Semantic Crash Bucketing

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
USA
rvt@cs.cmu.edu

John Kotheimer
Heinz College
Carnegie Mellon University
USA
john.kotheimer@alumni.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
USA
clegoues@cs.cmu.edu

ABSTRACT

Precise crash triage is important for automated dynamic testing tools, like fuzzers. At scale, fuzzers produce millions of crashing inputs. Fuzzers use heuristics, like stack hashes, to cut down on duplicate bug reports. These heuristics are fast, but often imprecise: even after deduplication, hundreds of uniquely reported crashes can still correspond to the same bug. Remaining crashes must be inspected manually, incurring considerable effort. In this paper we present Semantic Crash Bucketing, a generic method for precise crash bucketing using program transformation. Semantic Crash Bucketing maps crashing inputs to unique bugs as a function of changing a program (i.e., a semantic delta). We observe that a real bug fix precisely identifies crashes belonging to the same bug. Our insight is to *approximate* real bug fixes with lightweight program transformation to obtain the same level of precision. Our approach uses (a) patch templates and (b) semantic feedback from the program to automatically generate and apply approximate fixes for general bug classes. Our evaluation shows that approximate fixes are competitive with using true fixes for crash bucketing, and significantly outperforms built-in deduplication techniques for three state of the art fuzzers.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; **Maintaining software**; **Software defect analysis**; • **Security and privacy** → *Software security engineering*;

KEYWORDS

Crash Bucketing, Fuzzing, Bug Triage, Program Transformation, Automated Bug Fixing

ACM Reference Format:
Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238200

1 INTRODUCTION

The advent of large scale fuzzing services, such as Google's OSS-Fuzz [1, 45] and Microsoft's fuzzing service [9], attest to the effectiveness of automatic bug finding tools. When operating at scale, accurately identifying unique bugs is critical for (a) reducing time-consuming manual debugging efforts [14, 41], (b) characterizing the effectiveness of automated bug-finding tools [12, 14, 37, 42, 48], and (c) ranking interesting crashing test cases [14]. However, one outstanding challenge in effectively deploying automated fuzzing techniques is accurately identifying unique bugs during crash triage. Fuzzers often generate thousands of crashing inputs that ultimately correspond to the same bug [14], and the sheer number of crashing inputs preclude manual inspection. This is a hard problem, and an area of active research [17].

Automated crash triage techniques seek to approximately *bucket* multiple crashing (but ultimately equivalent) inputs [14, 17, 37, 41], to reduce the number of redundant bug reports an engineer must inspect by hand. At a high level, automated testing tools like fuzzers and symbolic executors typically use tool-specific, heuristic bucketing strategies. Both research and industry standard triage techniques have known limitations [17, 42]. Techniques may assume "best-effort" hardcoded values (e.g., the number of calls to consider in a call stack [2]) or require tool-specific instrumentation for feedback-driven approaches [3]. The varied sensitivity of such ad hoc techniques result in imprecise bug identification that can fail in two ways. *Overapproximation* occurs when multiple crashing tests caused by a single bug incorrectly bucket to more than one unique bug (i.e., duplicate bug reports). *Underapproximation* occurs when crashing tests due to multiple unique bugs are put in the same bucket [41, 48] (i.e., missed unique bugs).

Stack hash [37, 48] and branch sequence [7] techniques used in state-of-the-practice fuzzers [2, 3, 7] can suffer from both over- and underapproximation [41, 48]. Such techniques seek to determine bug uniqueness as a function of, e.g., crashing input [48], program traces [7, 17], program crash state [3], or a combination of these [14]. Recent (and more sophisticated) research advances propose to more precisely classify unique bugs using symbolic analysis [41], machine learning on crashing inputs [14], and backward taint analysis on program traces [17]. While such approaches promise more accurate bucketing by considering semantic program behavior (e.g., [17, 41]), their accuracy depends on sensitivity to a general semantic trait (e.g., symbolic branch uniqueness) and can still misbucket bugs. Built-in or hardcoded techniques further struggle to integrate specialist knowledge that can produce more accurate output for classes of bugs.[1]

[1] https://twitter.com/azonenberg/status/966738179486134272

We present a radically new approach to identifying unique bugs in the context of fuzzing: by modifying the program itself. Our insight is that bugs can be characterized by a semantic transformation on the program under test. For example, patching one of two buffer overflows in a single execution can distinguish crashes unique only to the second. Further, a fix can stop the same logical bug from manifesting on multiple unique execution paths.

Our insight draws on the fact that *fixing the program* offers a precise way to associate crashing inputs with a unique bug, since correct fixes should neutralize all crash-inducing inputs associated with the bug in question.

We introduce Semantic Crash Bucketing, which maps crashing inputs to bugs as a function of change (delta) in program semantics, where the delta approximates fixing the root cause of the bug. In general, root cause analysis is hard [27, 39], and automatically fixing bugs is an open problem [30]. However, existing work in automated program repair (APR) does demonstrate that programs can profitably be transformed to automatically improve quality [29, 34, 36]. The motivation behind our approach is that changing a program with *approximate* fixes can accurately and automatically constrain crashing behavior in a way that mimics real program fixes to detect unique bugs in fuzzer output.

Semantic Crash Bucketing contrasts with the usual sense of seeking program fixes with respect to a correctness oracle (such as tests [29, 34]). However, although the objective of Semantic Crash Bucketing is different from APR, it can similarly suffer from program transformations that overfit to the success criterion. For example, suppose a program contains more than one unique bug, each with independent fixes. Inserting `exit(0);` at the beginning of a program will satisfy the criterion of neutralizing all crashes, but will associate (and underapproximate) all unique bugs with a single fix. To be effective, program transformations must therefore have constrained semantic effects to precisely identify unique bugs under Semantic Crash Bucketing.

We propose a rule-based approach using fix templates to constrain the semantic transformations for crash bucketing. Our observation is that common bugs typically detected by fuzzers (e.g., buffer and integer overflows, null dereferences, etc.) have semantic properties that are amenable to a rule-based application of general fix templates (as found in analog APR work, e.g., [16, 26, 46]). At a high level, rule-based application of fix templates can integrate specialist knowledge of bug semantics into the triage process to produce more precise output. We demonstrate Semantic Crash Bucketing for buffer overflows and null dereferences on real-world bugs in the CVE database [4]. Buffer overflows and null dereference vulnerabilities account for some of the most common software security weaknesses [31] and are frequently discovered through fuzzing [7, 42, 45]. Our contributions are as follows:

- **Semantic Crash Bucketing**, a novel technique to automatically identify unique bugs as a function of changing a program's semantics. Semantic Crash Bucketing groups crashing inputs by applying program transformations to the program under test. We use Semantic Crash Bucketing to identify imprecise crash reporting in fuzzers, and to compare the effectiveness of developer-written fixes and approximate fixes.
- **Approximate fixes.** We present an automated procedure using bug-fixing patch templates and rule-based application of patches to approximate correct fixes. In general, correctly and automatically fixing a program is hard. The key insight is that the effectiveness of approximate fixes is competitive with using correct fixes for identifying unique bugs. We instantiate Semantic Crash Bucketing with approximate fixes for real-world bugs commonly found by fuzzers: buffer overflows and null dereferences and demonstrate effectiveness.
- **Empirical evaluation**. We comparatively evaluate Semantic Crash Bucketing using developer-written fixes and approximate fixes with deduplication techniques of three state-of-the-art fuzzers (AFL-Fuzz [7], CERT BFF [2], and Honggfuzz [3]). We show with Semantic Crash Bucketing that approximate fixes associate crashing inputs precisely (i.e., no under- or overapproximation) for 19 out of 21 bugs in 6 projects compared to ground truth fixes. We also show that bucketing with approximate fixes is more precise than built-in deduplication of all three fuzzers. Our results are available online.[2]

