27 to 45 minutes on Pocketlang, Libxml2, and libsodium. The biggest outlier is OpenSSL, one of our largest benchmark programs, where Infer takes ~5 hours to examine the 166,814 lines of code (LoC)[4], an unusually long duration given that Infer requires considerably less time for other subjects with similar LoC. Nonetheless, the median duration is less than 10 minutes, suggesting *minimal* additional time overhead for our SAST-guided fuzzing approach.

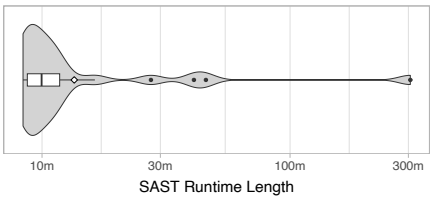


Figure 1: SAST runtimes (in minutes) for 24 subject programs when running five state-of-the-art analyzers in parallel. The diamond (◇) indicates the arithmetic mean.

Table 1: SAST detection rates.

Subject	Marked Funcs.	Recall	Precision
Libpng	0.256	0.778	0.064
Libsndfile	0.364	0.929	0.029
LibTIFF	0.238	0.385	0.025
Libxml2	0.389	0.857	0.011
OpenSSL	0.218	0.682	0.005
Mean	0.293	0.726	0.027

Bug-Finding Effectiveness. Table 1 shows the SAST performance[5] on the five programs from Table 3, where a vulnerable function is considered detected (true positive) if the analyzers flag one or more of its lines. On Libpng, Libsndfile, and Libxml2, the SAST tools reliably identify problematic functions, with detection rates ranging from 0.78 to 0.93. There, they flag between 26% and 39% of the functions, thus significantly reducing the amount of code to be fuzzed. For OpenSSL, the detection rate slightly drops to 0.68 while flagging only 22% of the functions. In the case of LibTIFF, the analyzers overlooked more than half of the vulnerable functions, deviating from the otherwise high detection rates. To mitigate the impact of such false negatives, we cleverly switch between coverage-based and directed fuzzing (see § 3.2.4). Furthermore, to address the high false-positive incidence (average precision of 0.027), we systematically deactivate SAST findings/fuzzing targets (see § 3.2.2).

In summary, multiple SAST tools in combination *effectively* locate (potentially) vulnerable functions with an average recall of 0.73 while flagging less than 30% of the functions. This narrows down the fuzzing efforts to roughly one-third of the functions (compared to coverage-based fuzzing) and more than half of the ~75% of ASan-instrumented/flagged functions targeted by sanitizer-guided fuzzing.

[4]We recommend performing an entire scan of the codebase once for larger programs, followed by incremental SAST runs focusing on the code changes.
[5]The precision values must be interpreted cautiously, as it is unclear whether undiscovered vulnerabilities still exist in these programs.

Table 2: Selected SAST tools.

Tool	Version	Techniques	Checks
Flawfinder	2.0.19	Code-pattern matching	Fixed rule-set
Semgrep	1.24.0	Code & AST-pattern matching; data-flow analysis	YAML-based rules
Infer	1.1.0	(Semi-)Formal reasoning	Fixed rule-set
CodeQL	2.12.0	Control & data-flow analysis; AST-based analysis	Datalog-based queries
Clang-SA	12.0.0	Symbolic execution	C++-based checkers

3 APPROACH

3.1 SAST-Based Target Acquisition

3.1.1 Selected SAST Tools. We employ five diverse and freely available C/C++ SAST tools (see Table 2) to (1) maximize bug-finding effectiveness, (2) support multiple different vulnerability types, and (3) discern the likelihood of code vulnerability via analyzer consensus (see § 3.1.3). Our selection ranges from code and abstract syntax tree (AST) pattern-matching analyzers (Flawfinder [5] and Semgrep [8]) over more complex tools that utilize comprehensive control and data-flow analyses (Infer [6] and CodeQL [4]) up to Clang-SA [2] which symbolically executes the system under test (SUT) to uncover bugs. These SAST tools are popular in the field (*cf.* their GitHub stars [15]) and have also been shown effective in finding vulnerabilities in prior research [20, 21, 37].

3.1.2 SAST Findings Grouping. We group the SAST-tool findings at the basic blocks (BBs) level. Specifically, all flagged lines within the same BB are considered a single target location (the block's starting line number). This is a valid reduction as all (flagged) lines will be executed as soon as the target BB is reached.

Definition 1 (Target BB Set)**.** We define the set of target basic blocks B_t as those blocks in the SUT's inter-procedural control-flow graph, which contain at least one code line flagged as (potentially) vulnerable by the SAST tools.

Unlike most existing work [13, 19, 33, 64], which aggregate target locations during instrumentation, we initiate this process in an earlier static analysis phase. This way, we markedly accelerate the subsequent distance calculations, a known *bottleneck* in current directed fuzzing [41].

3.1.3 Target Basic Block Weighting. In addition to grouping, we weight the identified basic blocks based on their vulnerability probability. The intuition is that a higher weight, later called *vulnerability score*, correlates with a higher likelihood of the target containing a vulnerability. This then enables (1) an enhanced target prioritization that selects target blocks with elevated vulnerability scores first, and (2) a systematic resource allocation that dedicates more fuzzing time to targets with higher scores.

Code Granularity. We use function-level SAST information for weighting the target BBs. This is because the available ground-truth datasets, essential for assessing the weighting's validity in § 4.2.1, only include code locations where vulnerabilities originate but not where they manifest, which is typically where SAST tools flag them [37]. However, since these locations typically occur within the same function [37], we consider the function level a valid granularity for approximating SAST vulnerability detection. Accordingly, all target blocks within the same function receive the same vulnerability score.

SAST-Based Defect Heuristics. Inspired by the references [39, 52] that use SAST-based machine-learning features in their defect prediction models, we utilize the following *defect heuristics* to determine the vulnerability scores.

Heuristic 1 (SAST-Tool Consensus). For a target basic block $b_t \in B_t$ in function *func*, we define SAST-tool consensus as the ratio of analyzers that flag *func*, *i.e.*, the number of SAST-tools that flag one or more lines in *func* divided by the number of all tools that analyzed *func*.

Our first heuristic operates on the premise that—similar to the *majority voting* principle—*consensus* among various static analyzers regarding a flagged function increases the possibility that the function is vulnerable.

Heuristic 2 (SAST-Flag Density). For a target basic block $b_t \in B_t$ in function *func*, we define SAST-flag density as the ratio of lines flagged by one or more analyzers in *func*, *i.e.*, the number of flagged lines in *func* divided by the lines of code in *func*.

Our second heuristic is based on the hypothesis that a higher number of flagged lines within a function indicates a higher *density* of security-critical statements and thus increases the probability that the function contains a vulnerability.

3.2 Directed Fuzzing with Dynamic Targets

Existing directed fuzzers keep *all* target blocks active during the entire campaign without fully deactivating targets, even after being thoroughly fuzzed. However, targets sourced from SAST tools contain false positives that would then still be targeted during directed fuzzing, negatively affecting the fuzzer's directedness. To address this, we introduce a *dynamic target scheduling mechanism* in our SAST-guided fuzzer SASTFuzz, which disables targets (from the distance computations) once they are deemed irrelevant.

3.2.1 Distance Measure. Generally, the distance serves as a fitness function, quantifying how far a fuzz input is from reaching the targets. SASTFuzz is built upon WindRanger [19], which improves AFLGo [13] by leveraging (1) data-flow information, as well as (2) more accurate control-flow information. For (1), WindRanger accounts for the complexity of the target-path conditions, *i.e.*, it mutates inputs that execute less complex conditions more frequently than those with a shorter overall distance value (to the same path) but a more complex path. For (2), WindRanger implements a more precise distance metric that relies on so-called *deviation basic blocks*, statically identified before fuzzing.

