Igor: Crash Deduplication Through Root-Cause Clustering

Zhiyuan Jiang
NUDT

Xiyue Jiang
NUDT

Ahmad Hazimeh
EPFL

Chaojing Tang
NUDT

Chao Zhang
Tsinghua University

Mathias Payer
EPFL

ABSTRACT

Fuzzing has emerged as the most effective bug-finding technique. The output of a fuzzer is a set of proof-of-concept (PoC) test cases for all observed "unique" crashes. It costs developers substantial efforts to analyze each crashing test case. This, mostly manual, process has lead to the number of reported crashes out-pacing the number of bug fixes. Automatic crash deduplication techniques, which mostly rely on coverage profiles and stack hashes, are supposed to alleviate these pressures. However, these techniques both inflate actual bug counts and falsely conflate unrelated bugs. This hinders, rather than helps, developers, and calls for more accurate techniques.

The highly-stochastic nature of fuzzing means that PoCs commonly exercise many program behaviors that are orthogonal to the crash's underlying root cause. This diversity in program behaviors manifests as a diversity in crashes, contributing to bug-count inflation and conflation. Based on this insight, we develop Igor, an automated dual-phase crash deduplication technique. By minimizing each PoC's execution trace, we obtain pruned test cases that exercise the critical behavior necessary for triggering a bug. Then, we use a graph similarity comparison to cluster crashes based on the control-flow graph of the minimized execution traces, with each cluster mapping back to a single, unique root cause.

We evaluate Igor against 39 bugs resulting from 254,000 PoCs, distributed over 10 programs. Our results show that Igor accurately groups these crashes into 48 uniquely identifiable clusters, while other state-of-the-art methods yield bug counts at least one order of magnitude larger.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Crash Grouping, Fuzzing

ACM Reference Format:
Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash Deduplication Through Root-Cause Clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3460120.3485364

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3485364

1 INTRODUCTION

The main focus of software-testing research is finding bugs. Maximizing bug discovery is a key subject of interest across the development stack, from the physical layer [14, 59, 61, 77] to the application layer [5, 15, 51, 62, 71, 83]. The assumption "we can always afford to fix bugs" powers the drive for bug-finding techniques that yield large numbers of crashes in short time frames, the most prominent of which is fuzz testing. Big players in the software industry also motivate this movement, providing open-access reporting platforms [4, 29, 56] and bug bounties for their large user-bases. With little incentive to triage crashes, users and software testers frequently submit raw findings/crash-dumps, leaving it to software maintainers to bear the weight of distilling crashes and fixing bugs. With more crashes reported, more time is spent on crash triage and pruning duplicate reports, leaving maintainers with less time for fixing bugs and improving software quality. Large fuzzing farms (e.g., ClusterFuzz [30], OSS-Fuzz [68])—which run around the clock and automatically submit crash reports—exacerbate this problem. For example, as of late April 2021, there were 979 open Linux kernel bugs, with the earliest submitted in November 2017 (based on syzbot statistics [31]).

Solutions to this problem range from collaborative, where maintainers rely on the community to provide actionable analysis in their reports, to systematic, where heuristics are used over large crash dumps to filter out redundancies and duplicates [1, 19, 21, 41, 80]. These heuristics typically rely on dynamic program behaviors to identify root causes and answer the question "given a crashing test case, what is the most likely cause for the crash?" For example, Aurora [7] presents a root cause identification method based on *delta debugging* [85]: both faulty and benign test cases are executed and a disjunction in program behaviors marks the bug. In contrast, *crash bucketing* (*cf.* grouping) shifts the focus from pinpointing the cause of a crash to grouping crashes based on their root cause. Crash bucketing answer the question "given a number of crashing test cases, which crashes trigger the same bug?" Crash bucketing techniques are widely used in practice [43] and rely on *crash sites*, *coverage profiles*, or *stack hashes*, to cluster crashes.

Crash Sites. All bugs crash at a particular program location. These crash sites (e.g., the address stored in the instruction pointer at the time of crash) serve as a coarse-grain bug identifier. Unfortunately, crash sites are imprecise and lead to bug misclassification (e.g., for use-after-free bugs, objects may be arbitrarily reused, triggering a broad set of "unique" crashes). Crash sites both under- and over-estimate bug counts and are not used in practice.

Coverage Profiles. Coverage-guided fuzzers commonly use their coverage data to uniquely identify crashes. For example, AFL [83]

considers crashes that exercise new control-flow edges or omit common edges as "unique". The fine-grained nature of edge coverage makes it sensitive to small changes in control-flow and causes crash count inflation; slight modifications in the path to a bug result in a new crash. Coverage profiles overestimate bug counts by 2–3 orders of magnitude [33, 43].

Stack Hashes. Stack hashes provide function-sensitive labels for crashes and are commonly used by fuzzers (e.g., honggfuzz [71]) and fuzzing farms (e.g., ClusterFuzz [30]) alike. Stack hashing accumulates (the last N) function calls (on the stack) leading to the crash site, hashing these traces to form unique bug identifiers. Compared to coverage profiling, stack hashing is more coarse-grained and results in fewer crash buckets. However, stack hashes are prone to misclassification, both over- and under-approximating the number of bugs (the former is due to different paths to the crash site [7, 22], while the latter is due to bugs sharing the same call sequence). Stack hashes overestimate bug counts by 1–2 orders of magnitude [43].

Inaccurate crash grouping techniques (such as those previously described) waste precious developer time. In particular, bug identifiers (e.g., coverage profiles, stack hashes) are susceptible to fluctuations in program behavior. Coverage-guided fuzzers—which typically aim to maximize code coverage—amplify such fluctuations, yielding many crashing test cases that exercise "noisy" code extraneous to the underlying root cause. This noise impedes control-flow-based deduplication techniques, inflating bug counts. Moreover, accurate bug classification relies on identifying a bug's *characteristic trigger*. Identifying this trigger in a partial execution trace is made difficult if (a) the trigger is missing from the execution trace (e.g., if the trace is too coarse-grained), and/or (b) the execution trace contains extraneous elements (e.g., if the trace is too fine-grained). This calls for a new approach to crash grouping; one that removes noise from the execution trace—to accurately group paths—and preserves critical control-flow information—to accurately isolate different root causes.

Whereas a coverage-maximizing strategy is ideal for finding bugs through fuzzing, we propose that the counterpart—*coverage-minimizing fuzzing*—is key to minimizing an execution trace and enabling effective crash grouping. We make crash labels more precise by trimming unnecessary execution trace elements (i.e., noise), leading to more concise deduplication. We also address the shortcomings of stack hashes (which operate at function-call granularity) by using control-flow graphs (CFG) for a more complete view of a crash's execution trace. Using CFGs preserves critical control-flow information and allows for aggressive pruning of redundant (executed) code, leading to more accurate crash grouping.

We present Igor[1], a dual-phase crash deduplication technique that leverages a coverage-reduction fuzzer and a CFG similarity metric to cluster crashes by their critical behaviors. By simplifying each crash's execution trace, we obtain test cases that exercise the minimized behavior necessary for triggering a bug. Then, we perform a graph similarity comparison over the CFGs of all minimized execution traces to group them into closely-packed clusters, each mapping back to a unique root cause.

We answer the following research questions:

RQ1 What constitutes ideal crash grouping and why is it not achievable in practice?

RQ2 How strong is the effect of dense execution traces on crash-count inflation, and can sparsity promote precision?

RQ3 What metric is best suited for capturing and isolating root causes?

We make the following contributions: (i) a *coverage-reduction* fuzzer which applies a *minimizing fitness function* over the collected edge coverage to shrink test cases; (ii) a *new metric for crash grouping*, based on control flow graph similarity using the Weisfeiler-Lehman Subtree Kernel algorithm [45]; and (iii) a *ground-truth benchmark* for evaluating crash grouping techniques, containing 52 CVEs[2] and 254,000 crashing test cases from 14 real world programs (generated over 58.7 CPU-years of fuzzing). With the source code of these programs and patches of these bugs, we develop analysis tools and manually label these test cases with ground-truth (in approximately 30 human-days). This benchmark enables us to test the accuracy of Igor.

To aid both developers (so they can quickly and accurately uncover a crash's root cause) and researchers (so they can reproduce and expand on our results), we make our Igor prototype and benchmark available at https://github.com/HexHive/Igor.

2 BACKGROUND AND INTUITION

Crash bucketing groups test cases to isolate a crash's root cause. Accurate crash bucketing requires: (i) *grouping together crashes with the same root cause* (minimizing type I errors); (ii) *creating new groups for crashes with different root causes* (minimizing type II errors); and (iii) capturing accurate *bug context*. Capturing accurate bug context requires complete modeling and analysis of program behavior. However, existing modeling/analysis techniques (e.g., symbolic execution) do not scale [6]. Instead, practical crash bucketing relies on error-prone heuristics. Reducing these errors requires (a) *behavioral metrics* that correlate with bug context, and (b) *execution trace trimming* to minimize noise.

2.1 Behavioral Metrics

Bug context is the critical set of program behaviors accumulated and leading up to the crash site. Capturing bug context is key to accurate crash grouping. However, approximating program behavior is a three-way trade-off between sensitivity, accuracy, and scalability. On the upper end of the sensitivity spectrum, full path coverage (control and data flow) is the most precise: each test case exercises a unique path. This metric thus has a minimal type II error rate, achieving high accuracy in distinguishing different bugs. However, its precision also results in fine-grained grouping of crashes, and thus boasting a high type I error rate and a low accuracy in identifying similar root causes. Collecting precise path coverage is also infeasible in practice due to the large state space most programs have, and the scalability challenges that arise from this. On the other end of the spectrum, in singling out what program behavior led to a crash, Boolean function coverage is imprecise: it only captures the set of functions possibly involved in triggering a crash, and does so without conserving the order of code execution. Test cases

[1] In Terry Pratchett's Ankh-Morpok, the *Igors* are a group of humble professional servants (often to mad scientists) that are proficient transplant surgeons.

[2] In this paper, when two programs share a CVE, we count it as two CVEs

```
switch(format):
  case TAG_FMT_USHORT:
    info_value->u = get16u(vptr, motorola_intel);
    break;

  case TAG_FMT_ULONG:
    info_value->u = get32u(vptr, motorola_intel);
    break;

  case TAG_FMT_SSHORT:
    info_value->ur.num = get32u(vptr, motorola_intel);
    info_value->ur.den = get32u(4+(char *)vptr,
    ↪  motorola_intel);
    break;
  ...
```

Listing 1: Code snippet for CVE-2018-14883.

```
void overflow_one(char* r, size_t size) {
  char buffer[10];
  memcpy(buffer, r, size);        // crash here!
  return;
}

void overflow_two(char* r, size_t size) {
  int buffer_size = size;
  char buffer[buffer_size];
  memcpy(buffer, r, strlen(r));   // crash here!
  return;
}
```

Listing 2: Two bugs sharing the same crash site.

resulting from the same bug or from different bugs are likely to display identical coverage, thus reducing type I error rates but raising those of type II. Nevertheless, Boolean function coverage requires minimal resources to measure and collect, improving scalability.

2.1.1 Research Context. According to Dhaliwal et al. [23], 80 % of the root causes of a bug are located on the call stack at the time of the crash. Several research projects have taken this notion and explored different stack hashing techniques [11, 13, 21, 42, 46, 64, 66]. Stack hashing is fast and easy to deploy, and is suitable for large-scale campaigns to batch-process a large number of test cases, but is prone to both type I and II errors and lacks high-quality classification.