2 MOTIVATING EXAMPLE

AFL-Fuzz is known to find null dereference and memory corruption bugs in even well-tested software [5]. Consider one such bug found in `SQLite`: a null dereference that was later fixed by the patch in Listing 1a. The `sqlite3WalkSelect` function (Line 7) walks the expression tree of a SQL select statement. The return value of `sqlite3WalkSelect` can indicate an error in a `SELECT ... FROM ...` statement, but the return value is not checked. This missing check can lead to a null dereference downstream during execution due to an invalid `FROM` clause. The fixing commit message says:

```
Make sure errors from the FROM clause of a SELECT cause
analysis to abort and unwind the stack before those errors
have a chance to mischief in the "*" column-name wildcard
expander.
```

The developer thus checks the return value of `sqlite3WalkSelect` and aborts, avoiding any null dereferences downstream (Line 8, Figure 1a).

Current fuzzers and symbolic executors can find many different crashing inputs that trigger bugs like these. For example, slight modifications in a crash-inducing `SELECT...FROM...` input could follow a different sequence of calls or branches, but still trigger the same bug. Existing techniques use generic heuristics to identify unique crashes from a set of many generated inputs. Call stack hashes [2, 3, 12, 20, 37] are predominant; instrumentation-based fuzzers may use program execution paths sensitive to branch sequences [3]. These heuristics are fast and moderately effective, but remain imprecise, because they are sensitive to inputs that vary program execution in a way that is unrelated to the actual bug. Depending on the heuristic and inputs, fuzzers report many duplicate crashes as unique.

Our approach defines bug uniqueness in terms of *program transformation*. The motivation is that fixing a bug (as the developer did in Figure 1a) ideally "catches" all crashing inputs related to the bug,

[2] https://github.com/squaresLab/SemanticCrashBucketing

```
--- a/src/select.c
+++ b/src/select.c
@@ -4153,7 +4153,7 @@ static int selectExpander(Walker
     *pWalker, Select *p){
      /* A sub-query in the FROM clause of a SELECT */
      assert( pSel!=0 );
      assert( pFrom->pTab==0 );
-     sqlite3WalkSelect(pWalker, pSel);
+     if( sqlite3WalkSelect(pWalker, pSel) ) return
     WRC_Abort;
      pFrom->pTab = pTab = sqlite3DbMallocZero(db,
           sizeof(Table));
      if( pTab==0 ) return WRC_Abort;
```

(a) `SQLite`: a developer fix that avoids a null dereference.

```
--- a/src/resolve.c
+++ b/src/resolve.c
@@ -164,6 +164,9 @@ int sqlite3MatchSpanName(const char *
     zSpan, const char *zCol, const char *zTab, const char
      *zDb){
   int n;

+  if(zSpan == NULL) {
+    exit(101);
+  }
   for(n=0; ALWAYS(zSpan[n]) && zSpan[n]!='.'; n++){}
   if( zDb && (sqlite3StrNICmp(zSpan, zDb, n)!=0 || zDb[n
        ]!=0) ){
     return 0;
```

(b) `SQLite`: autogenerated approximate fix for the null dereference.

Figure 1: Two fixes for a null dereference in `SQLite` 3.8.9. The actual fix is shown on the left (commit `10c478e`). Our approach automatically generates the patch on the right.

irrespective of call stacks or other program execution paths.[3] The challenge is that finding true fixes is hard. Automated root cause analysis is difficult and expensive [27, 33], especially for bugs like this one, that requires deep reasoning.

Our primary insight is that simpler *approximate* fixes can substitute for real fixes to precisely bucket crashing inputs. For example, Figure 1b presents an autogenerated approximate fix for the same `SQLite` null dereference bug. Semantically, it safely aborts the program if `zSpan` are `null`. It turns out that the `SQLite` bug leads directly to `zSpan` being `null` at this later program point (i.e., when the input statement contains a `*` expander described in the commit message). The approximate patch precisely "catches" similar crashing inputs like the actual patch.

Our approach uses syntactic templates and configurable "semantic cues" to generate such patches. Semantic cues act as predicates for applying patch templates. A concrete example is "Check whether any dereferenced variables at program point p is `null`. If so, return the variable name". A patch template can then be instantiated with the specific variable. In general, templates and rules for patch generation and application are specified just once per bug class (e.g., null dereferences and overflows.). We describe the procedure fully in Section 4, but provide a brief summary here for null dereferences. The patch template for null dereferences checks whether a variable is `null`, and safely aborts the program if so. This template contains a "hole" for the variable to check, and must be instantiated with a concrete variable. We configure the procedure to check for a semantic cue: whether variables are null at the point of crash using a debugger environment. In this case, our procedure finds that `zSpan[n]` could be a problematic dereference, and dynamically checks whether `zSpan` is null when the program crashes. Variable `zSpan` is found to indeed be `null`, generating the patch in Figure 1b. The patch is validated to confirm that the modified program no longer crashes for the input. That is, the autogenerated patch approximates the real fix effectively because it discovers and fixes the related null dereference triggered downstream during execution even though it does not deeply address the root cause.

[3] And, under correctness assumption of the fix, any other crashing input is associated with a different unique bug.

In essence, applying lightweight program transformation reduces noise compared to typical deduplication heuristics by focusing on the semantic properties of the bug. At the expense of slight up front cost per bug class, our approach provides a configurable mechanism that is sensitive to the semantic property of the bug to more precisely identify uniqueness.

A configurable approach is important: bugs exhibit different semantic traits to which program transformation must be sensitive. For example, null dereferences cause an immediate program crash which allows us to identify possible causes at the point of crash. On the other hand, buffer overflows typically only cause a crash once corrupted memory is accessed, and not when the overwrite actually occurs. Handling overflows therefore requires a different strategy (see Section 4).

Fuzzers can also underreport unique bugs. For example, under a naïve call stack approach, two unique null dereferences in a single function will be reported as just one unique bug. Our technique can identify each bug uniquely via independent program transformations.

3 SEMANTIC CRASH BUCKETING

This section introduces Semantic Crash Bucketing (SCB). Semantic Crash Bucketing is a general method for bucketing crashes *in terms of program transformation* (i.e., a semantic delta). Semantic Crash Bucketing can be performed with arbitrary program transformations. Our goal in this section is to develop a way for determining how well approximate fixes identify unique bugs compared to (a) ground truth fixes and (b) existing methods in fuzzers. We now introduce the problem definition and application of SCB for detecting inaccurate error reports.

3.1 Problem Formulation

A *bug* in our context is a software flaw that leads to an error (i.e., undesirable program behavior); an error is a deviation from expected behavior defined by a test oracle. We address on the types of bugs typically found by fuzzers, namely those that induce runtime crashes. For such bugs, the error oracle is signaled by a runtime failure: a crash results in `SEGFAULT`.