Definition 2 (Deviation BB Set)**.** For a set of target basic blocks (BBs) B_t, Du *et al.* [19] define the set of deviation BBs B_d in the SUT's inter-procedural control-flow graph (iCFG) as the first blocks in the program execution paths where one or more potential sub-paths deviate from reaching any $b_t \in B_t$.

Algorithm 1: Dynamic target BB scheduling.

```
   Data: B_t, numIvalInputs ∈ ℕ
   Result: Updated B_t
1  foreach b_t ∈ B_t do
2      if b_t.state ∈ {active, paused} then
3          reqTargetBBInputHits ← calcReqTargetBBInputHits(b_t,
             B_t, numIvalInputs);
4          if (reqTargetBBInputHits − b_t.inputHits) ≤ 0 then
5              b_t.state ← finished;
6          else
7              if coveredInPrevInterval(b_t) then
8                  b_t.state ← active;
9                  b_t.ivalSkips ← 0;
10                 b_t.ivalSkipsPrev ← 1;
11             else
12                 if b_t.ivalSkips = 0 then
13                     b_t.state ← paused;
14                     b_t.ivalSkips ← b_t.ivalSkipsPrev;
15                     b_t.ivalSkipsPrev ← b_t.ivalSkipsPrev + 1;
16                 else if (b_t.ivalSkips − 1) = 0 then
17                     b_t.state ← active;
18                     b_t.ivalSkips ← 0;
19                 else
20                     b_t.ivalSkips ← b_t.ivalSkips − 1;
```

Formally, the refined distance measure can be described via the function $\texttt{distance} : B_d \times B_t \to \mathbb{N} \cup \{\text{-1}\}$ as follows:

$$\texttt{distance}(b_d, b_t) := \begin{cases} \text{Shortest} & \text{if } b_t \text{ is statically reach-} \\ \text{\#iCFG-path,} & \text{able from } b_d, \\ \text{-1,} & \text{otherwise.} \end{cases} \quad (1)$$

Equation (1) better reflects the likelihood of a fuzz input reaching the target BBs, resulting in a less biased prioritization: Inputs with longer (average) distances can still get prioritized as long as they get closer to the targets.

3.2.2 Dynamic Target Scheduling. To mitigate the impact of false-positive and unreachable fuzzing targets, we suggest to (1) *permanently* deactivate targets from the distance computations (and thus from being targeted again) once the fuzz inputs have sufficiently often covered them and (2) *temporarily* suspend targets that remain unreached for increasingly longer periods. Concretely, during fuzzing, each target basic block $b_t \in B_t$ is either *activated* (included in the distance computations; initial state), *paused* (temporarily suspended), or *finished* (permanently excluded), stored in the variable b_t.state. To temporarily suspend targets, we use a scheduling mechanism that en- and disables unreached target blocks using b_t.{ivalSkips, ivalSkipsPrev}. To permanently exclude targets, we compute a *hit-count threshold*, derived from the target BB's (relative) vulnerability score (b_t.score) and its input hit-count (b_t.inputHits), which must be exceeded.

Target Deactivation & Scheduling. Algorithm 1 details our scheduling strategy, which is executed in *intervals* during fuzzing to (re-)evaluate the state of the non-finished targets. As parameters, it takes the target BB set B_t and the number of fuzz inputs generated during the last interval *numIvalInputs*. The algorithm itself consists of two parts: the first part is associated with deactivating sufficiently fuzzed targets (Lines 3–5) and the second with scheduling potentially unreachable ones (Lines 7–20).

Algorithm 2: Computation of required target BB hits.

```
1  Function calcReqTargetBBInputHits(targetBB ∈ B_t, B_t,
     numIvalInputs ∈ ℕ)
2      relTargetBBScore ← targetBB.score /
         sum({b_t.score | b_t ∈ B_t ∧ b_t.state ≠ finished});
3      return numIvalInputs ∗ relTargetBBScore;
```

Algorithm 3: Distance computation to target BBs.

```
1  Function calcTargetBBsDistance(b_d ∈ B_d, B_t)
2      distance, n ← 0;
3      foreach b_t ∈ B_t do
4          targetDist ← distance(b_d, b_t);
5          if b_t.state = active and targetDist > 0 then
6              distance ← distance + (1/targetDist);
7              n ← n + 1;
8      return (n/distance) if n > 0 else -1;
```

Part 1: While iterating over all non-finished target blocks b_t, Line 3 employs the function `calcReqTargetBBInputHits` from Algorithm 2 to establish the hit-count threshold (*i.e.*, b_t's required input hits before marked finished). In that function, Line 2 computes b_t's vulnerability score relative to all active/paused targets. Line 3 then determines the threshold value by multiplying the computed score with the total number of generated interval fuzz inputs. So the higher a target BB's vulnerability score, the more input executions are required to finish it (*i.e.*, more thorough fuzzing).

Back to Algorithm 1: In case b_t has been sufficiently often covered, *i.e.*, the if-condition in Line 4 is satisfied, b_t will be deactivated by shifting it from a paused or active state to the finished. Formally, this transition looks in our target scheduling as $\Phi(b_t) = \langle \ldots, a \textbf{ or } p \to f, \ldots, f \rangle$, where $\Phi(\cdot)$ denotes the state trace of a target block across the intervals.

Part 2: Should b_t remain insufficiently fuzzed yet was covered by at least one input in the last interval (indicated by the function `coveredInPrevInterval`), we directly active it for the next interval in Line 8, *i.e.*, $\Phi(b_t) = \langle \ldots, a \textbf{ or } p \to a, \ldots \rangle$. Furthermore, we reset the variables involved in pausing targets due to b_t's current reachability. However, if b_t remains uncovered for several intervals (or throughout the fuzzing campaign), the scheduling in Lines 12–20 looks as follows: $\Phi(b_t) = \langle a, p, a, p, p, a, p, p, p, a, \ldots \rangle$—we alternatively activate and pause b_t, thereby *incrementing* the number of pausing intervals. This way, we account for likely unreachable target blocks by gradually reducing their impact on the distance calculation until they are reached by chance or the campaign ends.

Target Distance Calculation & Prioritization. We use the function in Algorithm 3 to compute the harmonic mean distance[6] of a

[6] Using the harmonic mean mitigates the impact of large outlier distances.

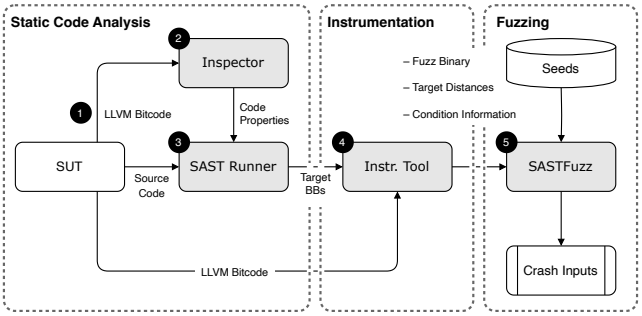


Figure 2: High-level system architecture of SASTFuzz.

given deviation basic block to all active target blocks. The mean distances for all deviation blocks are thereby stored in a *look-up table*, which is then used to calculate the distance achieved for an executed fuzz input (arithmetic mean over the covered deviation BBs). To maintain low computational overhead, this table is updated *only* after intervals that alter the state of one or more targets. Furthermore, after each distance update, we sort the fuzzer queue so that inputs with shorter distances are fuzzed first, further enhancing the chances of reaching targets.