Execution traces can also be used to classify test cases. This approach typically uses binary translation [52] or interposition [57]. Several taint-based approaches have also looked into classifying test cases as a form of crash analysis. For example, CrashFilter [39] automatically classifies a failing test case based on static taint analysis. RETracer[20] recovers early program state from a memory dump based on a reverse taint analysis, after which an analyst manually groups test cases. REPT [19] extends RETracer and achieves a more accurate classification result by reconstructing the data flow. Finally, POMP [80] classifies crashes from the perspective of the different contributions of data flow to program crashes.

These taint-based methods group crashes from the perspective of data flow. They are more precise than stack hashing. However, taint-based methods often do not scale to complex programs as they need to record a detailed program trace, which quickly exceeds several 100 GiBs for even simple programs, and require complex symbolic reasoning over paths that use tainted input data to make control-flow decisions. Taint-based methods can be effective if the bug only depends on data flow and not on control flow decisions based on tainted input data [15, 62, 76].

2.1.2 Challenges. Listing 1 (based on CVE-2018-14883 in PHP) illustrates how a single bug can manifest as a crash in multiple locations, breaking location- and call-stack- based grouping tools. When triggered, the bug will cause `vptr` to point to an illegal address. A crash occurs when the pointer is accessed, which happens at lines 3, 7, and 11. Different values of `format` will cause a crash in different functions (`get16u` or `get32u`). In contrast, Listing 2 shows how different vulnerabilities can crash in the same location. There is an `if-else` structure in the program containing a unique stack overflow vulnerability on each branch. After triggering the vulnerability, the program eventually crashes when calling `memcpy`. Although the crash location is the same, the crash is caused by different vulnerabilities. Due to the complexity of control flow paths in programs, methods based on call stacks or crash sites cannot distinguish between different and similar bugs, resulting in a high misidentification rate [43].

In practice, finding the balance between sensitivity, accuracy, and scalability requires identifying which behavioral metric most closely correlates to the root cause. We conclude from our results that minimized execution traces give the best insight into bug triggers, making them amenable for similarity matching.

2.2 Test Case Reduction

Imperfections in behavioral metrics can be amplified by extraneous data points in the recorded execution trace. While reducing the sensitivity of the metric can improve its resilience against noisy measurements, this only addresses a symptom of the noise but not its source: entropy. In their study on fault localization, Christi et al. [17] showed that the accuracy of localization benefits greatly from reduced test cases. *Test case reduction* refers to continuously reducing the size of the crashing input, under the premise that the same crash is always triggered. In practice, streamlined test cases improve the software development process; e.g., in *vulnerability mining*, a minimized test case improves mutation efficiency and guides the process towards interesting behaviors; and in *crash analysis*, it eliminates extraneous bytes to simplify control flow leading up to the crash.

2.2.1 Research Context. The two main methods used in test case reduction are *delta debugging* [85] and *taint analysis* [50].

Delta debugging is a popular approach that automatically minimizes test cases by using two algorithms: *simplification* and *isolation* [54]. The former continuously shrinks the size of the original input file until it cannot find a smaller file that crashes the program, and the latter searches for a passing input, which will become a failing input again after satisfying additional constraints. The

most popular test case simplification tool for fuzzing campaigns, `afl-tmin` [83], is based on this principle.

Taint analysis is a method that marks accessed registers and memory as tainted when the program crashes, and then tracks the source of the taint to mark all crash-related components in the input, thereby reducing the crashing test case to the relevant bytes. In general any forward or backward taint analysis can be used [18], but in practice, the length of the input or trace is often prohibitive. Long traces and complex inputs result in over-tainting or under-tainting along with difficulties of keeping control of the control-flow dependencies given the synthetic input. In practice, the application of taint analysis to crash grouping is limited.

2.2.2 Challenges. Aurora [7] is a root-cause identification method that leverages delta debugging to distinguish between critical and benign execution traces resulting from a crash. The process begins with a crash diversification phase where the faulty input is mutated to generate both crashing and non-crashing test cases. However, diversification carries the risk of introducing new bugs that are unrelated to the root cause under study. This is because Aurora follows an exploratory fuzzing process (i.e., increasing coverage) to generate new inputs.

Test case minimization may also introduce new bugs unrelated to the original bug. This is combated with a stricter fitness function: only a subset of the original trace must be executed, heavily limiting exploration. While this does not provide a guarantee against the introduction of new bugs, our evaluation shows that the error rate is negligible in practice. In comparison to AFL's crash mode [83]—which shares the fuzzer's fitness function—Igor introduces 90 % fewer false positives.

The problem of minimizing crashes is also non-convex. Typically, the fitness function for test case reduction is the size of the input (in bytes). We instead propose minimizing the size of program's *execution trace*, thus reducing the execution complexity, rather than the input's size. Pruning the execution trace of a crashing test case can yield more concise inputs that exercise the same desired program behavior. A bug is triggered by a subset of all possible execution traces, and through a minimization function over the input space, a critical path to the root cause can be found and used to triage the crash. However, program behavior is complex, and a hill-climbing minimization process is likely to converge to a local minimum. Intuitively, traces that converge to different local minima may suggest different root causes, further complicating analysis. However, we found that combining this process with a clustering procedure led to minor variations in traces being overlooked in favor of the global trace overlap.

2.3 Our Approach: Igor

While existing methods provide an initial step at reducing the large number of crashing test cases requiring triage, the type I and type II errors that these methods introduce result in uncertainty and may even increase developer effort and/or cause missed bugs. We address this imprecision by introducing a dual-phase approach for crash analysis. This approach builds on a key observation: each bug has a *core behavior* that must be executed to trigger the bug. A technique that extracts and matches this behavior can distil the large amount of crashes into a precise set of unique bugs. While analyzing large numbers of crashing test cases (together with their ground-truth), we observed that test cases for the same bug partially overlap in essential phases of their execution trace; *the bug trigger*. Our technique, Igor, extracts an approximation of bug triggers and then leverages topological graph matching to group similar test cases into bug classes. This dual-phase approach for efficient and effective crash grouping through root-cause analysis thus combines and extends two important areas of research: *test case reduction*—minimizing and simplifying test cases—and *crash grouping*—determining if two inputs trigger the same or different bugs. Broadly speaking, Igor leverages test case reduction to simplify the execution traces observed by the test cases, so that we can group crashes based on similarities in their coverage (i.e., we improve the latter by leveraging the former).

3 IGOR DESIGN

Fig. 1 depicts the main components and workflow of Igor. Starting with a set of crashing test cases, the *preprocessing* stage reduces unnecessary analysis costs by leveraging sampling and `afl-tmin`. Following this, the *trace generator* (`IgorFuzz`) records and minimizes the execution traces of each preprocessed test case. The *graph analyzer* then constructs control-flow graphs from the minimized traces and extracts graph similarity metrics that describe each test case. Finally, the *cluster builder* classifies the test cases into separate groups, each identifying a unique root cause, and leverages a validation loop to find an optimal clustering configuration.

3.1 Data Preprocessing

Igor's key objective is to distil crashing, proof-of-concept test cases (PoC) into unique bugs. To be practical, Igor must scale with increasing numbers of PoCs. However, minimizing and grouping several hundred thousands PoCs is time-consuming. Therefore, we employ a two-stage preprocessing phase to reduce processing cost while maintaining accuracy. First, we sample the PoC corpus to reduce the number of analyzed test cases. Second, we leverage `afl-tmin` as an initial test-case size minimization tool, allowing `IgorFuzz` to converge to a solution faster.

Sampling. Although stack hashing is generally imprecise, it is rare that two *different* root causes overlap in the entire call stack. In other words, one bug may result in many diverse stack hashes but one stack hash generally only maps to one bug. In our dataset, only six (of 71) bug pairs in three programs contain shared stack hashes. We leverage this observation to reduce the PoC corpus by grouping PoCs based on their full-length call stack hashes. If there are many PoCs mapping to the same unique stack hash, we only process the 50 most diverse PoCs. This allows us to remove highly-similar PoCs that map to the same bug, lowering the cost of processing without impacting precision.

Minimization. Minimized test cases benefit fuzzing in two ways: (i) the reduced size of the input leads to faster parsing and processing by the target, yielding higher fuzzing throughput; and (ii) by removing extraneous bytes from the input, the fuzzer's mutations are more likely to modify bytes critical to the behavior of the program. The case of minimum-coverage fuzzing is no different and

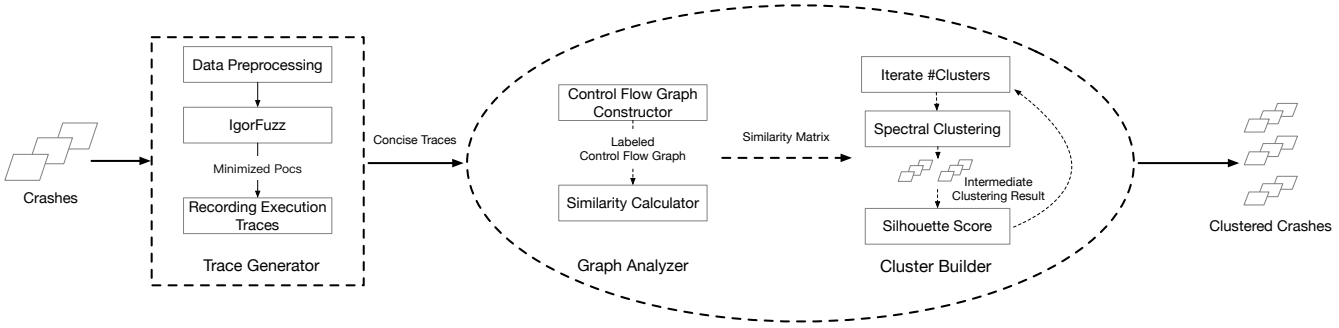


Figure 1: Igor overview.

can thus be improved by preprocessing test cases to remove extraneous bytes. We achieve this with the aid of `afl-tmin` [83], a lightweight test case reduction tool included with AFL. Although `afl-tmin` focuses exclusively on reducing the length of the input (i.e., this is the sole metric it optimized for), it indirectly shortens the length of the execution trace, further assisting `IgorFuzz` in finding a minimal PoC. `Afl-tmin` also allows us to merge PoCs that reduce to identical byte sequences. Using `afl-tmin` in the second preprocessing stage increases `IgorFuzz`'s effectiveness in exploring shorter execution traces. As discussed in Section 2.2, `afl-tmin` may introduce new bugs. However, uncovering a new bug during minimization with `afl-tmin` is rare: in our evaluation, only one out of the 5,531 PoCs triggered a different bug after minimization through `afl-tmin`. Our evaluation shows that the other nine PoCs (for the same bug) remain correct, allowing Igor to correctly cluster the nine PoCs to that root cause. While we lost one PoC (out of 5,531) during minimization, enough PoCs remained to correctly identify all unique bugs. This effectively means that we did not introduce any false negatives in our evaluation through `afl-tmin`.

3.2 `IgorFuzz`: Minimum-Coverage Fuzzing

Coverage-guided fuzzing typically incorporates feedback to *maximize* code coverage and to trigger crashes. The highly-stochastic nature of fuzzers means that they often find many diverse test cases that trigger the same bug. This results in extraneous execution traces that amplify the imprecision of the underlying metric and hampers clustering.