Semantic Crash Bucketing groups crashing inputs according to a program change that nullifies those inputs (i.e., cause the inputs to no longer crash the program). Thus, a true fix for a unique bug maps all crashing inputs for that bug to a unique bucket. Grouping crashes as a function of known fixing patches is a de facto method for establishing ground truth classification of fuzzer crash reports [14]. We use this idea to develop a general method of identifying misbucketing (e.g., duplicate crash reports) arising from approximate fixes and fuzzers.

Ideal Bucketing. We begin by defining an Ideal Bucketing, where the correct fixing patches for unique bugs in a program are known or presumed. This definition represents ground truth to measure the effectiveness of our approach (Section 5). The intuition is straightforward: some known or presumed program transformation T_i fixes all crashing input associated with a bug i, and only those inputs. T_i is by construction the theoretically ideal oracle transformation that correctly fixes the bug i and thus all of the crashing behavior it can cause. In practice, we may think of such a transformation as a correct developer-written patch for a single bug.

We express Ideal Bucketing in terms of unique bugs, the crashes they induce, and their fixes. Let $i \in n$ be the identifier for a unique bug i of n unique bugs in a program P. A unique bug i is associated with a set of one or more crashing inputs, which we denote by a bucket b_i. Let $T_i : P \rightarrow P$ be a function that applies a correct fix to the program P, for unique bug i. A correct fix T_i fixes all crash-inducing inputs b_i due to i, but none of the crashes due to a different bug j with crashing inputs b_j.

We express all buckets containing crashing inputs uniquely fixed by known T_i, $i \in n$ as disjoint partitions $B = b_1 \uplus \cdots \uplus b_n$ under the correctness assumption of T_i. For a particular T_i, ideal bucketing implies:

$$\begin{aligned}
&\forall\, b_i \in B,\\
&\forall\, b_j \in B \setminus b_i \ \texttt{s.t.}\\
&\quad \forall\, c_i \in b_i, \langle T_i(P),\ c_i\rangle \not\rightsquigarrow \mathit{crash}\\
&\quad \forall\, c_j \in b_j, \langle T_i(P),\ c_j\rangle \rightsquigarrow \mathit{crash}
\end{aligned}$$

Where $\langle T_i(P), c\rangle \not\rightsquigarrow \mathit{crash}$ expresses that the program P under transformation of fix T_i and executed on crashing input c does not induce a runtime crash. Ideal bucketing for a bug i expresses that the fix T_i associates non-crashing behavior with all previously crashing inputs $c \in b_i$, but not any crashes for other buckets $b_j \in B \setminus b_i$.[4]

One subtlety of Ideal Bucketing is the special case where a single input may trigger multiple bugs. For example, two separate buffer overflow copies (i.e., two bugs b_1 and b_2) along the same execution path may overwrite the stack (twice) in a single execution. From our definition, neither corresponding fix T_1 nor T_2 will bucket the crashing input. However, we can extend the definition to account for *composition* of transformations T_1 and T_2 to place such a crashing input into a separate bucket that represents a composite fault. Although conceptually useful, we focus on logically discrete fixes (based on developer patches) to associate crashing inputs with bugs so that it is tenable to experimentally compare real and approximate fixes. In practice, fuzzer-generated inputs typically trigger single bugs, and our results corroborate this observation. Classifying composite faults is an open problem [22] and we leave the consideration of using program transformation for classifying such faults to future work.

[4]Note: if $B \setminus b_i = \varnothing$ then the constraint on b_j holds vacuously.

3.2 Detecting Duplicates

One goal in fuzz triaging is to approximate the ground truth ideal bucketing strategy, minimizing overhead and confusion for the engineer using a fuzzer to identify defects. Approximations are done by, e.g., unique call stack hashes or unique branch sequences. Such approximations can fail, however, leading to *misbucketing* of crashing inputs. Misbucketing can be classified into two categories [41]:

(1) duplicate bug reporting and
(2) suppressed unique bugs: unreported unique bugs that are missed by crash bucketing (or "over-condensing").

In this paper, we deal with the first case of duplicate bug reports. We now describe how we detect duplicate bug reports in terms of fixing transformations T_i, where Ideal Bucketing does not hold. Consider two example bug reports produced by a fuzzer: bug 1 with a crash bucket $b_1 = \{c_1\}$ containing crashing input c_1, and bug 2 with $b_2 = \{c_2\}$. We say that b_2 is a *duplicate bug report* if c_2 actually crashes the program due to bug 1. That is, the correct bucketing implies $b_1 = \{c_1, c_2\}$ and no bug 2 should be reported. Duplicate misbucketing wrongly implies bug uniqueness, increasing the triage burden of engineers processing fuzzer output.

In an *imprecise* bucketing B, duplication occurs when the following is true for a particular T_i:

$$\begin{aligned}
&\exists\, b_i \in B,\\
&\exists\, b_j \in B \setminus b_i \ \texttt{s.t.}\\
&\quad \forall\, c \in b_i, \langle T_i(P),\ c\rangle \not\rightsquigarrow \mathit{crash}\\
&\quad \exists\, c_{\text{dup}} \in b_j, \langle T_i(P),\ c_{\text{dup}}\rangle \not\rightsquigarrow \mathit{crash}
\end{aligned}$$

That is, some crash $c_{\text{dup}} \in b_j$ *actually fixed* by T_i is considered a crash for a different unique bug j, belonging to b_j. By our correctness assumption of T_i, any crash fixed by T_i must belong to b_i for ideal bucketing to hold. Note that if c is the only crash in b_j then a unique bucket is implied, resulting in a duplicate report of bug i as some other bug j that should not exist.

In summary, given correct T_i's, we can determine ground truth Ideal Bucketing and detect duplicate bug reports as deviations from Ideal Bucketing.

3.3 Semantic Crash Bucketing Procedure

Our formulation leads to a straightforward procedure for identifying misbucketing in fuzzers. Figure 2 illustrates the process. A `Fuzzer` takes a program P and input to generate a set of crashes $C = c_1, \ldots, c_n$ ❶. The fuzzer reports a set of crashing input according to its built-in method for identifying unique bugs. We represent the fuzzer output as a disjoint set of unique bugs indexed by I: $B_{fuzzer} = \biguplus_{i \in I} b_i$ ❷.

As a matter of practicality, a fuzzer does not, by default, preserve all generated crashing inputs. Instead, a fuzzer discards any crashing input it believes triggers a bug it has already seen, and typically outputs one representative crash for each bug/bucket it considers unique. This is expressed as $B_{fuzzer} = (b_1 = \{c_1\}) \uplus (b_2 = \{c_2\}) \uplus \cdots \uplus (b_n = \{c_n\})$.

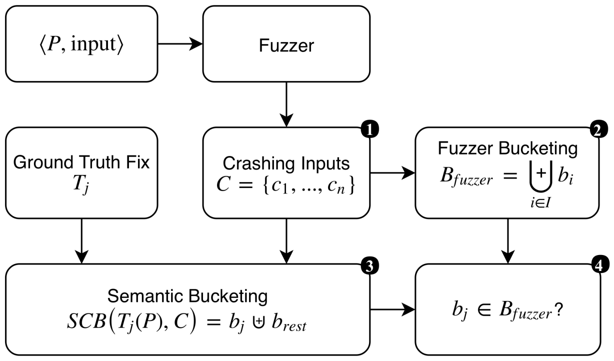
⟨P, input⟩
Fuzzer
Ground Truth Fix T_j
Crashing Inputs $C = \{c_1, ..., c_n\}$
Fuzzer Bucketing $B_{fuzzer} = \biguplus_{i \in I} b_i$
Semantic Bucketing $SCB(T_j(P), C) = b_j \uplus b_{rest}$
$b_j \in B_{fuzzer}$?
1
2
3
4

Figure 2: The Semantic Crash Bucketing Procedure.