3.2.3 Increasing Interval Length. Starting with a user-specified *initial interval length*, we increase the interval *logarithmically* throughout the fuzzing campaign. Hence, progressively more fuzzing time will be spent on hard-to-reach target blocks deeply located in the SUT. Furthermore, since deeper targets typically correspond to more intricate vulnerabilities [10], extended intervals lead to more input hits required to mark such targets as finished, thereby enhancing the chances of triggering complex vulnerabilities.

Generally, the length of the (initial) interval significantly influences the fuzzing performance. Longer intervals lead to more thorough fuzzing of single target blocks but may cover fewer overall, whereas shorter intervals cover more targets at the risk of overlooking bugs due to premature target deactivation. Section 4.2.2 explores this trade-off in more detail to establish an optimal initial interval length.

3.2.4 Dynamic Exploration & Exploitation Switching. Directed grey-box fuzzing involves two phases, *exploration* and *exploitation* [13]. The exploration phase aims to maximize the overall code coverage to expand the seed corpus and avoid local optima. In contrast, the exploitation phase steers the fuzzing process toward the specified target basic block. WindRanger thereby further improves AFLGo by introducing a dynamic switching strategy based on the execution frequency of the deviation blocks. Building upon this concept, SASTFuzz switches into exploration mode when all targets are finished or paused, enabling the discovery of new and/or vulnerable code locations missed by the SAST tools. The process returns to exploitation intervals when a new target block is reached or a previously paused one resumes activity. Once all targets are marked finished, SASTFuzz reactivates those with a vulnerability score of at least 0.5, focusing again on the more critical ones.

Table 3: Subjects for SASTFuzz investigation.

Subject	Commit	LoC	# Functions	# Vuln. Funcs.
Libpng	a37d483	9,860	430	9
Libsndfile	86c9f9e	27,037	1,224	14
LibTIFF	c145a6c	19,234	844	13
Libxml2	ec6e3ef	81,360	2,863	14
OpenSSL	3bd5319	166,814	13,724	22
Total		304,305	19,085	72

3.3 Application in Practice

Figure 2 shows the system architecture and execution steps of SASTFuzz, spanning three phases: static code analysis, program instrumentation, and (directed) fuzzing.

Step ❶ involves extracting the LLVM bitcode file from the system under test (SUT). Step ❷ utilizes this extracted program representation via the Inspector component to perform a whole-program analysis, identifying code properties such as the line ranges of all contained functions and basic blocks (BBs). Following this, step ❸ launches the SAST Runner, which runs the static analyzers (in parallel) on the SUT to identify potentially vulnerable code locations. Moreover, it uses the provided code properties to group the SAST findings into target blocks, which then receive the determined vulnerability score. Step ❹ shifts from static code analysis to program instrumentation, turning the bitcode file into a coverage-instrumented and thus fuzzable binary. Additionally, it outputs the deviation-to-target BB distances, as well as the path complexity information, both required to fuzz the instrumented SUT in step ❺.

4 SASTFUZZ CONFIGURATION

This section empirically examines (1) the validity of our SAST-based defect heuristic for assigning vulnerability scores and (2) the effectiveness of varying (initial) interval lengths in our target scheduling mechanism versus directed fuzzing with static targets. The insights will then guide the *configuration* of SASTFuzz for its comparative evaluation with other fuzzers in § 5.

4.1 Setup

4.1.1 Subject Programs. For this investigation, we randomly selected five subject programs, with a total of 72 vulnerable functions, from the Magma ground-truth benchmark [24] (see Table 3). Magma is characterized by a diverse collection of well-documented vulnerabilities, utilizing a technique called *bug front-porting*, by which previously found vulnerabilities (*i.e.*, validated CVE reports) are reinserted into newer versions of the same program.

4.1.2 Seeds & Campaign Length. All fuzzing-related experiments were conducted with a non-empty seed corpus provided by Magma. Each fuzzing campaign was carried out over 24 hours and repeated five times to mitigate the effects of the inherent randomness in fuzzing, which we believe is sufficient to depict the impact of the studied parameters.

4.1.3 Infrastructure. All experiments in this work were performed on a machine equipped with an AMD EPYC(TM) 7742 processor,

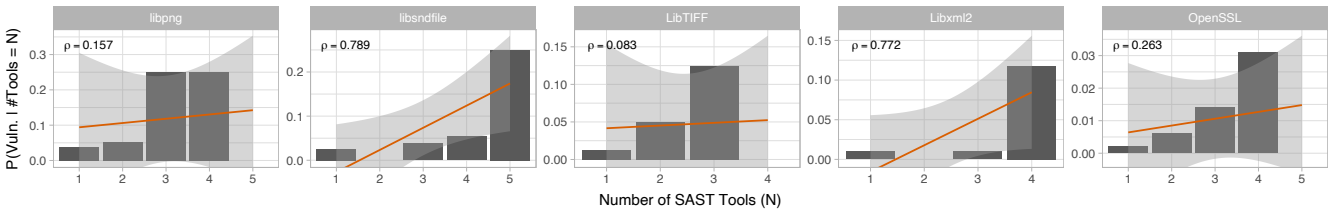


Figure 3: Relationship (Pearson correlation ρ) between the number of different SAST tools flagging a function and the (conditional) probability of this function being vulnerable (SAST-tool consensus heuristic).

which features 128 logical cores operating at a speed of 3.4 GHz. The system contains 995 GB of main memory and uses Ubuntu 20.04 (64-bit). Moreover, to measure the SAST-tool overhead in § 2 under realistic conditions, we restricted the analyzers' hardware access to 24 cores and 64 GB of memory.

4.2 Results

4.2.1 Validity of SAST-Based Defect Heuristics. To assess the validity of Heuristics 1 and 2, we calculated the (conditional) probability of a function being vulnerable based on different degrees of SAST-tool consensus and flag density observed in our subject programs. We hypothesize that a higher heuristic value *correlates* with an increased detection probability, measured by Pearson's ρ [55].

For Heuristic 1, Fig. 3 shows a positive trend line for almost all subjects, indicating that a function is more likely vulnerable if multiple analyzers flag it. In particular, we observe a strong positive correlation (ρ = ~0.8) in Libsndfile and Libxml2. In the remaining programs, we see a sharp positive trend up to $N = 3, 4$, followed by a decline due to zero probabilities in the last N values. Consequently, this results in a weak to moderate correlation ($\rho = 0.16, 0.26$) in OpenSSL and Libpng, and a negligible one ($\rho = 0.08$) in LibTIFF, despite strong positive trends up to $N = 3$.

In contrast, the same analysis Heuristic 2 revealed no positive correlation between the density of flagged lines in a function and the likelihood of this function being vulnerable in all five programs. This renders the SAST-flag density heuristic invalid, at least at the function level.

> Summary (SAST-Based Defect Heuristics). Functions flagged by multiple static analyzers (irrespective of the density of flagged lines) tend to be more likely vulnerable, making us exclusively use the SAST-tool consensus heuristic to determine the vulnerability score of fuzzing target blocks.

4.2.2 Optimal Initial Interval Length. We start SASTFuzz with different initial interval lengths between 15 minutes and 8 hours and then measure the number of target blocks (as identified by the SAST tools) covered in our subject programs (see Fig. 4a). Furthermore, we record the number of unique bugs triggered to assess the *performance gain* in terms of both target coverage (Vargha-Delaney $\hat{A}_{12}$ [9])[7] and additional bugs found (ΔB; see Fig. 4b)

[7]For instance, $\hat{A}_{12} = 0.5$ implies no performance difference between our approach and WindRanger's, while deviations from 0.5 indicate varying effect sizes.