The intuition behind minimum-coverage fuzzing is that a bug only manifests when the faulty code is executed. Minimizing the code executed before reaching faulty code reduces the amount of state that the developer has to analyze during debugging. Given a mechanism to measure what code was executed when a bug is triggered, and a mechanism to sort the measurements across different samples, it is possible to find the minimal code trace required to trigger the bug, which we identify as the **shortest bug-triggering path** (with the absolute minimum code trace being the empty set). This observation accounts for the possibility that a bug can be triggered through different paths. Through fuzzing, we perform a search over the state space in the vicinity of the different bug paths, with the objective of finding the shortest (simplest) execution trace. Since fuzzing is a dynamic technique, it is inevitable that multiple suboptimal solutions will be found; however, our evaluation shows that the different solutions display features that are similar enough to be grouped under the same root cause.

Ideally, the single shortest path to a bug is its best identifier. An oracle that determines the shortest path would solve the crash grouping problem, as all PoCs for the same bug would reduce to the same path. Reducing an arbitrary PoC to the minimal path is challenging, because it requires determining the shortest path that still satisfies all of the bug context's constraints. In practice, this results in long execution traces that exceed the capabilities of today's solvers. For example, the shortest paths in our evaluation contained over 91,000 basic blocks (in *LibPNG*), while the shortest execution trace contained up to 22,563,000 basic blocks (in *Poppler*).

Existing test case reduction techniques use an objective function that favors a smaller input size. This is an artifact of bug reporting guidelines, which typically require a *minimum working example* as a PoC. However, reducing the input size does not always translate to reduced trace complexity. To highlight this effect, we conduct a study on `afl-tmin` using five targets and ten bugs. While test cases generally reduce in size, this does not necessarily guarantee a reduction in the length of the execution trace. For example, `afl-tmin` reduces *OpenSSL*'s `x509` input size by 41.10 %, while only pruning 4.83 % of CFG edges. Moreover, although `afl-tmin` reduces the input size of `pdfimages` by 82.45 % (on average), 15.18 % of the test cases execute more edges after reduction by `afl-tmin`. This demonstrates that a smaller input size does not guarantee a simpler execution trace. See Table 6 and Appendix C for our full results.

Similar to vulnerability mining, fuzzing provides an opportunity for an efficient and scalable process that finds simpler inputs which exercise similar behavior. To reduce the complexity of the execution trace, we propose a new type of fuzzer with subtly different goals than existing fuzzers. The fitness function in our fuzzer, `IgorFuzz`, favors a *simpler execution trace*, by exercising fewer edges while still crashing the target at the same location.

The goal of reduction is to find simpler test cases that trigger the same bug. Multiple bugs can share the same crash site, and minimized test cases present the risk of triggering different bugs at the same location. However, in practice, the probability of *simplified* test cases triggering different bugs is insignificant. It is important to distinguish the use of crash sites between *grouping* and *simplifying* crashes. When grouping based on crash sites, inputs are diverse and presumably exercise different program behaviors. Due to this noise, misclassified bugs will be common. However, when simplifying a

Algorithm 1 IgorFuzz algorithm

```
1:  procedure FuzzTest(Prog, Seeds)
2:      Queue ← Seeds
3:      while true do
4:          for input in Queue do
5:              if ¬isWorthFuzzing(input) then
6:                  continue
7:              mutated ← Mutate(input)
8:              if ¬isInteresting(mutated) then
9:                  continue
10:             mutated.score ← calcScore(Prog, mutated)
11:             addToQueue(mutated)
12: procedure isFavorite(input)
13:     return trimsDisjunctEdges(input)
14: procedure isWorthFuzzing(Prog, input)
15:     return isFavorite(input) or isLarger(input.score)
16: procedure isInteresting(Prog, input)
17:     return isCrashing(input) and
            (hasSmallerBitmapSize(input) or
            isEdgePruned(input) or
            hasSmallerHitCountSum(input))
18: procedure calcScore(Prog, input)
19:     score ← input.score
20:     if executesFaster(Prog, input) then
21:         score ← calcScoreFast(input)
22:     if hitsFewerEdges() then
23:         score ← calcScoreSimple(input)
24:     return score
```

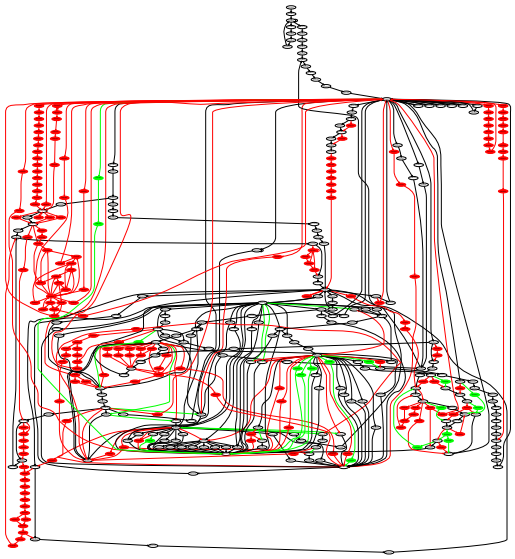


Figure 2: Function-call graph of `tiffcp`. This figure shows the nodes and edges removed (highlighted in red) or added (highlighted in green) by IgorFuzz.

test case based on its crash site, we reduce the complexity of the execution trace while exercising similar program behavior. This minimizes the likelihood of the new derived test case triggering a different bug.

State-of-the-art fuzzers aim for maximal coverage to aid bug discovery. Existing work in *maximizing coverage* can be grouped into three areas: seed retention strategies [28, 62, 75, 78], seed selection strategies [9, 15, 24], and seed scheduling strategies [8, 9, 81]. We guide the design of IgorFuzz based on insights from these three areas and modify a generic feedback-guided greybox fuzzer such as the one presented by Böhme et al. [9]. Algorithm 1 outlines our coverage-minimizing algorithm and Fig. 2 gives an example of IgorFuzz's effectiveness.

3.2.1 Seed Retention Principles. We modify the `isInteresting` method so that it only retains crashing test cases. We also introduce two complementary rules for seed retention:

Rule 1 The seed does not exercise a common edge (i.e., at least one edge is no longer executed, although overall coverage may increase).

Rule 2 The seed exercises some edges with fewer hit counts (i.e., at least one edge is executed fewer times and no edge is executed more times).

Through a hill-climbing optimization process over the coverage space, IgorFuzz incrementally approaches a minimal execution trace. To reduce the risk of converging at local minima and to promote diversification, IgorFuzz retains seeds that meet any of the two rules. Note that it does not suffice to measure if the new input executes strictly fewer edges than the original input, as IgorFuzz may get stuck at a local minimum from which it cannot easily escape. Rule 1 therefore prioritizes a seed that prunes common edges, but permits it to execute more edges if at least one previous edge is no longer executed. Rule 2 minimizes hit counts to simplify loops and recursive function calls. Together these rules drive coverage downwards, one removing edges (but potentially adding new ones), the other reducing edge counts. Note that at any point in time, multiple seeds are being scheduled for mutation depending on the seed selection principles below.

3.2.2 Seed Selection Principles. To achieve a higher efficiency at exploring the coverage space of a target, AFL [83] marks the seeds which exercise disjunct sets of edges as favorites. By finding a minimal set over the sets of edges executed by each seed (e.g., as performed by Karp [40]), AFL determines the seeds which are more likely to increase coverage in different directions of the code and more favourably mutates those seeds. Previous research on seed selection [35] also validates this strategy: the fewer and more distinct the seeds are, the better the fuzzer is able to maximize coverage.

In contrast to AFL's approach, IgorFuzz attempts to minimize coverage by favoring seeds which *trim* disjunct sets of edges. New seeds that trim groups of edges are more likely to prune those edges from the execution trace upon further mutation. For that reason, we mark such seeds as favorites and assign them higher energy.

3.2.3 Seed Energy Scheduling. Seed energy determines how many times a seed will be fuzzed after it is selected. Higher seed energy results in more mutations and executions. By mutating seeds with a smaller number of recorded edges, IgorFuzz can process inputs faster and is more likely to discover even simpler PoCs. We use a greedy heuristic to allocate more energy to seeds with a shorter execution length.

We also dynamically allocate energy to different seeds based on their coverage profile. We assign more energy to seeds with shorter execution paths. Not only do we assign more energy to simpler seeds, but we also ensure that the assigned energy is biased towards larger reductions in coverage.

3.2.4 Fuzzing Output Selection Criteria. To explore different code regions, fuzzing typically employs a meta-heuristic optimization over the target's coverage space. During the search process, fuzzing naturally encounters solutions that diverge from the original objective. In the context of `IgorFuzz`, that objective is finding the simplest PoC that triggers the same initial bug. With the help of BitmapSize—which AFL++ [27] uses to show how many unique edges were activated—we pick the simplest PoC by finding the PoC whose bitmap size is the smallest. Despite limiting search to the vicinity of the original execution trace, it is inevitable that other bugs are triggered. To filter out those erroneous solutions, we rely on crash sites as the penultimate selection criterion, and among the remaining seeds, we select the PoC with the smallest bitmap size. According to our evaluation, `IgorFuzz` has a low probability of discovering crashes caused by a different bug (Section 5.5).

3.3 Test Case Similarity Measurement

Existing approaches for measuring test case similarity leverage information available at the crash site (e.g., the call stack, instruction pointer, register contents). A fundamental limitation of these approaches is that a bug may cause the program to crash in many different locations, resulting in an over-approximation of the unique bug count. Backward slicing may alleviate this problem by tracing the flow of the crashing condition to its origin. Unfortunately, backward slicing does not apply to all types of vulnerabilities (e.g., in a use-after-free vulnerability, the location where data is reused can be independent from where it is freed). Moreover, scaling backward slicing to large and complex program traces across 100,000 basic blocks is an unsolved problem. Studying the limitations of current approaches led us to the following insight: all crashes that are due to the same bug *must* have a (partially) overlapping execution trace. Furthermore, methods for analyzing the topological structure of the execution trace can be used to extract a signal for crashes of the same bug.

The program execution trace is a *sampled sequence* of program addresses sorted by their execution order. To calculate their similarity, existing approaches leverage *Levenshtein Distance* [82] and *Longest Common Subsequences* [36]. Unfortunately, program loops increase the difference between execution traces, limiting the utility of these techniques. Even if these execution traces come from crashes of the same root cause, their sequence similarity is low. However, collapsing the execution trace onto a graph means that control-flow similarity is no longer affected by different loop iteration counts. Thus, our approach uses graph topology to calculate similarity between traces.

3.3.1 Control-flow Graphs (CFG). Control-flow graphs describe the execution process of a program. A node in a CFG represents a program address, and edges connect two successive addresses. To construct the CFG, we sample the execution trace at a predefined granularity. On one end of the spectrum, fine-grain instruction-level traces introduce redundant nodes into the CFG, since edges between consecutive instructions are implicitly captured by the sequential nature of program execution. Additionally, recording instruction-level traces imposes scalability challenges, rendering the approach intractable. On the other end of the spectrum, function traces are easily collected, but they are coarse-grained and can overlook key behaviors in the program that capture the bug context. Basic block-level granularity provides a balance between precision and scalability, and we use it to construct the CFGs for similarity matching.

Execution traces contain noise that must be filtered. There are two types of noise: (i) code executed in external shared libraries (e.g., glibc); and (ii) superfluous code after the program's crashing point (e.g., ASan's crash handler). Since we focus on a specific target program, the execution trace of external code is unnecessary. Thus, we filter any addresses that are outside our target code. Sanitizers detect bugs early by introducing additional software guards. When a bug is detected, the sanitizer executes additional code to collect and report information. This information collection and reporting adds unnecessary noise and is filtered as well. We discuss how we filter these two types of noise (during the execution and after the bug trigger) in Section 4.