The function *SCB* takes as input the set of crashes C and a ground truth fix T_j ❸. For a single fix T_j, *SCB* partitions the set of crashes C into a disjoint set $b_j \uplus b_{rest}$ by running each crash $c \in C$ on the modified program $T_j(P)$. The set b_j contains all inputs fixed by T_j, as in the Ideal Bucketing case, while b_{rest} contains all inputs that still cause $T_j(P)$ to crash. The final step ❹ tests if the crashes in a nonempty bucket b_j distinguished by T_j is contained in B_{fuzzer}. Because B_{fuzzer} contains a partition of unique bugs with just one representative crash in any b_i, there are only two outcomes for the test $b_j \in B_{fuzzer}$: (1) $|b_j| = 1$ and is equal to some b_i in B_{fuzzer}, implying that B_{fuzzer} precisely buckets the crashing input for a bug j, or (2) $|b_j| > 1$ implying that the crashes in $c \in b_j$ are partitioned across multiple buckets in B_{fuzzer}, implying that the fuzzer reported duplicate bugs. For simplicity, Figure 2 illustrates the procedure for a single fix T_j that fixes at least one crashing input in C. Ideal Bucketing checks that every crashing input in C can be fixed (and bucketed) uniquely by one or more fixes T.

4 GENERATING APPROXIMATE FIXES

This section explains how we instantiate our approach to perform SCB using *approximate fixes*. In practice, a developer fix provides the best assurance of correctly fixing a known bug, which we accept as ground truth T for SCB. However, our goal is to reduce the burden on developers to triage fuzzing output when the crash's fix is not immediately known. In general, fixing arbitrary bugs automatically is hard [30]. Our core insight is that an approximate fix $\widehat{T}$ is competitive with using T to identify unique bugs under SCB. In our approach, $\widehat{T}$ is an *automatic* production encoding the semantic properties necessary to fix possible crash-inducing bugs. To demonstrate, we instantiate SCB with approximate fixes on null dereferences and buffer overflows in C programs.

4.1 $\widehat{T}$ Production.

At a high level, $\widehat{T}$ is a production of a function $G(P, \mathcal{T})$ that takes two inputs: the source program P and a crash trace $\mathcal{T}$. A crash trace is produced by executing P on a single crashing input c. G generates patches from fixing templates, and applies them to the source. Patch application is predicated on certain information in the program source, dynamic trace, or both. We refer to these predicates as *semantic cues* that are sensitive to semantic properties of a bug class. If the predicates are not satisfied, the program is not modified.

We concretely represent $\widehat{T}$ as a source-level patch. This has two advantages. First, patches can be used as better bug reports [47], supporting human triage and debugging. Second, patches can apply without actually running the program, meaning static analyses (e.g., static symbolic execution) can also benefit from SCB.

We use `GDB` and `ltrace` to obtain dynamic crash traces. In principle, any dynamic technique or analysis can enrich the space of semantic cues to trigger program modification. We now describe in concrete terms how we obtain $\widehat{T}$ for null dereferences and buffer overflows.

4.2 Null Dereferences

Null dereferences are typically fixed in one of two ways: correctly initializing a variable or checking whether a variable is null before dereferencing it [46, 49]. At a semantic level, a fix must enforce a nonnull property for a variable that results in a null dereference crash. We use the template in Figure 3 to approximate fixing a null dereference. `%%%PVAR%%%` is a "hole" substituted with the offending program variable.

```
if (%%%PVAR%%% == null) {
  exit(101);
}
```

Figure 3: A template for null dereferences

The patch approximates error handling by exiting the program on condition of `PVAR` being `null` (similar to the common C idiom of `return -1;`). While simply exiting appears simplistic, it is in fact appropriate for our objective to accurately bucket crashing input. Consider if we chose a different strategy by returning a value or initializing `%%%PVAR%%%`. Besides the difficulty of correctly inferring appropriate values, we risk the possibility that the modified program may continue executing and cascade errors or crash in other unexpected ways. Without complete information of the root cause to actually fix the bug, exiting is a conservative strategy: it acts as an assertion ensuring the desired nonnull semantic property. The correct fix in our motivating example supports this strategy: `SQLite` conservatively aborts for error cases (but does some extra work propagating the error up the call stack). Since the template can be changed, our method does not preclude other possibilities; however, our experiments show that the template in Figure 3 approximates true fixes well enough for precise crash bucketing.

Template definition is only part of the larger problem: generating the final patch $\widehat{T}$ also relies on identifying the appropriate program variable and location to insert the patch. Semantic cues from a `GDB` trace inform patch application: whether a variable dereference at the point of crash is null. For example, our approximate patch in Figure 1b checks the variable `zSpan`. The general procedure for finding such crash-inducing variables works as follows:

(1) Attach `GDB` to the program, run it on the crashing input.
(2) Extract the source line and code reported at the crash.
(3) Parse the code for pointer dereference syntax (e.g., `p->q`).
(4) Working backwards, extract program variables that are dereferenced (e.g., extract `p` from `p->q`). Test, using `GDB`, whether the variable is `null` in the debugger environment.

(5) If the variable is null, return the variable and associated line number. If not, move backwards a basic block and continue from (3).

If the procedure succeeds, we substitute the template program variable and insert the candidate patch just before the null dereference. The null check could possibly be placed earlier, and a true fix may indirectly prohibit a particular variable from being null (cf. the correct `SQLite` fix in Figure 1a). Our decision is an inexpensive compromise that we show works well in practice.

Before we use the patch for SCB, we first validate that the modified program no longer crashes for input c. The patch generation procedure can produce more than one candidate patch, but our implementation takes the first crash-fixing patch for bucketing.

4.3 Buffer Overflows

Buffer overflows are a class of memory corruption bugs commonly discovered by fuzzers [7, 45]. Buffer overflows are typically fixed by performing array bounds checking on memory accesses. Our approximation to fixing buffer overflows thus focuses on array length as the underlying semantic property to change. Inferring array bounds can directly assist suggesting approximate fixes for arbitrary overflow bugs, but generally requires additional analysis techniques and remains an open problem [19, 21, 24]. Our approach is to truncate memory writes that may cause invalid accesses. Applications in failure-oblivious computing [44] and exploit mitigation [33] use a similar mechanism.

Unsafe C library functions commonly trigger buffer overflows [19, 28, 33, 38] and persist in modern software.[5] Our approach applies templates for common C library functions, such as `memcpy`, `strcpy`, `sprintf`, `gets`, `strcat`, etc.

We give an example template for `memcpy` in Figure 4; the templates for other overflows are conceptually similar. We rewrite existing calls and restrict the length of data copied to a default concrete value of 1. Restricting data to only one byte approximates a conservative *angelic value* [13] that is likely to lead to non-crashing program termination. Note that other possibilities exist: we may, for example, instrument the code to obtain actual angelic values observed at runtime and use these to construct fixes. Our experiments show that our current choice works well for precise bucketing.

```
// Modify a possible overflowing memcpy call
size_t angelic_length = 1;
memcpy(%%%DST%%%,%%%SRC%%%,angelic_length);
```

Figure 4: A template for `memcpy`. `%%%DST%%%` binds to the destination argument for the original `memcpy` call, and `%%%SRC%%%` is binds to the source argument.