The table shows that shorter initial intervals typically lead to an enhanced target coverage. However, while the campaigns with 15-minute starting intervals achieve the largest performance gain ($\hat{A}_{12} = 0.9$), they result in less thorough fuzzing (due to early target deactivation). Specifically, they find only one bug (in Libsndfile), also discovered by WindRanger ($\Delta B = 0$). Also, campaigns with 4- and 8-hour initial interval lengths detect no additional bugs because they reach fewer flagged, vulnerable code regions. In contrast, starting SASTFuzz with 30-minute, 1-hour, and 2-hour intervals results in three unique bugs found in Libsndfile, two more than WindRanger ($\Delta B = +2$).

> Summary (Initial Interval Length). An initial interval length of one hour provides the best balance between target coverage and bug exploitation. Using this configuration, SASTFuzz covers significantly more target basic block than WindRanger while also finding more distinct (security) bugs.

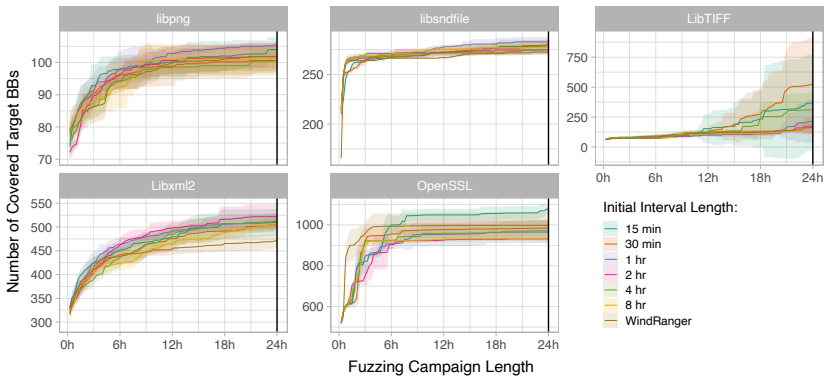
4.3 Discussion

SAST incurs a relatively low overhead, about 15 minutes, which can be further decreased by running analyzers incrementally on code changes, relevant, *e.g.*, for continuous integration (CI) pipelines with tight time constraints. The selected SAST tools thereby effectively identify (potentially) vulnerable functions while, on average, flagging less than one-third of the functions. This rate drops even further at the basic block level, the granularity of our fuzzing targets, thus greatly reducing the code scope for fuzzing. Additionally, our SAST-tool consensus heuristic enables assessing the vulnerability probability of the target blocks, which in turn enhances the target prioritization and resource allocation during fuzzing.

Regarding directed fuzzing, our dynamic target scheduling mechanism surpasses prevailing static-target directed fuzzing, like that used in WindRanger, in effectively reaching the target blocks. Here, we discovered that a one-hour initial interval length strikes an optimal balance (number of targets reached versus bugs triggered). However, for CI environments, shorter intervals (*e.g.*, 5 to 15 minutes) might be better suited to maximize target coverage for the limited time budget.

5 EVALUATION

Using the configuration identified in the previous section, we empirically evaluate SASTFuzz's bug-finding performance compared to common coverage-based and defect-guided fuzzers.



(a) Target coverage of SASTFuzz (with different initial intervals) and WindRanger; the error bands show the standard deviation around the arithmetic mean (solid lines).

Rank	Init. Length	$\hat{A}_{12}$	ΔB
1	15 min	0.900	0
2	1 hr	0.692	+ 2
3	30 min	0.635	+ 2
4	2 hr	0.598	0
5	4 hr	0.590	0
6	8 hr	0.515	0

(b) Performance gain(s) in target ($\hat{A}_{12}$) and bug coverage (ΔB) across the studied initial intervals.

Figure 4: Performance difference between SASTFuzz and WindRanger when using different initial intervals.

Table 4: Subset of evaluation subjects where bugs could be found by at least one of the four fuzzers.

Subject	Commit	LoC	# BBs	# Crashes	# Bugs
Discount	a096c1a	3,329	3,210	1,488	1
MIR	928e28f	17,496	13,668	9,731	44
MyHTML	4f98bb1	6,933	4,203	818	2
Parson	b800e9d	1,739	1,369	667	6
Pocketlang	e316d53	8,110	6,014	10,261	26
raylib	557aeff	19,840	10,655	10	1
Total		57,447	39,119	22,975	80

5.1 Setup

5.1.1 Research Questions. To assess if our SAST-based fuzzing approach surpasses coverage-based and other defect-guided approaches, we ask the following *research questions*:

- **RQ.1** *Bug-Finding Effectiveness.* How many new (security-critical) software bugs does SASTFuzz find compared to the baseline fuzzers?
- **RQ.2** *Different Bugs Found.* What is the overlap/difference between the bugs exclusively found by SASTFuzz and those only detected by the baseline fuzzers?
- **RQ.3** *Bug-Finding Efficiency.* How fast does SASTFuzz discover the same bugs also detected by the baseline fuzzers?
- **RQ.4** *Post-Bug Analysis.* To what extent is our SAST-based approach responsible for the exclusively detected/missed bugs by SASTFuzz?

5.1.2 Subject Programs. We evaluate the fuzzers on the *latest versions* (as of August 2023) of 19 diverse open-source programs with *unknown* bugs. This prevents *overfitting* (adapting tools to benchmark bugs) and *survivorship bias* (under/overrating tools that found the original bugs), thus offering a more realistic assessment of the fuzzers' bug-finding effectiveness [14, 28]. The subjects were thereby randomly selected from various application domains:

- **Data Parsing:** Discount, Jansson, jq, MyHTML, Parson, PDFio, MDB Tools
- **Multimedia Processing:** FLAC, Gifsicle, LibGD, libwebp, OpenJPEG, Opus
- **Network & Security:** libmodbus, libsodium
- **Programming:** MIR, Pocketlang, Zydis
- **Gaming:** raylib

Table 4 summarizes the programs in which bugs were found by at least one of the employed fuzzers. Each discovery is a strong indicator of a new security-critical bug and has been confidentially reported to the corresponding developers, with some bugs already confirmed as security vulnerabilities.

5.1.3 Baseline Fuzzers. We select AFL [1] (version 2.56b) as our baseline coverage-based fuzzer, which Google, among others, has used for years to find thousands of security vulnerabilities in critical open-source software. As SASTFuzz (over WindRanger and AFLGo) is based on AFL, it allows us to compare *methodologies* rather than technical implementations [54] and, hence, how SAST-guided (directed) fuzzing generally performs against coverage-based fuzzing. Furthermore, we compare SASTFuzz against two other defect-guided fuzzers, TortoiseFuzz [65] (v0.1) and ParmeSan [49] (no version specified), both published in top-tier research conferences, with their source code—unlike most directed fuzzers—available on GitHub. Both fuzzers have demonstrated superior bug-finding capabilities over many state-of-the-art fuzzers [49, 65], making them ideal baseline candidates.

5.1.4 Seeds & Campaign Length. For the fuzzer evaluation, we use non-empty seed corpora provided by various GitHub repositories[8]. We run each fuzzing campaign for 48 hours and repeat it ten times, allowing us to compute average scores that account for the randomness in fuzzing [28].