3.3.2 Calculating Graph Similarity. After recording traces, we construct a CFG for calculating graph similarity. Efficient graph similarity estimation has been studied extensively [44], and the results show that *kernel methods* are most suitable for estimating graph structure similarity. At present, most kernel methods for graph similarity estimation include two types: *graph embedding* and *graph kernel*. The former leverages traditional vector-wise kernel algorithms based on dimensional reduction of input graphs, which leads to a loss of structural information. The latter directly performs kernel algorithms in a high-dimensional Hilbert space, so that the structural information of graphs remains intact.

After surveying different graph kernels (e.g., labelled graphs, weighted graphs, or directed graphs), we found that the *Weisfeiler-Lehman Subtree Kernel* algorithm demonstrated the highest ability to differentiate test cases of varying root causes while assigning high similarity measurements to test cases of the same root cause. Thus, we adopt the Weisfeiler-Lehman Subtree Kernel to estimate the similarity between CFGs. The Weisfeiler-Lehman Subtree Kernel algorithm requires the nodes of a graph to be labelled. We use the basic block addresses (from the execution trace) as node labels.

3.4 Bug Clustering

Fuzzers are highly non-deterministic in how they discover bugs and paths through the target program. The fuzzer may discover one or more crashes for a bug, but it is not known *a priori* how many crashes it will find. Consequently, Igor cannot know how many bugs are discovered during a fuzzing campaign. For example, a given set of 100 crashes may map to 100 bugs with one crash each, one bug with 100 crashes, or anything in between. The key challenges for clustering are to (a) *discover the exact number of bugs*, and (b) *assign crashes to the correct bug*. As stated above, the number of expected clusters is not known and must be inferred during clustering. Our approach determines the number of clusters by running the clustering process multiple times, refining the assumed number of bugs based on a heuristic (see Section 3.4.2).

3.4.1 Data Characteristics. The Weisfeiler-Lehman Subtree Kernel algorithm produces a similarity matrix $\mathbf{M_s}$; let $\mathbf{M_d} = \mathbf{1} - \mathbf{M_s}$, where $\mathbf{M_d}$ is the corresponding distance matrix. We build our clustering algorithm on the two matrices.

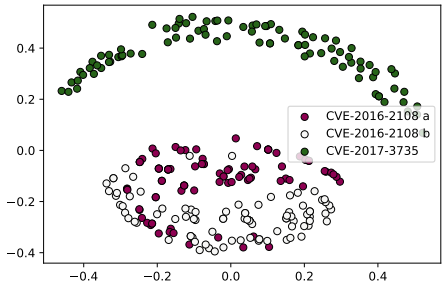


Figure 3: Distribution of samples for `x509`. The suffixes 'a' and 'b' indicate two call stacks for one CVE.

The matrices $\mathbf{M_s}$ and $\mathbf{M_d}$ are not sufficient to determine the distribution of samples. Multi-dimensional Scaling (MDS) [10] is an algorithm for dimensionality reduction. It projects high dimensional data to a lower dimensional space by finding abstract Cartesian coordinates in the lower dimensional space that keeps the distance between samples almost unchanged. We leverage MDS to visualize the distribution of samples in a two-dimensional plane.

We analyze distributions based on these visualizations. For example, Fig. 3 shows how samples of `x509` are distributed across the Cartesian plane. The shapes of clusters are not spherical, indicating that k-means-like methods are not suitable, and we therefore rely on alternative methods to recognize clusters of arbitrary shapes.

Igor leverages the *silhouette score* [63] to describe the quality of cluster structures. By simplifying test cases, the samples tend to have better cluster structure, see Fig. 7.

3.4.2 Clustering Algorithm. DBSCAN [26] is a density-based clustering algorithm that uses a similarity/distance matrix. The DBSCAN algorithm takes two parameters, ϵ and $MinPts$, that define the density threshold. The primary challenge is how to determine these parameters. Although heuristics exist [65, 67], changes in the density distribution of samples result in a failure of the clustering process. Alternative algorithms like OPTICS [3] and HDBSCAN* [12] perform hierarchical analysis to obtain better results. However, they still require predefined density descriptive parameters, which are must be tuned when analyzing new programs (because the samples' density distribution varies between programs).

We found that the *Spectral Clustering* [74] algorithm addresses the parameter-tuning challenges brought on by the high-dimensional information stored in the Weisfeiler-Lehman Subtree Kernel similarity matrices. Spectral Clustering takes only one parameter: the number of clusters. Given the number of clusters and a similarity matrix, Spectral Clustering builds a graph Laplacian on the basis of the similarity matrix. Eigenvectors of the graph Laplacian are calculated to realize dimensionality reduction. Then, the algorithm automatically groups samples into the designated number of clusters in a lower dimensional space.

Since the real number of clusters is unknown, we need a metric for evaluating the clustering result. For this purpose, we again use the silhouette score. Liu et al. [49] describe different clustering validation measures. In their study, they indicate that the optimal cluster number can be determined by maximizing the value of the silhouette score.

As the silhouette score is undefined when there is only one cluster, we assume that there are at least two clusters among the dataset. Then we enumerate the number of clusters to run the clustering process, and calculate the silhouette score of the result. Afterwards, we select the clustering result with the highest silhouette score. However, this approach may undercount the number of clusters. We therefore develop a heuristic to decide if we need to repeat clustering based on the number of full-length crash call stacks: for less than 20 call stacks, running the clustering process once suffices, while for more than 20 call stacks, we cluster twice. Tables 1 and 7 show our experimental results based on this heuristic.

3.4.3 Misclustering for single-bugs. The presence of tightly interspersed sub-clusters indicates that there is likely only one cluster. The minimum number of detectable clusters is two. If there is only one, then the number of clusters is raised artificially during enumeration to find a higher silhouette score. It is rare in practice for a fuzzing campaign to find hundreds of test cases for only one vulnerability. After thousands of CPU hours of fuzzing, we have never observed this case. In this case, using current crash grouping methods—e.g., stack bucketing—is sufficient.

To mitigate against this rare situation, Igor compares its cluster result with a call-stack based approach (e.g., `afl-collect`). If Igor reports more clusters than `afl-collect`, we indicate to the analyst that they should refer to the `afl-collect` results.

4 IMPLEMENTATION

Our prototype of Igor demonstrates the practical feasibility of our approach. We briefly explain important implementation details in the following sections. Complete source code is available at https://github.com/HexHive/Igor. We implement `IgorFuzz` on top of AFL++, record execution traces using Intel Pin, and develop several Python scripts that orchestrate `gdb` to select and filter execution traces. We construct CFGs from traces using NetworkX [32], visualize them with Graphviz [25], and calculate graph similarity using GraKeL [69]. The clustering phase leverages `scikit-learn` [58]. Our implementation consists of approximately 1,000 lines of C++ code and 2,500 lines of Python code, along with several small scripts.

Recording Execution Traces and Noise Filtering. We developed `smart-tracer`, a trace recorder based on Intel Pin that records execution traces at function call-, basic block-, and instruction-level granularity. During post-processing, we filter out function calls (a) into auxiliary code (e.g., calls into libc), and (b) that occur after the crash is triggered (e.g., sanitizer information collection).

CFG Similarity Metric. The filtered traces are first used to construct the CFG. Then, we use this CFG to calculate graph similarity using the Weisfeiler-Lehman Subtree Kernel algorithm.

Clustering. The clustering procedure runs up to fifteen times by default, enumerating the number of clusters from 2 to 16, and (re)calculating the silhouette score each time. This process outputs the clustering result with the highest silhouette score. Finally, a post-clustering scatter diagram is created to help analysts visually assess the quality of the clustering result.

5 EXPERIMENTAL EVALUATION

We evaluate Igor on 12 servers running Ubuntu 18.04 LTS, each with 200 GiB of RAM and an Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz with 40 cores. We source our benchmark dataset from 58.7 CPU-years of fuzzing campaigns. The Igor-specific evaluation took 6,143.7 CPU-hours and 30 human-days across all presented experiments. The `IgorFuzz` evaluation took 5,531 CPU-hours. Importantly, this is not necessarily indicative of the actual time required for minimization: we run `IgorFuzz` one hour for each sampled PoC (5,531 CPU-hours in total) to show the changes over time. In practice, we would let `IgorFuzz` run for 15 min per PoC (we demonstrate this in Section 5.4.1); thus minimizing all 5,531 PoCs requires 1,382.75 CPU-hours. Compared to the total cost of fuzzing, minimization is negligible and consumes less than 0.3 % of the length of a fuzzing campaign.

5.1 Benchmarks

We evaluate Igor on the Magma benchmark [33] and MoonLight dataset [34, 35]. These benchmarks contain 52 CVEs that belong to 14 target programs, containing more than 254,000 crashing PoCs that map to 39 unique bugs. We exclude four programs containing only a single CVE: `pdftotext` (*Poppler*), `exif` (*PHP*), `client` (*OpenSSL*), and `libxml2_xml_read _memory_fuzzer` (*LibXML*). As discussed in Section 3.4.3, Igor falls back to `afl-collect` in this case (we still report these results in Table 5). Additionally, we exclude (a) two CVEs that result in prohibitively large trace files (over 10 GiB per PoC), and (b) six CVEs that are duplicates, semantically equivalent, or partial fixes to one of the 39 unique bugs (see Section 6.3 for details). We augment these benchmarks with precise ground-truth data that verifies the root cause for each PoC.

Following the methodology outlined by Klees et al. [43] and Hazimeh et al. [33], we map crashes to bugs through the patches that fix the crash (when applied). Importantly, both the Magma and MoonLight datasets contain targets with multiple bugs. Unfortunately, multiple bugs (in a single target) may interfere with each other (e.g., a bug may mask/enable other bugs that would not manifest in a normal program execution). To minimize the risk of interference, we relabel our initial PoC dataset according to the patches that disable the crash. We show this process in Fig. 4.

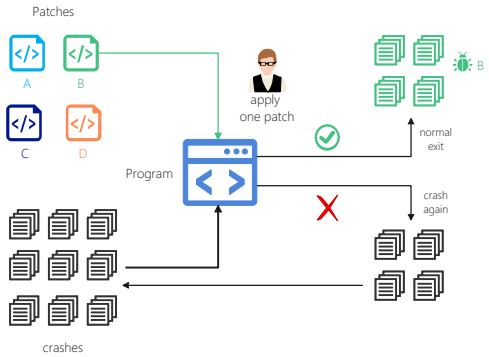


Figure 4: Tagging PoCs by applying patches.

Starting with an unpatched version of the target, we apply patches one at a time, each time processing the entire PoC corpus to flag changes in crash status. If a change is detected, that patch is considered a ground-truth label for the PoC, and is used for measuring the performance of our clustering method. We manually verified the completeness of patches in fixing the root cause of a crash. For each crash in our benchmark, we label root cause and call-stack hash. Our ground-truth data allows us to determine whether crashes are grouped correctly (i.e, if they correspond to specific CVEs).

5.2 Summary of Results

Table 1 shows the results of Igor's clustering evaluation with different `IgorFuzz` cut-off times. *Cut-off time* refers to the time we allow `IgorFuzz` to minimize a single PoC. Here we only report 15 min and 30 min cut-off times (due to space constraints) and provide results for the other cut-off times in Table 7 in Appendix D.

For Igor, we report clustering results for basic-block-level traces. We also evaluate Igor's clustering results at function-call (cheap and least precise) and instruction level (expensive and most precise) granularity. While the results are similar across the three tracing granularity, we see a correlation between better results and bugs with a larger number of crash sites. Specifically, our results indicate that Igor should switch to instruction-level tracing when the number of crash sites exceeds ~ 20 (e.g., for `xmllint` and `char2svg`).