Compared to approximating null fixes, overflow fixes do not attempt to stop execution: placing a condition on the length of a potential buffer proves problematic if we do not know its bounds. Conversely, simply exiting before calling an unsafe function will overfit to unique crashing inputs that would crash after the function. In addition, while memory corruption occurs *during* execution of the C library functions, the program only *crashes at a later point*: once an invalid memory access occurs in the heap, or when a corrupted return address is accessed on the stack.[6] These behaviors motivate different semantic cues compared to null dereferences, and emphasize the importance of a configurable approach. For buffer overflows, we implement a procedure to discover possibly problematic library calls and resolve their location. A patch template like Figure 4 then replaces the call. The steps are as follows:

(1) Use `ltrace` to obtain a trace of library calls from the crashing program run.
(2) Working backwards, resolve the source location of library calls in the trace for which we have fixing templates.
(3) Apply the template at the location and rerun the program on the original crashing input.
(4) If the program no longer crashes, emit the approximate fixing patch $\widehat{T}$. Else continue from step (2).

Similar to null dereferences, we validate that the program no longer crashes for any change done in step 3, and use the first crash-fixing patch for bucketing.

Extending Semantic Crash Bucketing. The patch templates and rules for patching are embedded in Python scripts and are easy to change. Users can define their own patch templates and semantic cues for patch application depending on the semantic properties of the bug types or application-specific APIs. The `GDB` interface and `ltrace` output is available in the scripting framework for customization. Additional analysis tools can be integrated (e.g., `valgrind`), though naturally this requires some extra effort.

5 EXPERIMENTAL DESIGN

Ultimately, we want to know how well approximate fixes $\widehat{T}$ distinguish unique crashes compared to (a) ground truth bucketing by T (developer fixes) and (b) built-in fuzzer deduplication (the previous state of the art). We conduct a controlled experiment with real bugs for which we know the ground truth fix (Section 5.1). Unfortunately, for the purposes of our experiments, state of the art fuzzers do not all neatly decouple fuzzing campaigns from crash deduplication (e.g., deduplication is invoked during fuzzing iterations). Instead, we first generate, for each bug, an upper bound of inputs that trigger the same bug (i.e., a "crash corpus") which aim to exercise different execution paths triggering the same bug (Section 5.2). We then provide this crash corpus as input to each fuzzer, and run a campaign for a fixed length (2 hours), forcing the fuzzer to perform deduplication on the crash corpus during fuzzing iterations (Section 5.3). We use the developer fix and apply SCB to obtain the ground truth number of duplicate bug reports *after* the campaign (which includes each fuzzer's deduplication effort on the corpus). We then apply SCB with *approximate fixes* and measure (a) the difference from ground truth, and (b) deduplication improvement over existing fuzzers.

Hardware. We ran our experiments on an Ubuntu 16.04 LTS server with 2 Xeon E5-2699 CPUs and 20GB of RAM. Crash Corpus generation and fuzzing campaigns all ran on a single CPU core. We used four cores to recompile when validating whether an approximate fix stops a crash.

[5] A `strcpy` vulnerability has been found in the Linux distribution as recent as 2017 [8].

[6] Memory fence-posts can detect overwrites immediately, and don't require a program to `SEGFAULT`. This requires code instrumentation and extra shadow memory that hurts fuzzing performance. Approximate fixes can be adapted accordingly, but we currently do not assume such instrumentation.

5.1 Bugs with Ground Truth

We evaluate on a sample of 18 null dereference and 3 buffer overflow bugs in 6 real-world projects. For each bug we (a) extracted a ground truth developer fix from the project and (b) sourced a crashing input that triggers the bug (e.g., from online bug reports).

Projects with multiple bugs. `SQLite` is well-tested, popular database software; `w3m` is a text-based web browser. For these projects, we curated datasets of multiple bugs in a single revision. To be useful, a deduplication strategy should correctly bucket crashing inputs associated with a bug, but *only* that bug (and not those for other bugs). That is, $\widehat{T}$ should be as close to Ideal Bucketing as possible, giving strong assurance that $\widehat{T}$ does not overfit the input crashes.

Thus, we curated a dataset of fixes for 12 null dereference bugs in a *single* `SQLite` revision. This is an onerous task because developer fixes are often interspersed over long periods of time[7] and fixing patches cannot always be automatically applied to previous revisions due to intermediate code changes. In addition, a single patch may contain multiple fixes, which we must separate for each respective bug. We therefore manually minimized and backported patches to support a large, controlled ground truth study on multiple bugs in `SQLite`.

We selected `w3m` out of a list of projects with reported CVEs [6] and found that it also has multiple null dereferences for which we could find developer fixes and crashing inputs that work on the same revision.[8] We demonstrate SCB on four (4) null dereference bugs on a single revision of `w3m`.

Other projects. We identify two null dereference bugs in different versions of `PHP`, a large, popular project with well-documented bugs and ground truth patches. We demonstrate SCB on real-world overflow bugs in `R`, a large and popular software suite for statistical computing; `Conntrackd`, a networking utility; and `libmad`, an MPEG audio decoder. We apply SCB to a `strcpy` vulnerability in `R`. To demonstrate real-world utility, we demonstrate SCB on two of our own 0-days found in previous fuzzing campaigns: a `strcpy` vulnerability in `Conntrackd` and a `memcpy` vulnerability in `libmad`.[9]

5.2 Crash Corpus Generation

For each bug, we generate a large baseline corpus of crashing inputs from the initial crashing input, aiming to exercise different execution paths triggering the same bug. We use this corpus to test how well each fuzzer's deduplication method copes with varying behavior that trigger the same bug. Although a typical fuzzing campaign begins with one or more non-crashing seed files as input, it is hard to trigger a specific bug starting with arbitrary seed files: the input search space is huge, and fuzzing nondeterminism means it is difficult target specific areas of code. Isolating features in test cases is one strategy for producing crashing test cases that may correspond to the same bug [11, 23], but can take several days to produce a large test set. Instead, we pursue a conceptually similar approach, mutating an initially crashing input to explore different execution paths that trigger a particular bug. We then use this corpus as input to the other state-of-the-art fuzzers.

To do this, we use the existing "Crash Mode" procedure implemented in AFL-Fuzz [7]. The crash exploration procedure tracks branches executed by the input, and mutates input to try and force execution along different branches, where the objective function is to preserve crashing behavior. Inputs that fail to explore interesting paths or crash the program are discarded. We run crash exploration for two hours *per bug*, producing crash corpora of related inputs for each bug's crashing seed file.

5.3 Evaluating Fuzzers

We compare to three state of the art fuzzers: AFL-Fuzz [7], CERT BFF [2], and Honggfuzz [3]. These fuzzers are frequently used in industrial and research settings [10, 45, 48] and implement different deduplication techniques. In general, fuzzers do not decouple fuzzing campaigns with crash deduplication; crashes are deduplicated during fuzzer iterations. To trigger crash deduplication, we seed fuzzing campaigns for each fuzzer with the crash corpus.