5.1.5 Bug Deduplication. In total, the launched fuzzing campaigns triggered 22,975 program crashes. As manual deduplication of these many crashes is impractical, we employ ClusterFuzz's method [3] to approximate unique (security) bugs: Two crashes are deemed to

[8] https://github.com/google/oss-fuzz
https://github.com/dvyukov/go-fuzz-corpus
https://github.com/strongcourage/fuzzing-corpus

Table 5: Arithmetic means μ with standard deviations (in grey) and total bug counts Σ achieved by SASTFuzz and the baseline fuzzers.

	SASTFuzz		AFL		TortoiseFuzz		ParmeSan	
Subject	μ	Σ	μ	Σ	μ	Σ	μ	Σ
Discount	1.0 ± 0.0	1	0.8 ± 0.4	1	1.0 ± 0.0	1	0.0 ± 0.0	0
MIR*	21.0 ± 2.9	37	19.6 ± 2.6	32	19.2 ± 3.0	33	–	–
MyHTML	1.1 ± 0.3	2	1.0 ± 0.0	1	1.2 ± 0.4	2	0.0 ± 0.0	0
Parson	5.8 ± 0.4	6	6.0 ± 0.0	6	5.9 ± 0.3	6	0.0 ± 0.0	0
Pocketlang	12.1 ± 2.0	23	11.0 ± 1.4	19	10.4 ± 1.1	16	3.6 ± 1.0	7
raylib	0.3 ± 0.5	1	0.7 ± 0.5	1	0.0 ± 0.0	0	0.0 ± 0.0	0
Total		70		60		58		7

* ParmeSan aborts prematurely when executed on this program.

expose the same vulnerability if they share the same vulnerability type (issued by ASan) and the identical top five stack frames in their execution trace. Ultimately, this process yielded 80 unique, previously unknown bugs (see Table 4), which we inspected again to confirm the validity of this approximation.

5.2 Results

Despite troubleshooting efforts, ParmeSan aborts prematurely when executed on MIR, so we have no evaluation results for this subject.

5.2.1 RQ.1: Bug-Finding Effectiveness. Table 5 presents both the arithmetic means (including the standard deviations) and the total number of newly detected (security) bugs (over 10 fuzzing campaign repetitions) by SASTFuzz and the baseline fuzzers AFL, TortoiseFuzz, and ParmeSan. Regarding the total bug count, SASTFuzz consistently uncovers more unique bugs than the baseline fuzzers, never performing less effectively than the next-best fuzzer. For instance, on Pocketlang, our fuzzer finds four more bugs than AFL, seven more than TortoiseFuzz, and sixteen more than ParmeSan. Concerning the arithmetic means, SASTFuzz shows an equal or superior average performance in Discount, MIR, and Pocketlang. However, on MyHTML, Parson, and raylib, SASTFuzz shows marginally lower means than the most effective baseline fuzzer despite identifying the same number of unique bugs. This indicates a slightly larger variation/fluctuation in SASTFuzz's performance between the campaign repetitions.

> Answer (RQ.1). SASTFuzz is more effective than the selected baseline fuzzers, detecting across all subject programs 10 more unique, previously unknown (security) bugs than AFL, 12 more than TortoiseFuzz, and 63 more than ParmeSan, despite slightly larger performance fluctuations between the fuzzing campaign repetitions.

5.2.2 RQ.2: Different Bugs Found. Figure 5 illustrates the overlap/differences in unique (security) bugs found by SASTFuzz, AFL, TortoiseFuzz, and ParmeSan. The Venn diagram reveals that SASTFuzz discovers 10 distinct bugs that all baseline fuzzers miss, while AFL and TortoiseFuzz exclusively detect only 4 and 5, *i.e.*, 2.5 and 2 times fewer bugs, respectively; ParmeSan fails to identify any bugs

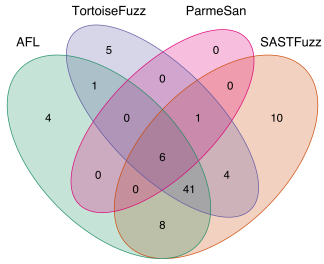


Figure 5: Intersection/divergence of the unique (security) bugs found by the four different fuzzers.

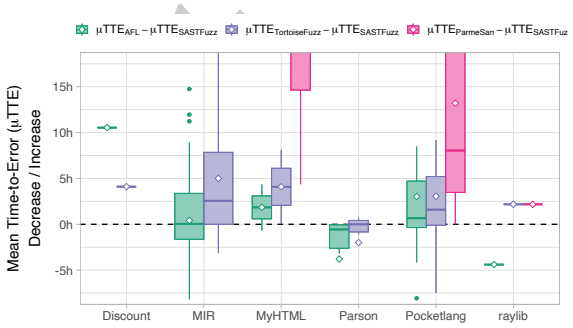


Figure 6: Difference in mean time-to-error (μTTE) per unique (security) bug between SASTFuzz and the baseline fuzzers. The diamonds ($\diamond$) indicate the arithmetic means.

not found by the other fuzzers. Compared to AFL, SASTFuzz exclusively identifies 15 bugs, three times more than those solely found by AFL (5 bugs); relative to TortoiseFuzz, 18 unique bugs, again a threefold increase compared to the 6 only found by TortoiseFuzz. Also note that SASTFuzz and TortoiseFuzz jointly detect 76 out of 80 bugs, which accounts for 95% of all bugs found.

> Answer (RQ.2). SASTFuzz effectively detects different (security) bugs, uncovering 10 distinct bugs missed by AFL, TortoiseFuzz, and ParmeSan. Also, SASTFuzz outperforms both AFL and TortoiseFuzz, exclusively detecting 15 and 18 bugs, respectively, three times as many as either fuzzer. Compared to ParmeSan, our fuzzer identifies 63 additional bugs.

5.2.3 RQ.3: Bug-Finding Efficiency. Figure 6 displays the difference in mean time-to-error (μTTE) per identified (security) bug between SASTFuzz and the baseline fuzzers, with campaigns that miss the respective bugs receiving a 48-hour TTE (*i.e.*, the maximum campaign duration). Hence, positive values indicate faster bug detection by SASTFuzz, while negative values signify faster discovery by the respective baseline fuzzer. Compared with AFL, SASTFuzz finds bugs faster in Discount, MyHTML, and Pocketlang by averages of 10.5, 1.8, and 3 hours, respectively. However, the opposite holds on Parson and raylib, where AFL requires 3.8 and 4.4 hours less for bug detection. Against the defect-guided fuzzers, SASTFuzz's

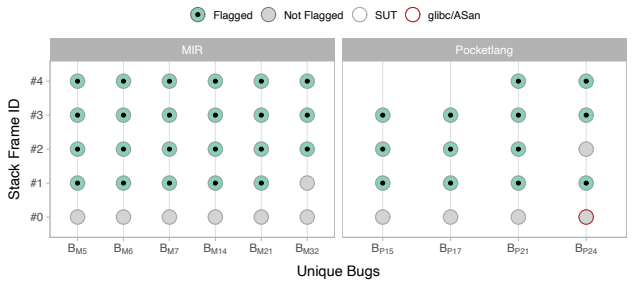


Figure 7: Flagged stack frames of the bugs exclusively found by SASTFuzz. Frame #0 indicates the last execution point where the bug manifests, which can be outside the system under test (red frames).

time savings become more apparent. Here, SASTFuzz is in five of the six subjects 2.2 to 5 hours faster than TortoiseFuzz but lags by about 2 hours on Parson. Moreover, SASTFuzz is significantly faster than ParmeSan, with 13.2 hours faster bug detection on Pocketlang. For the other programs, the time differences are even higher (over 20 hours; not shown in the zoomed boxplot) due to the many overlooked bugs by ParmeSan.