While grouping crashes based on stack hashes is common practice, different tools use different stack depths. For example, `afl-collect` hashes all stack frames, honggfuzz leverages the last seven stack frames, and CERT BFF [38] uses the last five frames. We compare Igor against these three configurations and a baseline of a single stack frame (called "Top Frame" in Table 1). To measure classification quality, we calculate precision, recall, purity [2, 70], inverse purity [2] and F-measure [47, 70]. These metrics are all standard approaches from the machine learning community.

Our results show that—compared to Top Frame, BFF-5, honggfuzz, and `afl-collect`—Igor achieves the highest F-measure in 90 % of our experiments (and is 20 % off in the remaining 10 %) and achieves higher purity and inverse purity scores in most cases. From a quick glance at Table 1, it may appear that crash sites are the best grouping method, but crash sites are often not unique because different bugs crash at the same program location (see the unique bug addresses in the crash address column in Table 5).

We also assess the loss of precision caused by our bug count inference process (which may be inaccurate in itself). We rerun the clustering process with ground-truth bug count, and compare these counts to the results of clustering that relies on an inferred bug count. Table 7 (in Appendix D) shows that Igor's results are as accurate as the ground-truth bug counts.

5.3 Test Case Reduction

We evaluate `IgorFuzz` (over all 39 bugs) to demonstrate its ability to remove redundant paths traversed by each PoC. For each bug we use `afl-cmin` to select representative PoCs from our corpus and run `IgorFuzz` for 15 min. Per Section 5.4.1, we found that 15 min is sufficient for each minimization. Fig. 5 shows the *mean* edge count (i.e., the fuzzer's bitmap size) executed by the newly-minimized PoC, while Fig. 11 shows how `IgorFuzz` reduces traces across three dimensions (basic block counts, edge counts, and trace length). Due

Table 1: Evaluation results. *ASan rep(s), HO, SO, FP* and *HU* represent *Heap buffer Overflow, Stack buffer Overflow, Floating-Point exception* and *Heap Use-after-free* (according to `Address Sanitizer` reports), respectively. The *# of samp.* represents the number of samples in one experiment; *C. Igor* is the number of clusters Igor outputs; and *Top Frame, BFF-5, honggfuzz,* and `afl-collect` show crash grouping scores obtained by comparing call stacks of length 1, 5, 7, and full-length. *P, IP* and *F* are abbreviations of *Purity, Inverse Purity,* and *F-measure,* respectively. Finally, *cut-off time* represents the running time we limit `IgorFuzz`'s minimization for a single PoC. The best performing entries are highlighted.

Program	# bugs	ASan rep(s)	# samp.	# crash addr/uniq	# call stack	C. Igor	Top Frame (%)			BFF-5 (%)			honggfuzz (%)			afl-collect (%)			cut-off time (minutes)	Igor (%)		
							P	IP	F	P	IP	F	P	IP	F	P	IP	F		P	IP	F
pdfimages	3	HO, SO	410	12	23	3	100	48	63	100	47	62	100	47	62	100	47	62	15	100	100	100
						3													30	99	99	99
pdftoppm	3	HO, FP	161	3	5	2	100	100	100	100	95	98	100	95	98	100	95	98	15	69	100	80
						2													30	70	100	80
tiffcp	5	HO	991	6/3	20	6	85	89	80	88	76	74	88	67	68	88	67	68	15	100	98	99
						6													30	100	99	99
tiff2pdf	3	HO, HU	385	3/2	8	3	92	91	89	99	89	93	99	89	94	99	89	94	15	98	98	98
						3													30	98	98	98
x509	2	HO	150	2	3	2	100	100	100	100	100	100	100	75	83	100	75	83	15	100	100	100
						2													30	100	100	100
libpng_read_fuzzer	2	FP	150	2	3	2	100	72	81	100	72	81	100	72	81	100	72	81	15	100	100	100
						2													30	100	100	100
xmllint	8	HO, SO, HU	1581	25/14	901	14	78	59	64	78	55	62	78	55	62	79	3	4	15	96	67	77
						18													30	97	66	77
char2svg	5	HO, SO, HU	1087	21/20	67	8	100	66	72	100	14	20	100	14	20	100	14	20	15	100	67	79
						8													30	100	67	79
sox (MP3)	4	HO, SO	260	6	8	4	100	63	74	100	58	71	100	58	71	100	58	71	15	100	100	100
						4													30	100	100	100
sox (WAV)	4	HO, SO, FP	356	6	8	4	100	67	77	100	66	76	100	66	76	100	66	76	15	100	100	100
						4													30	100	100	100

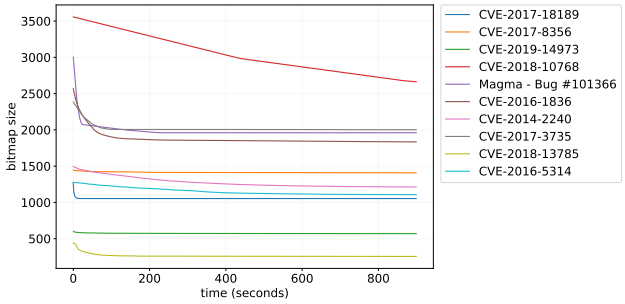


Figure 5: Bitmap size (number of edges) trend over time.

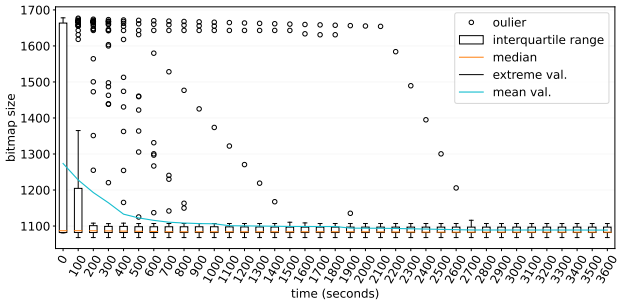


Figure 6: Boxplot of bitmap size for CVE-2016-5314.

to space constraints, we limit our results to these figures (the remaining results are available at https://github.com/HexHive/Igor).

Fig. 5 shows that the bitmap size monotonically decreases over time. We found that the length of the reduction process is proportional to the length of the program trace executed by the PoC. Larger bitmaps provide more opportunities for shrinkage, as evident from the topmost plots in Fig. 5. The larger the bitmap size of the initial PoC, the slower the compression process.

The mean values in Fig. 5 ignore outliers. Fig. 6 highlights how Igor reduces outliers for CVE-2016-5314. Fig. 6 shows that the interquartile range drops significantly after `IgorFuzz`'s trace reduction. Furthermore, the outliers' bitmap size gradually converges to the mean, and the number of outliers continues to decrease over time (until they disappear completely).

5.4 Graph Similarity After Minimization

We cluster and compare the original and reduced PoCs to assess whether `IgorFuzz` improves similarity analysis. This involves recording execution traces of the target program and calculating similarity based on the CFGs constructed from these traces.

Our results (across our ten target programs) demonstrate that coverage-reduction fuzzing makes PoCs more distinguishable. For example, Fig. 7 shows the distribution of CFGs before and after performing coverage-reduction fuzzing on `tiffcp`. The cluster contains four vulnerabilities. Per Fig. 7(a), because the bug-irrelevant paths have not been pruned, the test cases of different vulnerabilities are not distinguishable; they are combined with each other, resulting in nine clusters and a surplus of five misclassified bugs.

Clustering precision greatly improves after using `IgorFuzz` to shrink PoCs. Per Fig. 7(b), clustering shrunken PoCs leads to an accurate grouping of the 842 test cases into four clusters (based

on their root causes). Table 2 shows the change in silhouette score before and after reduction.

Data distribution affects the results of the cluster [67]. Unfortunately, this distribution of data cannot be predicted in many real-world scenarios. We study Igor's ability to process different data distributions, and our results show Igor achieves ideal results under different data distributions (see Appendix A).

5.4.1 Suitable Cut-off Time. The time/cost tradeoff is an important factor for determining the practicality of Igor. Given more time, `IgorFuzz` can generate more concise PoCs, and our results show that pruning more bug-irrelevant paths leads to more accurate clustering results. Our results also show that each time Igor's running time is doubled, we achieve a higher (mean) silhouette score (indicating more bug-irrelevant paths are pruned). But improving silhouette scores results in diminishing returns (considering performance of the second stage). After reaching a threshold, the cluster structure is highly identifiable and clustering results no longer improve.

According to our silhouette scores (Table 2), `IgorFuzz` increases the average silhouette score by 14.4 % during the first 15 min (the 0 min column corresponds to the time before any reduction has occurred). The next two intervals increase the average silhouette score by only 1.8 % and 0.5 %, respectively. This yields minimal benefits for clustering. We use the minimized PoCs generated by these three cut-off times for clustering (see Tables 1 and 7). These results show that although longer times will give us more precise results, it is sufficient to run `IgorFuzz` with 15 min. Therefore, we suggest 15 min as the default cut-off time.

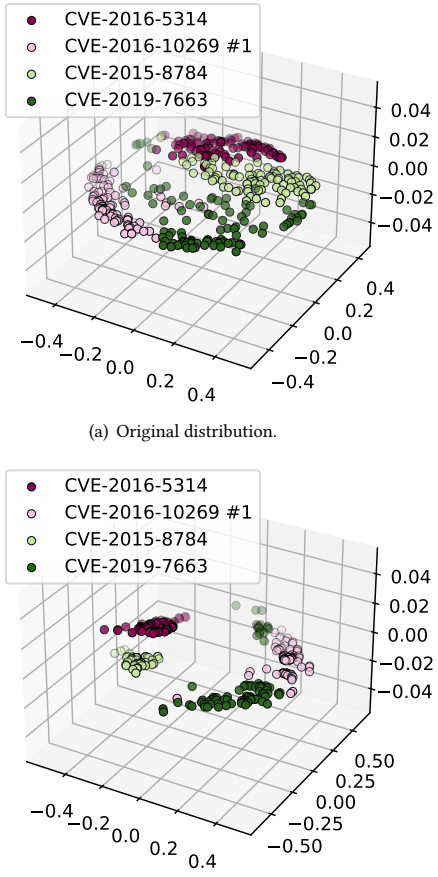
Table 2: Silhouette scores.

Program	0 min (%)	15 min (%)	30 min (%)	60 min (%)
`pdfimages`	52.9	70.2	70.0	69.4
`pdftoppm`	59.0	80.2	80.0	79.0
`tiffcp`	33.2	70.2	73.9	77.4
`tiff2pdf`	66.0	70.6	70.7	71.2
`x509`	76.9	94.1	94.3	94.6
`libpng_read_fuzzer`	68.5	96.6	96.6	97.7
`xmllint`	38.3	49.1	55.3	54.7
`char2svg`	44.5	53.7	53.8	54.0
`sox` (MP3)	78.1	74.6	81.1	81.3
`sox` (WAV)	83.1	85.0	86.6	88.0
Average	60.0	74.4	76.2	76.7

5.5 Verifying Minimization Results

Here we answer a key minimization question: does a minimized PoC trigger the same bug as the original PoC? We label a result a *false positive* if the minimization process discovers a new bug with a different root cause *and* this root cause replaces the original test case. We verify minimization results via the process described in Section 5.1 (i.e., by successively applying patches).

We designed two experiments from two perspectives. First, we verify whether a minimized PoC introduces an error. We recorded *zero* false positives out of our 5,535 minimized PoCs. Secondly, we checked whether errors (i.e., PoCs triggering a new bug) were generated during fuzzing. If there are many error seeds in the queue, `IgorFuzz` may be misled by these error seeds, and generate more



(a) Original distribution.