Industrial-strength fuzzers are highly configurable. We sought to evaluate on default options across varying parameters in coverage-based fuzzing, call stack depth, branch sequences, and point-of-failure information. We evaluate on five configurations across the three fuzzers:

AFL-Fuzz. We use AFL-Fuzz in its default configuration. AFL is instrumentation-driven, and keeps track of branches taken during fuzzing. Roughly, this means that AFL is sensitive to uniquely executed paths. AFL's default method for fuzzing uses the same mechanism as "Crash Mode", starting from a non-crashing seed and with an objective of discovering arbitrary crashes. One key difference, however, is that "Crash Mode" does not deduplicate the crash corpus by default. Therefore, to approximate AFL's deduplication in a real campaign (while avoiding a redundant fuzzing campaign), we use AFL's own minimization procedure directly on the crash corpus, then remove equivalent duplicates.

CERT BFF. We run CERT BFF in its default configuration, which uses a call stack hash based on, by default, the *five* last calls (frames) leading to a crash. This number is configurable. Thus, for the second configuration, we set BFF to use a call stack of just *one* frame to determine bug uniqueness. BFF also invokes a built-in input minimizer while fuzzing on-the-fly.

Honggfuzz. We run Honggfuzz in its default configuration, which uses a call stack hash of seven calls. By default, Honggfuzz considers information at the point of failure when a crash occurs (e.g., the last known PC instruction and faulting address) to report uniqueness. Honggfuzz can enable a feedback-driven fuzzing mode, provided a program is compiled with the coverage instrumentation. In the first configuration, we disable coverage; in the second, we enable coverage.

Note that due to input mutation during the campaign, a fuzzer may trigger a bug that we do not have a fix for. As a final post processing step, we use the ground truth fix T to filter out only the crashes fixed by T.

6 EXPERIMENTAL RESULTS

Our main result is that SCB with approximate fixes is *just as precise* as using the ground truth fix for 19 out of 21 bugs across all configurations. Approximate fixes suffer only small imprecision,

[7]The `SQLite` bugs were fixed over a period of four months.
[8]https://github.com/tats/w3m/issues?q=Null+pointer+is:closed
[9]Vulnerability disclosure is in progress with CERT under VRF#18-07-YMMKT and VRF#18-07-XKJZJ .

Table 1: Semantic Crash Bucketing results. For each fuzzer configuration, we show the Ground Truth number of duplicate crashes reported by the fuzzer (GT) compared to the number of duplicate crashes reported using approximate fixes with SCB (SCB+$\widehat{T}$). Crash Corpus is the number of crashing inputs that initially seed fuzzing campaigns for each configuration. For example, Bug #1 (first row) in the HFuzz, GT column shows that HFuzz reports 10 duplicates (determined by the ground truth fix), while the approximate fix (SCB+$\widehat{T}$) reports 0 duplicates. When SCB+$\widehat{T}$ reports 0 duplicates, it is as precise as ground truth.

			Crash	**AFL**		**BFF-5**		**BFF-1**		**HFuzz**		**HFuzz-Cov**	
Project	**Type**	**ID**	**Corpus**	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$
		1	191	25	0	2	1	1	1	10	0	9	1
		2	482	85	0	2	0	1	0	4	0	2	0
		3	153	38	0	6	0	0	0	16	0	14	0
		4	326	48	0	0	0	0	0	1	0	0	0
		5	139	34	0	0	0	0	0	0	0	0	0
SQLite	Null-deref	6	66	21	0	0	0	0	0	0	0	0	0
		7	97	20	0	0	0	0	0	0	0	0	0
		8	235	82	0	1	0	0	0	3	0	3	0
		9	389	29	0	1	0	0	0	1	0	1	0
		10	270	65	0	0	0	0	0	1	0	1	0
		11	167	45	1	0	0	0	0	4	2	1	1
		12	108	36	0	0	0	0	0	0	0	0	0
		Subtotal	**2,623**	**528**	**1**	**12**	**1**	**2**	**1**	**40**	**2**	**31**	**2**
		13	458	103	0	25	0	1	0	75	0	77	0
w3m	Null-deref	14	545	23	0	0	0	0	0	0	0	0	0
		15	507	36	0	0	0	1	0	6	0	4	0
		16	525	11	0	0	0	1	0	0	0	0	0
		Subtotal	**2,035**	**173**	**0**	**25**	**0**	**3**	**0**	**81**	**0**	**81**	**0**
PHP	Null-deref	17	81	8	0	0	0	0	0	0	0	0	0
		18	272	32	0	0	0	0	0	0	0	0	0
R	Overflow	19	7	5	0	3	0	0	0	145	0	198	0
Conntrackd	Overflow	20	25	0	0	0	0	0	0	770	0	427	0
libmad	Overflow	21	138	8	0	1	0	0	0	1	0	0	0
		Total	**5,181**	**754**	**1**	**41**	**1**	**5**	**1**	**1,037**	**2**	**737**	**2**

and perform significantly better deduplication compared to state-of-the-art fuzzer deduplication in our experiments.

Speed and Project Size. Automatic patch generation for approximate fixes is fast. Generating a patch from crashing input *and* validating that it fixes the crash (including project recompilation) takes just 18 seconds on average across all bugs. The minimum time for patch generation and validation is 2 seconds, the maximum 49 seconds. Our sample uses large real-world projects, ranging from 12 KLOC to 1 MLOC.

6.1 Overall Results

Table 1 shows results. Each row corresponds to a unique bug, with assigned "ID". "Crash Corpus" is the number of crashing inputs that initially seed the fuzzing campaigns. We deduplicate crashing inputs for each bug using five fuzz campaign configurations (Section 5): **AFL**, **BFF-5** and **HFuzz** are default configurations for the three fuzzers. **BFF-1** configures BFF to use just one call in its call stack hash; **HFuzz-Cov** turns on coverage instrumentation for feedback fuzzing in HonggFuzz. "GT" is the Ground Truth number of duplicate reports for each respective configuration, which we obtain using the *actual* developer fix T for each bug. Column "SCB+$\widehat{T}$" is the number of duplicate bugs for a campaign reported using approximate fixes with SCB.

Except for Bugs 1 and 11 in `SQLite` (discussed subsequently), approximate fixes are *as precise* as the ground truth fix across all configurations. That is, approximate fixes detect and remove all duplicates across all fuzzing configurations for 19 out of 21 bugs. For projects `SQLite` and `w3m` containing multiple bugs, none of our approximate patches suppress any other unique bug. In aggregate, SCB with approximate fixes significantly reduces the number of duplicate crash reports compared to the default configurations: from 754 and 1,037 to just two duplicates for **AFL** and **HFuzz**, respectively, and a reduction of 41 duplicates to one duplicate for **BFF-5**. In practice, crash reports produced by fuzzers must be further triaged manually. Our results show that applying approximate fixes can automatically cut down on the time that an engineer spends on further triage.

Ground truth fixes expose different "semantic sensitivities" of error reporting across configurations and bug types. AFL-Fuzz on average reports more duplicate bugs; this is expected due to its sensitivity to unique execution paths, especially for null dereferences. On the other hand, AFL and BFF report moderate numbers of duplicates for overflow bugs, whereas Honggfuzz reports hundreds of crashes for two stack-based overflows (Bugs 19 and 20). Honggfuzz's default sometimes considers portions of overflowing stack data to signal unique bugs. Bug 21 is a heap-based overflow, and does not adversely affect Honggfuzz compared to stack-based bugs. **BFF-1** uses just one call to calculate a unique stack hash per bug, and reports the least amount of duplicate bugs. Although **BFF-1** appears to perform well, the configuration is nonstandard in practice because it has the caveat that unique bugs triggered in the same function are easily missed. None of the bugs in our sample exposes this weakness in the **BFF-1** configuration, but it is uncommon in real fuzzing campaigns. We include it as one extreme example where coarse, aggressive deduplication can be performed at the cost of potentially missing unique bugs.