> Answer (RQ.3). SASTFuzz surpasses AFL in bug-detection speed in half of the programs by 1.8 to 10.5 hours (during 48-hour fuzzing campaigns) but takes in two subjects about 4 hours longer on average. Compared to the defect-guided fuzzers, SASTFuzz consistently shows higher efficiency, detecting bugs 2.2 to 5 hours faster than TortoiseFuzz and more than 13 hours faster in most subjects than ParmeSan.

5.2.4 RQ.4: Post-Bug Analysis. This research question investigates (1) whether the bugs exclusively identified by SASTFuzz are due to the SAST-provided targets, and (2) why SASTFuzz fails to detect certain bugs found by the baseline fuzzers.

Exclusive SASTFuzz Bugs. Figure 7 shows the stack frames flagged by the SAST tools for the bugs solely discovered by SASTFuzz. In MIR, almost all bugs had four of the top five functions/frames flagged, *effectively* guiding our fuzzer. These bugs, primarily segmentation violations (segv), exhibit different execution traces but manifest at adjacent code locations in frame #0 (missed by the analyzers). The same high proportion of SAST-flagged frames can be seen for the segmentation bug B_{P21} in Pocketlang. Moreover, bugs B_{P15} and B_{P17} show a pattern where three out of four frames (maximum trace length) were flagged. For the stack-overflow B_{P24}, where frame #0 is outside the analyzers' scope, three of the four frames on the corresponding trace were again flagged.

Exclusive Baseline Fuzzer Bugs. Among the ten bugs not detected by our fuzzer, the SAST tools failed to flag any of the top five frames only of B_{P23}, causing SASTFuzz to ignore this false-negative target during execution. For the other bugs, the static analyzers flagged many frames in their stack traces, though fewer than those of the bugs detected by SASTFuzz. However, replaying the stored fuzz inputs from SASTFuzz's queue of the respective

Table 6: Bug-finding performance of SASTFuzz, including time-to-first-error differences (ΔTTFE) compared to the fastest baseline fuzzer, when using the stack frame locations of previously missed bugs as targets.

Subject	Bug ID	Vuln. Type	Detected	ΔTTFE
MIR	B_{M4}	Segv	✓	13 hr 32 min
	B_{M16}*	Segv	✓	31 hr 09 min
	B_{M25}	Segv	✓	18 hr 04 min
	B_{M31}	Segv	✓	+ 4 hr 33 min
	B_{M36}	Segv	✓	16 hr 41 min
	B_{M40}	Segv	✓	0 hr 53 min
	B_{M41}*	Segv	✓	30 hr 52 min
Pocketlang	B_{P12}	Segv	✓	+ 2 hr 23 min
	B_{P16}	Segv	✗	–
	B_{P23}	Stack-overflow	✗	–

* Unlike the baseline fuzzer crashes, SASTFuzz's were intercepted at stack frame #0 by ASan instead of the operating system. However, as all other frames match, including the vulnerability type, we consider them detected by SASTFuzz.

campaigns reveals that the functions where these bugs manifest could not be reached. Here, the many specified (false-positive) SAST findings/fuzzing targets specified forced SASTFuzz to limit the executions per target to still maximize target coverage, thereby shifting direction before the bugs could be reached or triggered.

This is also evidenced by the results in Table 6, showing that when filtering out the false positive targets by steering SASTFuzz directly towards the functions in the crashing execution traces, 8 of the 10 previously missed bugs could be detected. Furthermore, except for B_{M31} and B_{P12}, SASTFuzz finds these bugs on average ~13 hours earlier than the fastest baseline fuzzer. This not only indicates the impact of false positives on our approach but also SASTFuzz's *potential performance increase* as the accuracy of static analyzers improves over time.

> Answer (RQ.4). For the bugs exclusively found by SASTFuzz, most of the top five functions on the execution trace towards the corresponding bug locations were flagged by the analyzers, pointing our fuzzer in the right direction. In contrast, the missed bugs result from too many (false-positive) targets specified, forcing SASTFuzz into less exhaustive fuzzing. Yet, they are likely detectable with more accurate analyzers.

5.3 Discussion

SASTFuzz detects previously undiscovered (security) bugs more effectively than the widespread baseline fuzzers. Our post-bug analysis shows that this performance gain largely stems from the SAST tools that direct our fuzzer to potentially vulnerable code regions. Besides, as static analyzers improve over time, SASTFuzz *automatically* becomes more effective. With the currently employed analyzers, SASTFuzz finds twice as many distinct bugs as the best-performing baseline fuzzer, making it a valuable addition to existing fuzzer ensembles. We consider this a major achievement, given that oftentimes, *a single new vulnerability* can have wide-ranging consequences, as illustrated by a recent high-severity vulnerability

in libwebp (CVE-2023-4863), affecting millions of devices. However, while SASTFuzz detects bugs faster than other defect-guided fuzzers, its time-savings over AFL are marginal. This can be attributed to AFL's higher throughput of fuzz inputs, which is slowed down by SASTFuzz's complex distance computations. Further engineering to optimize these computations could thereby enhance SASTFuzz's efficiency.

6 THREATS TO VALIDITY

External Validity. To minimize the risk of non-generalizable results, we conduct a *large-scale empirical evaluation* that compares SASTFuzz against one coverage-based (AFL) and two defect-guided fuzzers (ParmeSan and TortoiseFuzz). Moreover, we employ 19 diverse open-source programs with previously unknown (security) bugs, which also prevents survivorship bias and potential overfitting of the fuzzers and analyzers.

Internal Validity. This threat category concerns the extent to which our study minimizes potential methodological mistakes.

The many program crashes triggered in our fuzzing experiments forced us to mechanize the process of deduplicating crashes into unique bugs. Here, we utilized a *well-established heuristic* from the fuzzing domain in which crashes that share the same vulnerability type and the top N stack frames are considered to have the same underlying vulnerability. By *manually examining* the deduplicating results for different frame lengths, we found that $N = 5$ yields the most precise approximation.

Furthermore, due to the substantial time and hardware costs involved in fuzzing, we compromised in our evaluation on the number of experiment repetitions (10 instead of 20 or 30) in favor of longer fuzzing campaigns (48 instead of 24 hours). We consider this a reasonable compromise, as new and, thus, more complex bugs can often only be found with fuzzing efforts longer than the recommended 24 hours.

7 RELATED WORK

Improving Fuzzing Directedness. Böhme *et al.* [13] pioneered directed grey-box fuzzing with their directed AFL-variant AFLGo, building the basis [63] for sophisticated seed prioritization strategies [16, 19, 23, 27, 29, 33, 35, 36, 41, 48, 58, 62, 64, 66]. Also, new mutation strategies emerged that, for instance, dynamically adjust the mutation granularity [10, 16, 59, 70] or apply data-flow/taint-analysis techniques [19, 26, 62, 69, 70] to further improve fuzzing directedness. To speed up directed fuzzing, recent works suggest filtering out inputs unlikely to reach any target locations by predicting their target reachability via deep learning [76], pruning unreachable program paths by discarding executions deviating from target-reachable paths [25, 61], and/or selective path exploration using static control-/data-flow analysis methods [41].

To reach targets more effectively, we *dynamically disable* them during fuzzing, an approach used only by Zheng *et al.* [23] so far. Their fuzzer focuses on the 20% of targets with the lowest execution frequency while excluding unreached targets entirely from the exploitation phase. In contrast, our target scheduling (1) *fully deactivates* apparent false-positive targets, allowing SASTFuzz to focus on more likely vulnerable targets and (2) *systematically includes* unreached targets during exploitation to increase their coverage chances. However, most of the optimizations mentioned above are *complementary* to ours.