(b) Distribution after reduction.

Figure 7: Data distribution before and after reduction.

error seeds during the fuzzing process. The number of queues containing error seeds (namely, "error queues") is shown in Fig. 8. This figure shows the number of error queues among all the queues of their corresponding programs. Only 2.10 % of all 5,531 queues contain errors. This shows that `IgorFuzz` has a low probability of introducing errors during the reduction process. Further, we also studied the proportion of error seeds in the error queues and found, with the exception of `xmllint`, this proportion is less than 10 % (see Appendix D). This means `IgorFuzz` will not waste time on error seeds. Although errors appeared during minimization in these programs, it does not affect the correctness of `IgorFuzz`'s output (due to our seed selection principles, detailed in Section 3.2.4). According to our evaluation results, all errors were filtered out successfully.

6 CASE STUDIES

We now present three case studies that—while challenging for classic crash grouping techniques—demonstrate the effectiveness of Igor. These case studies include: (i) assessing the accuracy of CVEs; (ii) highlighting rare crashes; and (iii) detecting semantically equivalent bugs.

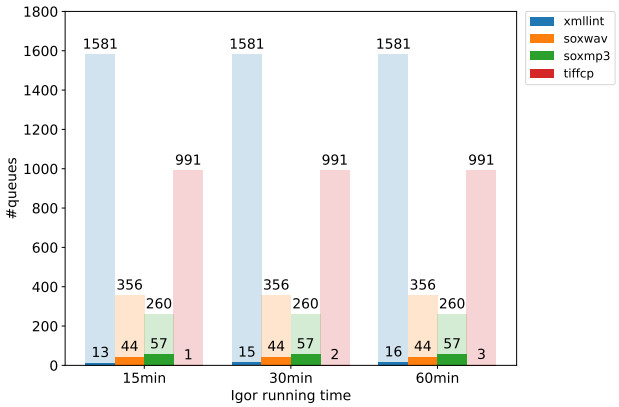


Figure 8: Queue error rate.

6.1 Assessing CVE Accuracy

Software maintainers manually analyze reported PoCs, assigning CVEs according to their root cause. However, a recent study [48] shows that even software developers misjudge root causes when faced with a large number of PoCs, resulting in imprecise CVEs where either different vulnerabilities are assigned the same CVE or different CVEs are assigned to the same vulnerability. Igor's benchmark includes both cases.

For CVE-2016-10269 #1 and CVE-2016-10269 #2 (`tiffcp`), we sampled 400 PoCs and grouped them according to the root cause. Igor clusters these PoCs into two distinct groups. After verification, we discovered that there are indeed two different vulnerabilities. As the functions executed before the crash are similar, the developer mistakenly classified these two vulnerabilities into one category.

For CVE-2019-8354 and CVE-2019-8366 (SoX), Igor clusters 57 PoCs into a single group. These two CVEs have different crash addresses and call stacks, but the developer confirmed that these two vulnerabilities are indeed caused by the same root cause, the assigned CVEs are duplicates.[3]

For CVE-2015-9290 and CVE-2015-9381 (`char2svg`), we found that their patches both partially fix the same vulnerability. Igor correctly clusters the corresponding 41 PoCs into a single group.

These three scenarios demonstrate how Igor successfully highlighted the inaccuracy of assigned CVEs (inaccurate CVEs are labelled with superscripts in Table 5).

6.2 High-value Needles in Test Case Haystacks

Fuzzers cannot guarantee a similar number of PoCs for each bug. For bugs that are harder to find (e.g., due to deeper code depth, or harder trigger conditions), the number of PoCs generated by the fuzzer is often small. Such rarely-triggered bugs may be lost because crash grouping may wrongly merge them with other bugs.

Igor ensures that such rare PoCs are correctly analyzed. By first bucketing PoCs based on the crash address, Igor detects crash locations with few PoCs and uses AFL's crash mode to amplify them. In our current implementation, Igor amplifies the number of PoCs at crash locations with fewer than ten PoCs to at least 50.

As an example, we study CVE-2016-5314 (`tiffcp`). Consider our analysis for trial 3 in Table 3. If we instead limit the number of PoCs for CVE-2016-5314 to three, then this CVE is subsumed by another CVE. When we amplify the three test cases to 50, the CVE is successfully separated.

[3]https://gitlab.alpinelinux.org/alpine/aports/-/issues/10523

6.3 Detecting Semantically Equivalent Bugs

Our benchmark distinguishes vulnerabilities based on unique patches. If the patch used to eliminate the failing test case is different, it is considered to be a different vulnerability. Based on this rule, we found that Igor's result misclassifies PoCs for `char2svg`. The `char2svg` PoCs come from eight CVEs with a different patch for each CVE, indicating that these are eight different bugs (see Table 5).

However, we discovered that Igor groups the PoCs from CVE-2014-9663, CVE-2014-9669, and CVE-2015-9383 into a single group. We reviewed the corresponding code and patches in `char2svg`, and found that these three CVEs share the same semantic root cause. Specifically, we found that all three CVEs check whether `FT_INVALID` is too short to avoid heap-buffer-overflows and all three patches are in the same source file. There are six similarly named functions that validate the input (e.g., `tt_cmap4_validate`, `tt_cmap8_validate`, `tt_cmap10_validate`). All six functions are buggy, and share the same root cause. The developer first committed the patch for CVE-2014-9669, in which five of the six vulnerable functions were fixed. The forgotten vulnerable function, `tt_cmap4_validate`, was fixed in the patch for CVE-2014-9663 ten days later. However, the patch for CVE-2014-9669 is an incomplete fix, because it misses one aspect of `FT_INVALID`. This went unnoticed until CVE-2015-9383 was reported, and the patch for CVE-2015-9383 finally correctly fixes this bug.

We argue that these three CVEs fix the same single bug from a semantic point of view: the same bug condition is shared among all three CVEs, and can be described as "as long as `FT_INVALID` is too short, it will cause the same variable be over-flowed". Igor noticed that the control flow of the three CVE are indeed similar when they trigger the bug and, thus, Igor grouped them into a single bucket.

7 LIMITATIONS

Using silhouette score to determine the number of clusters is not guaranteed to distinguish all clusters. Therefore, after the initial clustering is completed, it is necessary to manually review the clustering results. If there are still gaps between samples within the same cluster, it indicates that a second clustering process is required to obtain more reasonable clusters. Additionally, when there is only one bug, *intra*-class differences are regarded as *inter*-class differences. The clustering algorithm then may divide the input samples into a large number of clusters. Igor will invoke `afl-collect` to group the crashes when it finds the group number given by itself is larger than that of `afl-collect`. In this case, Igor falls back to the precision of `afl-collect`.

`IgorFuzz` requires more computing resources than simple stack hashing. We argue that the increased precision of Igor is worth the additional computation cost (less than 0.3 % in our experiments) due to the large amount of saved developer cost.

8 RELATED WORK

Crash bucketing builds on a diverse background in fuzzing, test case reduction, crash grouping, crash deduplication, and fault localization. We highlight how Igor compares against these areas.

8.1 Test Case Reduction

`Afl-tmin` [83], a popular *test case minimizer*, removes as much data as possible from a seed while keeping the target in a crashing state. While fast and easy to use, it has limitations: although it can significantly reduce the input size, it cannot effectively reduce the number of edges that the program executes before crashing. Compared to `afl-tmin`, our method emphasizes the reduction of the execution trace instead of input size, which means the vulnerability trigger becomes more direct after the reduction. MacIver and Donaldson [53] presented internal reduction by manipulating the behavior of the generator that produced them. Its input and purpose is different from ours, as it tries to shrink redundant operation of input and keep the same behavior, while we want to execute fewer edges before the program crashes.

8.2 Crash Grouping and Deduplication

Crash grouping reduces the crash analysis cost by leveraging specific metrics to measure the affinity between test cases, grouping similar test cases in one group.

Chen et al. [16] calculate edit distances and coverage profiles between test cases to group them. Both van Tonder et al. [72] and Pham et al. [60] utilize symbolic execution technique to collect crash constraints to assist grouping. Molnar et al. [55] introduces *fuzzy stack hashing* by collecting multiple crash metrics and then hashing these information together as a grouping metric. Holmes and Groce [37] proposes an alternative to crash grouping: if two failing test cases can be fixed by applying the same mutant, those two tests are likely related to the same root cause.

CrashLocator [79] discovers faulty functions that do not reside in the crash stack by expanding the given crash stack based on a function call graph. Although this method also takes advantage of static call graphs, it only uses them to recover traces, while our method uses dynamic call graphs to measure the similarity of PoCs.

ReBucket [21] proposed a method for clustering duplicate crash reports based on call stack similarity, the problem with this method is that it only relies on the local information at the time of the crash, when categorizing vulnerabilities with more number of call stacks at the time of the crash, it is easy to divide the PoCs that have same root cause into multiple different groups. Besides, the inputs of ReBucket are crash reports, but our inputs are PoCs, so the application scenarios of the two systems are different. SPIRiT [73] combines various methodologies to compute a specialized test case distance measurement, and drives a customized hierarchical test suite clustering algorithm that groups similar test cases together.

The primary difference between our method and the existing methods for crash grouping is that Igor evaluates the similarity of executions of a program from a global perspective by utilizing the simplified execution trace, while the aforementioned ones mainly focus on local features of programs.

8.3 Fault Localization

The purpose of fault localization is to determine the root cause of bugs through crashes or core dump files. Statistical methods and Delta debugging are the main two technical solutions. Aurora [7] is a representative work of fault location based on statistical methods. Similar to ours, it uses crash mode to increase the number of test cases. The difference is that Aurora collects both crash and non-crash test cases and locates root cause by comparing them. The risk of this method is that crash mode does not guarantee that the generated test cases are still caused by the same vulnerability. When the crash sample is caused by other vulnerabilities, this method will cause false positive. Xu et al. [80] propose a method for locating the root cause through data flow analysis, the key of this method is to recover the data flow from core dumps automatically to provide richer debugging information and reduce the difficulty of manual analysis. Zamfir et al. [84] uses execution traces to help developers locate the root cause. CrashLocator [79] locates faults based on call stack. REPT [19] uses reverse debug to locate fault. Kim [42] uses multiple crashes to diagnosis root cause from the perspective of crash graphs.

9 CONCLUSION

Fuzzing has become ubiquitous and is today's key driver for bug discovery. Unfortunately, the number of reported crashes outpaces developers' ability to triage and fix bugs. Nonetheless, the number of crashes is generally an over-approximation of the total number of bugs, and crash grouping can drastically reduce the burden on developers. Existing approaches are light-weight but prone to misclassification. We argue for trading little computational time to effectively reduce the number of crashes by two orders of magnitude, close to the number of real bugs, lightening developer workload.

Our approach, Igor, prunes bug-irrelevant paths and calculates the shortest bug-triggering path using a novel coverage reduction fuzzing approach. Crashes are then grouped based on the topological similarities between the CFGs of minimized execution traces.

Our evaluation demonstrates that Igor outperforms existing approaches, resulting in the most precise bug clusters for 90 % of our evaluated programs. Based on a ground-truth comparison, we show that Igor groups crashes of various root causes and vulnerability types precisely. The open-source prototype of Igor, `IgorFuzz`, and our labelled ground-truth benchmark are available at https://github.com/HexHive/Igor.