Note that even for cases where a fuzzing configuration reports no duplicates for a particular bug, approximate fixes do *as well* as fuzzer deduplication, and *strictly better* for the majority of cases where duplicates are reported. This emphasizes an important point: approximate fixes uniformly bucket crashes via configurable sensitivity to bug-class semantics. Our results show that lightweight program transformation can effectively avoid imprecision due to varying (yet broadly applied) choices made by built-in fuzzer deduplication methods.

6.2 Project-Specific Results

SQLite. Approximate fixes for `SQLite` perform identically to ground truth except for Bugs 1 and 11. Patches for Bugs 1 and 11 fail to fix 7 crashing inputs out of a larger duplicate crash set of 62 crashes reported by fuzzers. We analyzed these inputs and found that they generally trigger different crashing behavior downstream in execution that our approximate patches do not catch (but which correct patches handle earlier upstream). The implication is not severe: SCB+$\widehat{T}$ only reports 7 duplicates over all configurations, which is comparatively low compared to duplicate fuzzer reports.

w3m. SCB+$\widehat{T}$ perfectly simulates ground truth bucketing for `w3m`. Our approximate fixes are semantically close to developer fixes: each approximate patch checks the same program variable for `NULL` as the corresponding developer patch. Interestingly, Bug 13 produces far more duplicate crashes compared to the other three bugs across all configurations. This demonstrates a latent benefit of our approach: SCB can reveal properties about buggy behavior (e.g., we speculate that Bug 13 can be triggered along many execution paths and different call chains compared to the other bugs).

We confirmed that crash bucketing with T and $\widehat{T}$ result in *disjoint* buckets for multiple bugs in `SQLite` and `w3m`, and corresponds to the assumptions of Ideal Bucketing (i.e., zero overlap of crashing inputs of distinct bug fixes).

PHP. We applied SCB to one bug each in `PHP` v5 (CVE-2016-6292) and v7 (CVE-2016-10162). SCB improves over AFL's reports; the other configurations do not report duplicates.

R and **Conntrackd** both contain `strcpy` overflow bugs. The `R` bug is assigned CVE-2016-8714. We discovered a 0-day `strcpy` bug in `Conntrackd` in our own fuzzing efforts. Since no developer fix exists for a 0-day, we manually debugged to develop a ground truth patch. We have disclosed the bug and recommended the patch to the maintainers. As mentioned, Honggfuzz is particularly sensitive to changes in the stack, especially overflow vulnerabilities affecting the stack. Honggfuzz provides a way of blacklisting stack hashes to compensate,[10] but this option is disabled by default.

libmad. We also discovered a 0-day `memcpy` bug in `libmad` with our own fuzzing. We developed our own patch to perform the correct bounds checking on the length of bytes to copy. Interestingly, developers added a C `assert` statement before the `memcpy` call that checks the correct bounds. However, `assert` statements are not compiled in release versions and the bug results in a `SEGFAULT`. We used the assert statement to inform a ground truth fix for checking the buffer bound. Our deduplication gains is smaller for `libmad`, but remains precise. Our `libmad` example shows that approximate fixes extend to varieties of API calls in real world bugs with little effort.

6.3 Discussion

Merits of SCB and approximate fixes. Our approach can be layered on top of existing fuzzer deduplication methods or as a drop-in replacement. In general, SCB opens the opportunity to parameterize bucketing using targeted program transformation. One advantage of automated patch generation is resilience to changes in unrelated code across revisions. Concretely, we can generate an approximate fix for any revision containing the bug. This is not true for static, developer written patches. As explained, we had to take careful effort to isolate and backport existing patches for a ground truth study.

Approximate fixes can also improve fuzzing performance and coverage [40]. Fuzzers are known to get stuck on shallow bugs that restrict execution past a memory corruption bug.[11] Our approach provides a lightweight, parameterizable solution to augment fuzzing behavior and overcome such obstacles. We are currently investigating these extensions and additional bug classes.

Limitations. Our approach requires some up front manual cost to parameterize automated behavior for generating approximate fixes. Complexity of the targeted bug class also bears on the difficulty of specifying appropriate semantic cues and patch templates, and various approximations will affect accuracy of semantic bug containment. We demonstrated, however, that conceptually simple patch templates and semantic cues work well for common bugs found by fuzzers in real world programs. We speculate that the approach generalizes further to, e.g., division-by-zero, arithmetic overflows and use-after-free bugs. In general, we offer that one-off specifications per bug class is competitive with per-fuzzer configuration that preclude fine-grained semantic control.

Our time spent selecting projects to evaluate was dominated by whether we could find ground truth fixes and crashing inputs. Though the sample is small, every project that satisfied these criteria has worked with our approach so far (i.e., we do not fail to find an approximate fix), modulo the need for incremental refinements in our approach (e.g., we added a preprocessing step that expands macros in `PHP` to discover null dereference syntax when the program crashes).

In our experiments we observe that both approximate and developer fixes address unique bugs with a single check. Conceptually, we can imagine a case where some buggy behavior (e.g,. a null pointer) may be checked once before branching on multiple paths,

[10] https://github.com/google/honggfuzz/pull/29
[11] https://github.com/google/fuzzer-test-suite/tree/master/libxml2-v2.9.2

or alternatively along two different paths. Depending on the transformation, crashing inputs may thus map to one or two buckets. Our approach for null dereferences currently follows the second strategy for bucketing (since we add the check close to where the dereference occurs). In our experiments, this matched the behavior of developer fixes. Note that we can extend our approach to use first strategy (i.e., by searching for branch points and inserting checks upstream) and even compare different strategies; exploring and comparing such transformation strategies holds interesting potential for future work.

Due to the difficulty of performing organic fuzzing campaigns for known bugs, we purposely generate a crash corpus by mutating existing crashing inputs. The generated crash corpus likely inflates the number crashes that an organic campaign would encounter. Nevertheless, the crash corpus serves as a useful upper bound to quantify precision of default deduplication techniques in fuzzers versus SCB.

We evaluate our approach on existing state of the art fuzzers under default options and with small modifications to BFF and Honggfuzz. We recognize that deduplication can be tweaked and improved with additional parameters and post processing (e.g., stack hash blacklists), but we generally believe that fuzzers (like other tools), should run with sensible defaults.

7 RELATED WORK

Our approach relates generally to existing work in identifying bug uniqueness and bucketing crashing inputs [14, 17, 18, 41]. Of particular interest, Chen et al. [14] propose a machine learning approach that ranks interesting test cases for compiler fuzzer output, and use fixing patches as ground truth to map crashing inputs to unique bugs. Semantic Crash Bucketing draws on the idea of using ground truth fixes to precisely identify unique bugs, obtaining similar precision to ground truth by automatically approximating fixes.