Automated Fuzzing Target Identification. Directed fuzzing often integrates with defect prediction techniques or tools for automated target acquisition. Early studies in this direction focused on methods targeting one specific vulnerability type, *e.g.*, use-after-free [48, 62], buffer and integer overflow [22, 26, 44], and denial-of-service [30, 32, 53, 67]. By utilizing multiple advanced SAST tools that support various vulnerability types [37], we can identify *different* types of vulnerabilities with a single fuzzer, thereby significantly reducing the associated execution costs.

Recent defect-guided fuzzing works address a wider range of vulnerabilities at both binary and source code levels. Binary-level methods primarily utilize static analysis [27, 48, 51] or machine/deep learning techniques [34, 72] on the reverse-engineered system under test (SUT) to identify interesting fuzzing target, contrasting with our approach that operates on the *source code level*. Approaches that require access to the SUT's source code either leverage code churn metrics [71, 73], software patches [43, 70], and/or bug tracker data [65, 70] to guide fuzzing toward bug-prone code locations. In contrast, our approach involves running multiple SAST tools on the SUT before fuzzing to *instantly* provide the targets without depending on pre-existing data.

For instant target identification, related approaches employ deep learning [74] or analyze the control/data flow [35, 68], the instruction semantics [59, 65], and/or the code complexity [18] of the SUT. Moreover, existing studies guided fuzzing towards sanitizer-instrumented lines [23, 49, 58] or sanity checks [17]. Unlike these methods, SASTFuzz uses SAST tools for automated target acquisition, a *new direction* that has not been explored until now. In our evaluation, SASTFuzz shows superior bug-finding performance than TortoiseFuzz [65], which focuses on memory-modifying instructions, and the sanitizer-guided fuzzer ParmeSan, demonstrating the effectiveness of our approach.

8 CONCLUSION

Coverage-based fuzzing, commonly used for detecting security bugs, *inefficiently* allocates resources by treating all code regions as equally important/vulnerable. Therefore, we suggest using static application security testing (SAST) tools to direct fuzzing towards *potentially vulnerable regions*, thus improving bug-finding effectiveness and efficiency by reducing the code scope to be fuzzed. Our approach also includes a *target scheduling mechanism* (implemented in our SAST-guided fuzzer SASTFuzz) that mitigates the impact of false-positive and unreachable SAST findings/fuzzing targets on the fuzzing performance. Furthermore, SASTFuzz *dynamically* alternates between directed and coverage-based fuzzing, helping to find bugs missed by the static analyzers (false negatives). Through large-scale empirical experiments, we show that (1) our target scheduling strategy *surpasses* prevailing directed fuzzing in terms of target locations reached and that (2) SASTFuzz not only finds *more* and *different* previously unknown (security) bugs but also detects bugs generally several hours *earlier* than the selected baseline fuzzers (AFL, TortoiseFuzz, and ParmeSan). Hence, SASTFuzz can decisively contribute to counteracting the escalating costs of current grey-box fuzzing.

REFERENCES

[1] 2023. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/. Accessed: 2023-12-09.

[2] 2023. Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed: 2023-12-09.

[3] 2023. ClusterFuzz. https://google.github.io/clusterfuzz/. Accessed: 2023-12-09.

[4] 2023. CodeQL for Research. https://codeql.github.com/. Accessed: 2023-12-09.

[5] 2023. Flawfinder. https://dwheeler.com/flawfinder/. Accessed: 2023-12-09.

[6] 2023. Infer: A Tool to Detect Bugs in Java and C/C++/Objective-c Code. https://fbinfer.com/. Accessed: 2023-12-09.

[7] 2023. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2023-12-09.

[8] 2023. Semgrep: Find Bugs and Enforce Code Standards. https://semgrep.dev/. Accessed: 2023-12-09.

[9] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. *Proceedings of the International Conference on Software Engineering*, 1–10. https://doi.org/10.1145/1985793.1985795

[10] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. *Proceedings of the IEEE Symposium on Security and Privacy* 2020-May, 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[11] Nathaniel Ayewah, David Hovemeyer, David J. Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25 (2008), 22–29. Issue 5. https://doi.org/10.1109/MS.2008.130

[12] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. *Proceedings of the Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 713–724. https://doi.org/10.1145/3368089.3409729

[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. *Proceedings of the Conference on Computer and Communications Security*, 2329–2344. https://doi.org/10.1145/3133956.3134020

[14] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-based Fuzzer Benchmarking. *Proceedings of the International Conference on Software Engineering*, 1621–1633. https://doi.org/10.1145/3510003.3510230

[15] Hudson Borges and Marco Tulio Valente. 2018. What's in a Github Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[16] Hongxu Chen, Bihuan Chen, Yinxing Xue, Xiaofei Xie, Yang Liu, Yuekang Li, and Xiuheng Wu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. *Proceedings of the ACM Conference on Computer and Communications Security*, 2095–2108. https://doi.org/10.1145/3243734.3243849

[17] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. *Proceedings of the IEEE Symposium on Security and Privacy* 2020-May, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[18] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics. *Proceedings of the International Conference on Software Engineering* 2019-May, 60–71. https://doi.org/10.1109/ICSE.2019.00024

[19] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. *Proceedings of the International Conference on Software Engineering* 2022-May. https://doi.org/10.1145/3510003.3510197

[20] Christoph Gentsch. 2020. Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C. , 37 pages. https://elib.dlr.de/133945/

[21] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the Capability of Static Code Analysis to Detect Security Vulnerabilities. *Information and Software Technology* 68 (12 2015), 18–33. https://doi.org/10.1016/j.infsof.2015.08.002

[22] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowser: A Guided Fuzzer for Finding Buffer Overflow Vulnerabilities. *The magazine of USENIX and SAGE* 38, 16–19. Issue 6.

[23] Han, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Zheng Payer. 2023. FishFuzz: Catch Deeper Bugs by Throwing Larger Nets. *Proceedings of the USENIX Security Symposium.*

[24] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2021. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* 4 (2021), 81–82. Issue 3. https://doi.org/10.1145/3410220.3456276

[25] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing With Provable Path Pruning. *Proceedings of the IEEE Symposium on Security and Privacy* 2022-May. https://doi.org/10.1109/SP46214.2022.9833751

[26] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. *ACM International Conference Proceeding Series* 2018-January. https://doi.org/10.1145/3274694.3274746

[27] Tiantian Ji, Zhongru Wang, Zhihong Tian, Binxing Fang, Qiang Ruan, Haichen Wang, and Wei Shi. 2020. AFLPro: Direction Sensitive Fuzzing. *Journal of Information Security and Applications* 54 (2020). https://doi.org/10.1016/j.jisa.2020.102497

[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. *Proceedings of the Conference on Computer and Communications Security*, 2123–2138. https://doi.org/10.1145/3243734.3243804

[29] Gwangmu Lee and Byoungyoung Lee. 2021. Constraint-Guided Directed Greybox Fuzzing. *Proceedings of the USENIX Security Symposium.*

[30] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. *Proceedings of the International Symposium on Software Testing and Analysis.* https://doi.org/10.1145/3213846.3213874

[31] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A Survey. *Cybersecurity* 1 (12 2018), 6. Issue 1. https://doi.org/10.1186/s42400-018-0002-y

[32] Penghui Li, Yinxi Liu, and Wei Meng. 2021. Understanding and Detecting Performance Bugs in Markdown Compilers. *Proceedings of the International Conference on Automated Software Engineering.* https://doi.org/10.1109/ASE51524.2021.9678611