10 ACKNOWLEDGEMENTS

The authors would like to thank: Adrian Herrera and the anonymous reviewers, for their careful feedback along with the opportunity for a major revision which greatly improved the clarity of this paper; Ruilin Li and Shuitao Gan, for their helpful feedback; Zheyu Jiang and Zhiwei Li, for their valuable support during the evaluation. This project has received funding from the European Research Council (ERC) under grant agreement No. 850868, DARPA HR001119S0089-AMP-FP-034, ONR award N00014-18-1-2674, and National Natural Science Foundation of China under grant 61772308, 61972224, and U1736209. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Karan Aggarwal, Finbarr Timbers, Tanner Rutgers, Abram Hindle, Eleni Stroulia, and Russell Greiner. 2017. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process* 29, 3 (2017).

[2] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. 2009. A comparison of extrinsic clustering evaluation metrics based on formal constraints. (2009), 461–486.

[3] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod record* 28, 2 (1999).

[4] Apple. 2010. Diagnosing Issues Using Crash Reports and Device Logs. https://developer.apple.com/library/archive/technotes/tn2004/tn2123.html

[5] 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)*. The Internet Society.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018).

[7] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[8] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[9] 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)*. ACM.

[10] Ingwer Borg and Patrick J.F. Groenen. 2005. . Springer-Verlag.

[11] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. 2005. Automated problem determination using call-stack matching. *Journal of Network and Systems Management* 13, 2 (2005).

[12] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer.

[13] Brian Chan, Ying Zou, Ahmed E Hassan, and Anand Sinha. 2010. Visualizing the results of field testing. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE.

[14] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[15] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[16] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[17] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society.

[18] James Clause and Alessandro Orso. 2009. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 249–260.

[19] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association.

[20] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. 2016. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE.

[21] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society.

[22] Sanjeev Das, Kedrian James, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2020. A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-) Behavior to Discover New Bugs. In *Annual Computer Security Applications Conference*. 345–359.

[23] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE.

[24] William Drozd and Michael D Wagner. 2018. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490* (2018).

[25] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2004. Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. In *Graph Drawing Software*. Springer-Verlag.

[26] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press.

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[28] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[29] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM.

[30] google. [n. d.]. clusterFuzz. https://google.github.io/clusterfuzz (accessed January, 2021).

[31] google. [n. d.]. Syzbot dashboard. https://syzkaller.appspot.com (accessed January, 2021).

[32] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using networkx. In *In Proceedings of the 7th Python in Science Conference*.

[33] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3 (2020).

[34] Adrian Herrera, Hendra Gunadi, Liam Hayes, Shane Magrath, Felix Friedlander, Maggi Sebastian, Michael Norrish, and Antony L. Hosking. 2020. Corpus Distillation for Effective Fuzzing: A Comparative Evaluation. (2020). arXiv:1905.13055

[35] 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* (Virtual, Denmark) *(ISSTA)*. ACM, 230–243. https://doi.org/10.1145/3460319.3464795

[36] Daniel S Hirschberg. 1977. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)* 24, 4 (1977).

[37] Josie Holmes and Alex Groce. 2020. Using mutants to help developers distinguish and debug (compiler) faults. *Software Testing, Verification and Reliability* 30, 2 (2020).

[38] Allen D Householder and Jonathan M Foote. 2012. *Probability-based parameter selection for black-box fuzz testing*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[39] Hyeon-Gu Jeon, Seong-Kyun Mok, and Eun-Sun Cho. 2017. Automated crash filtering using interprocedural static analysis for binary codes. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE.

[40] Richard M. Karp. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*. Springer.

[41] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*.

[42] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. IEEE Computer Society.

[43] 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)*. ACM.

[44] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. 2011. Algorithms for graph similarity and subgraph matching. In *Proc. Ecol. Inference Conf*, Vol. 17.

[45] Nils M. Kriege, Pierre-Louis Giscard, and Richard C. Wilson. 2017. On Valid Optimal Assignment Kernels and Applications to Graph Classification. (2017). arXiv:1606.01141

[46] Joseph B Kruskal. 1983. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review* 25, 2 (1983).

[47] Bjornar Larsen and Chinatsu Aone. 1999. Fast and effective text mining using linear-time document clustering. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*.

[48] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1547–1559.

[49] Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. 2010. Understanding of Internal Clustering Validation Measures. In *International Conference on Data Mining*. IEEE Computer Society.

[50] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Vol. 14. USENIX Association.
[51] LLVM Developers. 2019. libFuzzer: A Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html
[52] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005).
[53] David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
[54] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering*. ACM.
[55] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in X86 Binary Linux Programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*. USENIX Association.
[56] Mozilla. 2012. Crash Reports. http://crash-stats.mozilla.com
[57] Pradeep Padala. 2002. Playing with ptrace, Part I. *Linux Journal* 2002, 103 (2002).
[58] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011).
[59] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing {USB} Drivers by Device Emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.
[60] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing Failing Tests via Symbolic Analysis. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering*, Vol. 10202. Springer-Verlag.
[61] Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*.
[62] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17.
[63] Peter J. Rousseeuw. 1987. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *J. Comput. Appl. Math.* 20, 1 (1987).
[64] Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, and Alf Larsson. 2017. Durfex: a feature extraction technique for efficient detection of duplicate bug reports. In *2017 IEEE international conference on software quality, reliability and security (QRS)*. IEEE.
[65] Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 1998. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery* 2, 2 (1998).
[66] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE.
[67] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems* 42, 3 (2017).
[68] Kostya Serebryany. 2017. OSS-Fuzz: Google's continuous fuzzing service for open source software. (2017).
[69] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2020. GraKeL: A Graph Kernel Library in Python. *Journal of Machine Learning Research* 21, 54 (2020).
[70] Michael Steinbach, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. (2000).
[71] Robert Swiecki. 2016. honggfuzz. http://honggfuzz.com/
[72] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM.
[73] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. 2009. Test Case Comparison and Clustering Using Program Profiles and Static Execution. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM.
[74] Ulrike von Luxburg. 2007. A Tutorial on Spectral Clustering. (2007). arXiv:0711.0189
[75] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[76] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.
[77] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. 2013. RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Transactions on Internet and Information Systems (TIIS)* 7, 8 (2013).
[78] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*.
[79] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM.
[80] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association.
[81] 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 *29th {USENIX} Security Symposium ({USENIX} Security 20)*.
[82] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007).
[83] Michał Zalewski. 2015. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl
[84] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. 2013. Automated Debugging for Arbitrarily Long Executions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association.
[85] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002).

A BUG/TEST CASE DISTRIBUTION CLUSTERING

Clustering precision is impacted by the data distribution [67]. However, the number of crashes and vulnerabilities is unknown for each crash grouping task, which means that we cannot guarantee a uniform number distribution for crash grouping. Therefore, whether accurate clustering results can still be obtained under different data distributions is a key indicator to measure the effectiveness of our method. Tables 3 and 4 show the clustering results of Igor under different test case distributions.

Selecting `tiffcp` as the target program, we alter the number of test cases of five bugs respectively, the test case distribution in trial 1 is the same as what it is in Table 1, the distribution in trial 2 is roughly inverse to trial 1, with a decrease in total number, while in trial 3, the test cases are evenly distributed. Based on trial 3, we alter the number of bugs. The results show that Igor is stable against varying test case distributions.

Table 3: Clustering results of `tiffcp` with different sample distribution.

Bug	# samples		
	Trial 1	Trial 2	Trial 3
CVE-2016-5314	341	50	100
CVE-2016-10269 #1	251	90	100
CVE-2016-10269 #2	149	200	100
CVE-2015-8784	50	278	100
CVE-2019-7663	200	100	100
Result			
Sum	991	718	500
Purity (%)	99.4	99.4	99.6
Inverse Purity (%)	99.4	99.4	99.6
F-measure (%)	99.4	99.4	99.6

B DATA SET STATISTICS

We surveyed our dataset and counted the number of unique crash addresses and call stacks. These results are presented in Table 5.

Table 4: Clustering results of `tiffcp` with randomly sampled bugs. The notation A-E represent the bug ID of the corresponding position in Table 5. The field *clusters* is the number of clusters output by Igor, fields *P*, *IP* and *F* are the same as Table 1.

Trial	Bugs					Statistics			
	A	B	C	D	E	# clusters	P (%)	IP (%)	F (%)
1	✓	✓	✓	✓	✓	5	99.6	99.6	99.6
2	✓	✓		✓	✓	4	99.7	99.7	99.7
3		✓	✓		✓	3	99.3	99.3	99.3
4	✓			✓		2	100	100	100

C IGORFUZZ MITIGATES AFL-TMIN LIMITATION

As a dual-phase crash deduplication technique, Igor requires that the control flow of each PoC is concise after the first phase, so that the second phase can get more precise clustering results. Clustering directly after `afl-tmin`, i.e., without `IgorFuzz` results in low precision as `afl-tmin` is non-monotonic and provides inadequate control flow reduction. `IgorFuzz` mitigates these limitations.

During our evaluation, we found that `afl-tmin` may increase a PoC's bitmap size, e.g., in *OpenSSL*'s `x509` corpus (CVE-2016-2108). We selected ten seeds with the largest change in bitmap size being processed by `afl-tmin` to demonstrate this phenomenon. As shown in Fig. 9, the bitmap size of the ten PoCs rose in the first 20 seconds, which corresponds to the reduction process of `afl-tmin`. This indicates that as `afl-tmin` deletes bytes from a PoC file, the corresponding bitmap size does not always decrease monotonically. If we pass the `afl-tmin`-reduced PoCs to the second phase of Igor, we may worsen results because the control flow of the PoCs contains additional bug-irrelevant paths.

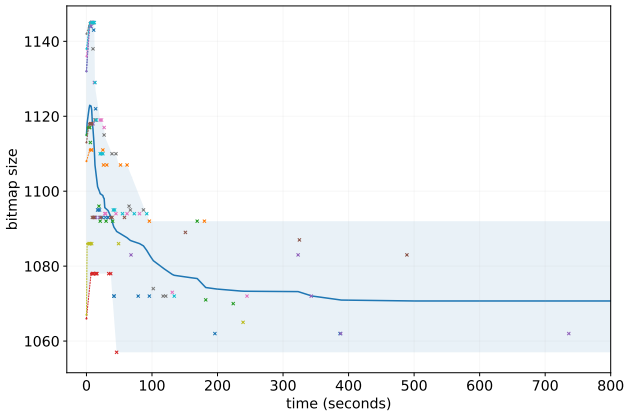


Figure 9: `Afl-tmin`'s side effect and Igor's ability to mitigate against it. The colored dots show individual test cases, the blue line indicates mean bitmap size and light blue the error band.

However, `IgorFuzz` mitigates this limitation. The blue line in the same figure shows mean bitmap size of the ten PoCs during reduction. Taking the `afl-tmin`-reduced PoCs as input, `IgorFuzz` reduced the bitmap size monotonically, and all of the ten PoCs finally got smaller bitmap size than their original size.

In summary, `afl-tmin` sometimes increases the bitmap size, which contradicts the requirement of Igor's first phase. `IgorFuzz` mitigates this issue by focusing on bitmap size reduction rather than input size reduction, making it more practical to do classification.

D IGORFUZZ DOES NOT WASTE TIME ON ERROR SEEDS

To study the minimization process of error introducing PoCs, we explicitly selected all the queues generated during the minimization process of the error PoCs, analyzing the change in the proportion of the error seed in all seeds over time. According to the evaluation result shown in Fig. 10, the proportion of error seeds is small, which accounts for less than 10 % in three programs, and the lowest is tiffcp, which is only 0.93 %. Except for xmllint, the proportion of erroneous seeds decreases over time, which means the effect of error seed on minimization is ever decreasing and minimization mainly explores the paths near the original vulnerability.