Recent work by Pham et al. [41] uses a clustering algorithm that relies on a semantic characterization of inputs as constraints over paths, with particular applicability to symbolic executors. Our approach also promotes a semantic characterization of bugs, but focuses on being sensitive to semantic properties of bugs themselves, rather than summarizing crashing inputs in terms of path constraints. Broadly, current techniques manipulate and analyze program input or otherwise instrument programs to obtain "read-only" behavior of programs (such as input coverage [14], constraints on input [41], or crash callstack [37]) to group crashes. To the best of our knowledge, SCB is the first technique that appeals to *program modification* for precisely grouping crashing input in the absence of ground truth fixes.

Angelic debugging [13] seeks to modify programs by replacing expressions with values, which bears conceptual similarity to our approximating fixes for C library functions. Our problem focus differs, however: we seek accurate crash bucketing in the presence of duplicated or unreported bugs, while Angelic debugging seeks to fix failing test cases while preserving existing passing test cases.

In terms of program modification, our work relates to failure-oblivious computing [35, 44]. For instance, our rule-based application of fix templates share similarities with the idea proposed by Long et al. [35], who modify a program so that a null dereference does not cause it to crash. The objective of failure-oblivious computing, however, is to make program execution resilient to crash-inducing effects of bugs such as null dereferences or divide-by-zero errors. In contrast, SCB seeks to *isolate* unique bugs by selectively applying program transformation, rather than providing an automatic catch-all technique for keeping a program running in the interest of resilience. Syntactic patches promote the benefit of "patches as better bug reports" [47] so that engineers can analyze semantic effects that influence crash bucketing. Peng et al. [40] show that applying program transformation while fuzzing can increase program coverage and reveal more bugs; while our approach focuses on accurate crash bucketing, our technique complements this recent idea.

Fault localization [15, 25, 32, 43] and automatic program repair [30, 33, 49] share similar high level goals for identifying bugs and automatically fixing them. This work is broadly complementary to ours, providing techniques that can assist with accurately identifying fault locations for patch placement, and appropriate program transformations for different bug classes.

8 CONCLUSION

We introduced Semantic Crash Bucketing, a way to perform crash bucketing using lightweight program transformation. We then developed an automatic approach that applies patch templates to approximate real developer fixes to perform crash bucketing. Our approach uses configurable rules (specified once per bug class) that instantiate and apply patch templates based on crashing behavior. Unlike coarse deduplication methods, rules and templates are sensitive to bug-specific semantic properties and crashing behavior. We developed approximate fixes for null dereferences and buffer overflows. We performed a ground truth study comparing SCB and approximate fixes to (a) true developer fixes and (b) crash deduplication of three state of the art fuzzers (AFL, BFF, and Honggfuzz). Our results show that approximate fixes are competitive with crash bucketing precision of true developer fixes, and performs strictly better deduplication than all tested fuzzer configurations.

ACKNOWLEDGMENTS

This work is partially supported under NSF grant number CCF-1750116 and CCF-1563797. All statements are those of the authors, and do not necessarily reflect the views of the funding agency. The authors would like to thank Jeremy Lacomis for copy editing earlier versions of this work.

REFERENCES

[1] 2018. https://github.com/google/oss-fuzz. Online; accessed 26 April 2018.
[2] 2018. https://www.cert.org/vulnerability-analysis/tools/bff-download.cfm. Online; accessed 26 April, 2018.
[3] 2018. https://github.com/google/honggfuzz. Online; accessed 26 April, 2018.
[4] 2018. https://cve.mitre.org/. Online; accessed 26 April, 2018.
[5] 2018. https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html. Online; accessed 26 April, 2018.
[6] 2018. https://access.redhat.com/security/security-updates/#/cve. Online; accessed 26 April, 2018.
[7] 2018. AFL-Fuzz. http://lcamtuf.coredump.cx/afl/. Online; accessed 26 April, 2018.
[8] 2018. CVE-2017-12762. https://patchwork.kernel.org/patch/9880041/. Online; accessed 26 April, 2018.
[9] 2018. Microsoft Security Risk Detection. https://www.microsoft.com/en-us/security-risk-detection/. Online; accessed 26 April, 2018.

[10] 2018. Public Vulnerabilities Discovered Using BFF. https://vuls.cert.org/confluence/display/tools/Public+Vulnerabilities+Discovered+Using+BFF. Online; accessed 26 April, 2019.

[11] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating focused random tests using directed swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA '16)*. 70–81.

[12] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *International Conference on Software Engineering (ICSE '14)*. 1083–1094.

[13] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. 2011. Angelic debugging. In *International Conference on Software Engineering (ICSE '11)*. 121–130.

[14] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Conference on Programming Language Design and Implementation (PLDI '13)*. 197–208.

[15] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *International Conference on Software Engineering (ICSE '05)*. 342–351.

[16] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *International Conference on Software Engineering (ICSE '13)*. 792–801.

[17] 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 *International Conference on Software Engineering (ICSE '16)*. 820–831.

[18] 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 *International Conference on Software Engineering (ICSE '12)*. 1084–1093.

[19] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. 2003. Buffer overrun detection using linear programming and static analysis. In *Conference on Computer and Communications Security (CCS '03)*. 345–354.

[20] Patrice Godefroid and Daniel Luchaup. 2011. Automatic partial loop summarization in dynamic test generation. In *International Symposium on Software Testing and Analysis (ISSTA '11)*. 23.

[21] Denis Gopan, Evan Driscoll, Ducson Nguyen, Dimitri Naydich, Alexey Loginov, and David Melski. 2015. Data-Delineation in Software Binaries and its Application to Buffer-Overrun Discovery. In *International Conference on Software Engineering (ICSE '15)*. 145–155.

[22] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2017. The Theory of Composite Faults. In *International Conference on Software Testing, Verification (ICST '17)*. 47–57.

[23] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA '12)*. 78–88.

[24] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE '06)*. 232–241.

[25] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated Software Engineering (ASE '05)*. 273–282.

[26] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE '13)*. 802–811.

[27] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. 2015. Automatic Rootcausing for Program Equivalence Failures in Binaries. In *Computer Aided Verification (CAV '15)*. 362–379.

[28] David Larochelle and David Evans. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*.

[29] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering (ICSE '12)*. 3–13.

[30] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443.

[31] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Conference on Computer and Communications Security (CCS '17)*. 2201–2215.

[32] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Programming Language Design and Implementation (PLDI '05)*. 15–26.

[33] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *Symposium on Information, Computer and Communications Security*. 329–340.

[34] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Principles of Programming Languages (POPL '16)*. 298–31.

[35] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *Conference on Programming Language Design and Implementation (PLDI '14)*. 227–238.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering (ICSE '16)*. 691–701.

[37] D Molnar, XC Li, and DA Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*. 67–82.

[38] Paul Muntean, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert. 2015. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. In *Computer Safety, Reliability, and Security (SAFECOMP '15)*. 441–456.

[39] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. *International Conference on Software Engineering*, 772–781.

[40] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*.

[41] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing Failing Tests via Symbolic Analysis. In *Fundamental Approaches to Software Engineering Conference (FASE '17)*. 43–59.

[42] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*. 861–875.

[43] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *International Conference on Automated Software Engineering (ASE '03)*. 30–39.

[44] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing.. In *OSDI*, Vol. 4. 21–21.

[45] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*.

[46] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *International Conference on Mining Software Repositories (MSR '16)*. 512–515.

[47] Westley Weimer. 2006. Patches as better bug reports. In *Generative Programming and Component Engineering (GPCE '06)*. 181–190.

[48] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Conference on Computer & Communications Security (CCS '13)*. 511–522.

[49] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55.