[33] Rundong Li, HongLiang Liang, Liming Liu, Xutong Ma, Rong Qu, Jun Yan, and Jian Zhang. 2020. GTFuzz: Guard Token Directed Grey-Box Fuzzing. *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, 160–170. https://doi.org/10.1109/PRDC50213.2020.00027

[34] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. 2020. V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs. *IEEE Transactions on Cybernetics* (2020), 1–12. https://doi.org/10.1109/TCYB.2020.3013675

[35] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence Directed Hybrid Fuzzing. *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering.* https://doi.org/10.1109/SANER48275.2020.9054807

[36] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. 2019. Sequence Coverage Directed Greybox Fuzzing. *Proceedings of the International Conference on Program Comprehension* 2019-May. https://doi.org/10.1109/ICPC.2019.00044

[37] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. *Proceedings of the International Symposium on Software Testing and Analysis*, 544–555. https://doi.org/10.1145/3533767.3534380

[38] Stephan Lipp, Daniel Elsner, Severin Kacianka, Pretschner Alexander, Marcel Böhme, and Sebastian Banescu. 2023. Artifacts for the Paper: "Green Fuzzing: A Saturation-Based Stopping Criterion Using Vulnerability Prediction". https://doi.org/10.5281/zenodo.7944722

[39] Stephan Lipp, Daniel Elsner, Severin Kacianka, Alexander Pretschner, Marcel Böhme, and Sebastian Banescu. 2023. Green Fuzzing: A Saturation-Based Stopping Criterion Using Vulnerability Prediction. *Proceedings of the International Symposium on Software Testing and Analysis.* https://doi.org/10.1145/3597926.3598043

[40] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, Chao Zhang, and Chao Zhang. 2020. A Large-scale Empirical Study on Vulnerability Distribution Within Projects and the Lessons Learned. *Proceedings of the International Conference on Software Engineering*, 1547–1559. https://doi.org/10.1145/3377811.3380923

[41] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing With Selective Path Exploration. *Proceedings of the IEEE Symposium on Security and Privacy* 2023-May. https://doi.org/10.1109/SP46215.2023.10179296

[42] 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), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[43] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* https://doi.org/10.1145/2491411.2491438

[44] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. 2020. Fitness Guided Vulnerability Detection with Greybox Fuzzing. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 513–520. https://doi.org/10.1145/3387940.3391457

[45] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1393–1403. https://doi.org/10.1145/3468264.3473932

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

[47] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, 529. https://doi.org/10.1145/1315245.1315311

[48] Manh Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-Level Directed Fuzzing for Use-After-Free Vulnerabilities. *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*.

[49] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. *Proceedings of the USENIX Security Symposium*, 2289–2306.

[50] Mathias Payer. 2019. The Fuzzing Hype-train: How Random Testing Triggers Thousands of Crashes. *IEEE Security and Privacy Magazine* 17 (1 2019), 78–82. Issue 1. https://doi.org/10.1109/MSEC.2018.2889892

[51] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dVul: Discovering 1-Day Vulnerabilities Through Binary Patches. *Proceedings of the International Conference on Dependable Systems and Networks*. https://doi.org/10.1109/DSN.2019.00066

[52] Jose D.Abruzzo Pereira, Joao R. Campos, and Marco Vieira. 2021. Machine Learning to Combine Static Analysis Alerts with Software Metrics to Detect Security Vulnerabilities: An Empirical Study. *Proceedings of the European Dependable Computing Conference*. https://doi.org/10.1109/EDCC53658.2021.00008

[53] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. *Proceedings of the Conference on Computer and Communications Security*. https://doi.org/10.1145/3133956.3134073

[54] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. 2016. On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE. *Proceedings of the International Conference on Software Engineering* 14-22-May-2016. https://doi.org/10.1145/2884781.2884835

[55] Patrick Schober and Lothar A. Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia and Analgesia* 126 (2018). Issue 5. https://doi.org/10.1213/ANE.0000000000002864

[56] Kostya Serebryany. 2017. OSS-Fuzz: Google's Continuous Fuzzing Service for Open-Source Software. USENIX Association, Vancouver, BC.

[57] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2019. AddressSanitizer: A Fast Address Sanity Checker. *Proceedings of the USENIX Annual Technical Conference*, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[58] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing. *Proceedings of the Conference on Computer and Communications Security*. https://doi.org/10.1145/3548606.3560648

[59] Lingyun Situ, Linzhang Wang, Xuandong Li, Le Guan, Wenhui Zhang, and Peng Liu. 2019. Energy Distribution Matters in Greybox Fuzzing. *Proceedings of the International Conference on Software Engineering (ICSE-Companion)*. https://doi.org/10.1109/ICSE-Companion.2019.00109

[60] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. *Proceedings of the IEEE Symposium on Security and Privacy* 2019-May, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[61] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. 2022. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-Tailored Program State Restriction. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3564625.3564643

[62] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-After-Free Vulnerabilities. *Proceedings of the International Conference on Software Engineering*, 999–1010. https://doi.org/10.1145/3377811.3380386

[63] Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2022. SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *SSRN Electronic Journal* (2022). https://doi.org/10.2139/ssrn.4129684

[64] Shen Wang, Xunzhi Jiang, Xiangzhan Yu, and Shuai Sun. 2021. KCFuzz: Directed Fuzzing Based on Keypoint Coverage. *Proceedings of the Lecture Notes in Computer Science* 12736 LNCS. https://doi.org/10.1007/978-3-030-78609-0_27

[65] 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. *Proceedings of the Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2020.24422

[66] Zi Wang, Ben Liblit, and Thomas Reps. 2020. TOFU: Target-Oriented Fuzzer. (4 2020).

[67] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory Usage Guided Fuzzing. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 765–777. https://doi.org/10.1145/3377811.3380396

[68] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 789–800. https://doi.org/10.1145/3377811.3380388

[69] Jiaxi Ye, Ruilin Li, and Bin Zhang. 2020. RDFuzz: Accelerating Directed Fuzzing With Intertwined Schedule and Optimized Mutation. *Mathematical Problems in Engineering* 2020 (2020). https://doi.org/10.1155/2020/7698916

[70] Wei You, Peiyuan Zong, Kai Chen, Xiao Feng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. *Proceedings of the Conference on Computer and Communications Security*. https://doi.org/10.1145/3133956.3134085

[71] Jia Ming Zhang, Zhan Qi Cui, Xiang Chen, Huan Huan Wu, Li Wei Zheng, and Jian Bin Liu. 2022. DeltaFuzz: Historical Version Information Guided Fuzz Testing. *Journal of Computer Science and Technology* 37 (2022). Issue 1. https://doi.org/10.1007/s11390-021-1663-7

[72] Yuyue Zhao, Yangyang Li, Tengfei Yang, and Haiyong Xie. 2020. Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning. *Proceedings of the Lecture Notes in Computer Science* 12020 LNCS. https://doi.org/10.1007/978-3-030-42921-8_8

[73] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. *Proceedings of the Conference on Computer and Communications Security*, 2169–2182. https://doi.org/10.1145/3460120.3484596

[74] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Camtepe Seyit, and Yang Xiang. 2020. DeFuzz: Deep Learning Guided Directed Fuzzing. (10 2020).

[75] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. *Proceedings of the International Conference on Software Testing, Verification and Validation*, 421–428. https://doi.org/10.1109/ICST.2010.32

[76] 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. *Proceedings of the USENIX Security Symposium*.