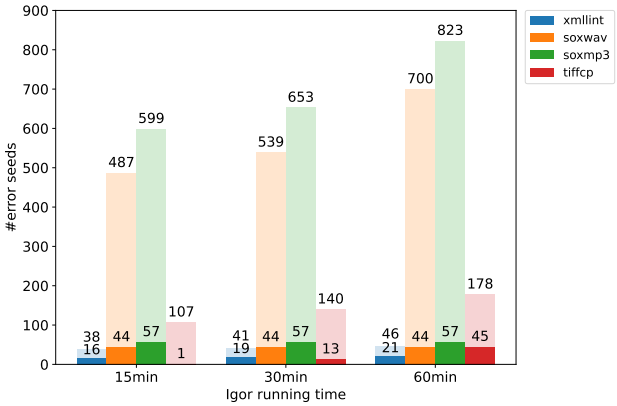


Figure 10: Seed error rate. In this figure, the bars in faded colors indicate total number of generated seeds in error queues (including normal seeds and error seeds); the bars in dark colors are total number of error seeds in error queues.

Error seeds occupy a significant proportion in the queue of xmllint, but the proportion decreases over time. We analyzed the reasons for the relatively high error seed in the queue of xmllint: The length of the PoCs from xmllint is much shorter than the length of PoCs of other programs, and the code depth of the vulnerability is relatively shallow. The short seed length combined with shallow bugs makes it likely for a fuzzer to discover alternate bugs through only small mutations.

Still, error seeds occupy a relatively small proportion in the queues overall and individually, which means that they have a small probability of being further mutated. After manual debugging, we found that the crash addresses of these error seeds are all different from those of the original PoCs. Therefore, even if a small part of the error seed is generated during the minimization, all these errors will be eliminated by our selection criteria (see details in Section 3.2.4).

Table 5: Statistics of unique crash addresses and unique call stacks. When the crash address count is given as n/m, n is the number of crash addresses, while m is the number of non-unique crash addresses. The bug IDs of a specific program with superscript * or $^+$ share the same underlying vulnerability. There are two unique underlying bugs assigned the same CVE—*CVE-2016-10269*. We use #1 and #2 to distinguish them.

Target	Program	Bug ID	Crash address count	Call stack count
Poppler	pdftoppm	CVE-2017-14617	1	1
		CVE-2019-7310	1	3
		Bug #101366	1	1
	pdfimages	CVE-2017-9865	1	1
		CVE-2018-10768	1	1
		CVE-2019-7310	1	2
		CVE-2017-9776	10	21
libtiff	tiffcp[4]	CVE-2015-8784	1	1
		CVE-2019-7663	2/2	4
		CVE-2016-10269 #1	3/2	5
		CVE-2016-10269 #2	2/2	3
		CVE-2016-5314	1	7
PHP	exif	CVE-2018-14883	13	50
OpenSSL	client	CVE-2016-6309	1	1
	x509	CVE-2016-2108	1	2
		CVE-2017-3735	1	1
libxml2	libxml2_xml_read_memory_fuzzer	CVE-2017-9047	3	28
libpng	libpng_read_fuzzer	CVE-2013-6954	1	2
		CVE-2018-13785	1	1
libxml2	xmllint	CVE-2015-8317	15/10	204
		CVE-2015-7497	1/1	198
		CVE-2016-1835	1	3
		CVE-2016-1836	1/1	5
		CVE-2016-1762	6/3	245
		CVE-2016-3627	3	3
		CVE-2015-7942	1	1
		CVE-2015-7499*	1/1	1
		CVE-2015-7498*	7/6	242
libtiff	tiff2pdf	CVE-2019-14973	3/1	6
		CVE-2017-17973	1/1	1
		Bug #C	1	1
Poppler	pdftotext	CVE-2019-12293	1	3
FreeType	char2svg	CVE-2014-9663*	1	4
		CVE-2015-9290+	1	1
		CVE-2014-9658	2	8
		CVE-2014-9669*	11/1	23
		CVE-2014-2240	1	23
		CVE-2014-9659	2	3
		CVE-2015-9383*	3/1	4
		CVE-2015-9381+	1	1
SoX(MP3)	sox	CVE-2019-8355	2	3
		CVE-2019-8357	5/2	5
		CVE-2019-8354*	1/1	1
		CVE-2019-8356*	1/1	2
		CVE-2017-18189	1	1
		CVE-2019-13590	1	1
SoX(WAV)	sox	CVE-2019-8355	2	3
		CVE-2019-8357	5/1	5
		CVE-2019-8354*	1/1	1
		CVE-2019-8356*	1/1	2
		CVE-2017-11332	1	1
		CVE-2017-18189	1	1

Table 6: PoC shrink rate with `afl-tmin`: PoC length, bitmap size, edges hit count, and side effects are listed.

Program	Bug	# Samples	Median			Mean			Variance			Negative	
			len %	map %	hit %	len %	map %	hit %	len	map	hit	map %	hit %
x509	CVE-2016-2108	100	24.14	4.94	6.28	41.10	4.83	22.57	0.165	0.002	0.103	15	12
	CVE-2017-3735	100	0	15.61	7.13	3.25	15.73	6.57	0.021	0.001	0.006	0	19
libpng_read_fuzzer	CVE-2018-13785	80	1.23	17.48	1.93	2.81	14.43	3.80	0.010	0.009	0.034	1.25	36.25
	CVE-2013-6954	100	14.37	0.13	-0.69	21.58	1.33	-1.27	0.054	0.0009	0.016	18	65
pdftoppm	Bug #101366	91	99.11	5.88	36.62	82.45	12.17	33.97	0.118	0.021	0.090	15.18	12.04
	CVE-2019-7310	100	0	-1.06	16.12	0	0.85	11.11	0	0.006	0.088	53.5	39.5
	CVE-2017-14617	97	96.05	5.81	32.35	96.31	8.06	31.36	0.0001	0.003	0.030	2.53	4.56
pdfimages	CVE-2017-9865	83	98.20	13.52	12.57	65.18	12.41	5.50	0.199	0.003	0.049	1.20	34.93
tiffcp	CVE-2016-5314	341	0.78	11.85	18.12	14.40	12.56	19.33	0.086	0.002	0.006	0	0
	CVE-2016-10269 #1	251	10.71	16.32	24.15	14.22	16.16	24.97	0.031	0.001	0.002	0	0
	CVE-2016-10269 #2	149	1.25	15.08	20.20	10.50	15.13	19.85	0.045	0.001	0.002	0	0
	CVE-2015-8784	50	1.25	15.56	21.08	13.57	19.76	25.54	0.066	0.012	0.026	0	2
	CVE-2019-7663	200	10.71	20.23	32.96	37.33	19.72	38.39	0.134	0.004	0.024	1.5	0

Table 7: Evaluation results (0 and 60 min `IgorFuzz` running time, the "*" in cut-off time column indicates clustering results that ground-truth bug labels are assigned).

Program	# bugs	ASan rep(s)	# samp.	# crash addr/uniq	# call stack	C. Igor	Top Frame (%)			BFF-5 (%)			honggfuzz (%)			afl-collect (%)			cut-off time (minutes)	Igor (%)		
							P	IP	F	P	IP	F	P	IP	F	P	IP	F		P	IP	F
pdfimages	3	HO, SO	410	12	23	2 3	100	48	63	100	47	62	100	47	62	100	47	62	0 60 *	88 100 99	94 100 99	87 100 99
pdftoppm	3	HO, FP	161	3	5	2 2	100	100	100	100	95	98	100	95	98	100	95	98	0 60 *	70 69 100	99 100 100	79 80 100
tiffcp	5	HO	991	6/3	20	17 5	85	89	80	88	76	74	88	67	68	88	67	68	0 60 *	98 100 90	65 100 90	77 100 92
tiff2pdf	3	HO, HU	385	3/2	8	3 3	92	91	89	99	89	93	99	89	94	99	89	94	0 60 *	98 98 98	98 98 98	98 98 98
x509	2	HO	150	2	3	2 2	100	100	100	100	100	100	100	75	83	100	75	83	0 60 *	100 100 100	100 100 100	100 100 100
libpng_read_fuzzer	2	FP	150	2	3	2 2	100	72	81	100	72	81	100	72	81	100	72	81	0 60 *	100 100 100	100 100 100	100 100 100
xmllint	8	HO, SO, HU	1581	25/14	901	4 19	78	59	64	78	55	62	78	55	62	79	3	4	0 60 *	72 97 94	95 63 74	77 74 82
char2svg	5	HO, SO, HU	1087	21/20	67	6 9	100	66	72	100	14	20	100	14	20	100	14	20	0 60 *	90 100 86	67 67 67	71 79 69
sox (MP3)	4	HO, SO	260	6	8	4 4	100	63	74	100	58	71	100	58	71	100	58	71	0 60 *	100 100 100	100 100 100	100 100 100
sox (WAV)	4	HO, SO, FP	356	6	8	4 4	100	67	77	100	66	76	100	66	76	100	66	76	0 60 *	100 100 100	100 100 100	100 100 100

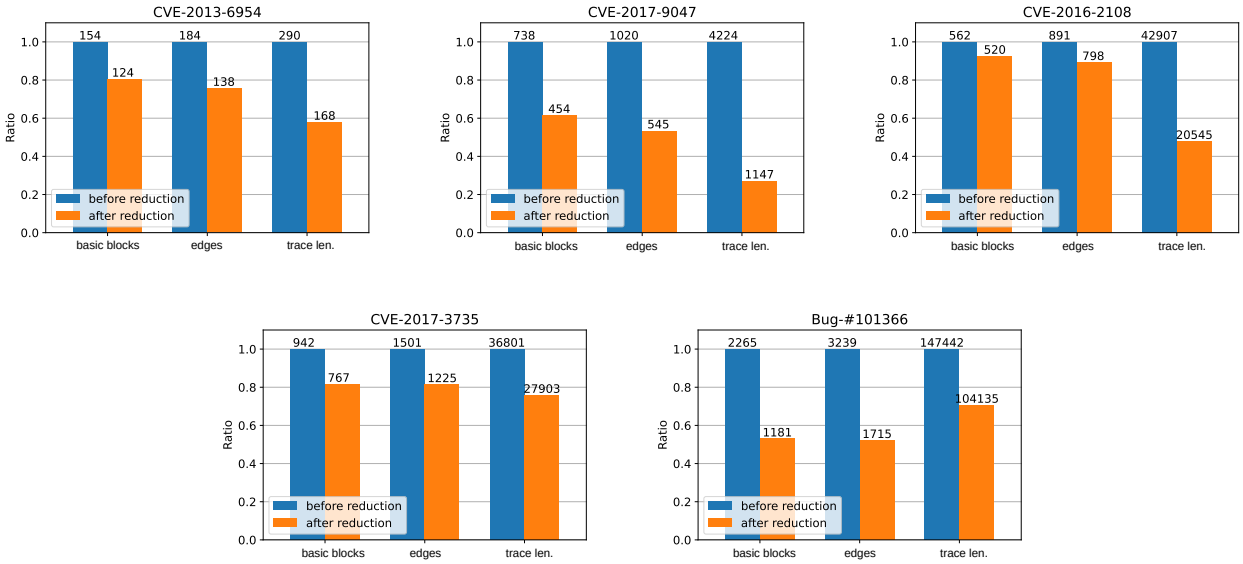


Figure 11: Comparison in three dimensions (basic blocks, edges, and trace length) before and after reduction with `IgorFuzz`.