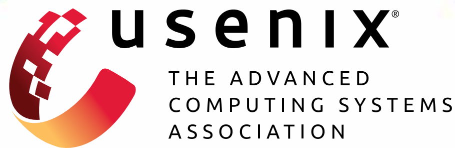

FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing

Zenong Zhang and Zach Patterson, *University of Texas at Dallas;* Michael Hicks, *University of Maryland and Amazon;* Shiyi Wei, *University of Texas at Dallas*

https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the
31st USENIX Security Symposium is
sponsored by USENIX.

FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing

Zenong Zhang[†], *Zach Patterson*[†], *Michael Hicks*[‡], *and Shiyi Wei*[†]
[†]*University of Texas at Dallas* [‡] *University of Maryland and Amazon*[*]

Abstract

Fuzz testing is an active area of research with proposed improvements published at a rapid pace. Such proposals are assessed *empirically*: Can they be shown to perform better than the status quo? Such an assessment requires a benchmark of target programs with well-identified, realistic bugs. To ease the construction of such a benchmark, this paper presents FixReverter, a tool that automatically injects realistic bugs in a program. FixReverter takes as input a *bugfix pattern* which contains both code syntax and semantic conditions. Any code site that matches the specified syntax is *undone* if the semantic conditions are satisfied, as checked by static analysis, thus (re)introducing a likely bug. This paper focuses on three bugfix patterns, which we call *conditional-abort*, *conditional-execute*, and *conditional-assign*, based on a study of fixes in a corpus of Common Vulnerabilities and Exposures (CVEs). Using FixReverter we have built RevBugBench, which consists of 10 programs into which we have injected nearly 8,000 bugs; the programs are taken from FuzzBench and Binutils, and represent common targets of fuzzing evaluations. We have integrated RevBugBench into the FuzzBench service, and used it to evaluate five fuzzers. Fuzzing performance varies by fuzzer and program, as desired/expected. Overall, 219 unique bugs were reported, 19% of which were detected by just one fuzzer.

1 Introduction

Fuzz testing (a.k.a. *fuzzing*) has proved to be surprisingly successful at discovering security vulnerabilities. For example, AFL [1], one of the most mature and widely used fuzzers, has an extensive trophy case. This success has spurred research toward addressing fuzzing's weaknesses, with dozens of published improvements in the last few years [2].

Most proposed fuzzing improvements are judged *empirically*. A proposed improvement's implementation is evaluated by running it on a set of target programs, comparing its performance against that of one or more baseline fuzzers.

A key question is what performance measure to use. One popular measure, employed by Google's FuzzBench [3], is *code coverage*; if a fuzzer *A* (the improvement) is able to generate tests that execute more distinct lines/branches in a target program than the baseline *B*, then one could argue *A* will find more bugs. Multiple studies have been performed with the goal of understanding the relationship between code coverage and bug finding [4–6]. Unfortunately, a recent study finds that while there is a strong correlation between the coverage achieved and the number of bugs found by a fuzzer, there is not strong agreement on which fuzzer is superior if coverage is used to compare the fuzzers [7].

Another popular measure is to count the number of distinct, crash-inducing inputs generated, a.k.a. *unique crashes*. Since two different inputs can easily trigger the same bug, researchers often employ deduplication heuristics; two popular heuristics are AFL's "coverage profiles" and fuzzy stack hashes [8]. However, a study by Klees et al. [9] showed that both heuristics could still yield many false positives (many "deduplicated" inputs still trigger the same bug) and also some false negatives ("deduplicating" an input can actually remove evidence of a distinct bug). For one program, their study found that a result that appeared to show fuzzer *A* was superior to baseline *B* disappeared when ground truth was used, rather than "unique" crash heuristics.

1.1 Developing a Fuzzing Benchmark

Klees et al. recommended developing a *benchmark* of buggy programs with which to empirically compare fuzzer implementations [9]. Systematizing their advice, we identify four goals for an effective benchmark suite:

G1 it should use relevant, real-world target programs;

G2 those programs should contain realistic, relevant bugs (e.g., memory corruption/crash bugs);

[*]Work done prior to starting at Amazon.

G3 these bugs should be triggerable in a way that clearly indicates when a particular bug is found, to avoid problems with deduplication;

G4 the benchmark should defend against overfitting.

There are several extant fuzzing benchmarks, but none meet all of these criteria. UNIFUZZ [10] comprises many real-world programs with known bugs, satisfying **G1** and **G2**. But triggering a bug does not emit a telltale sign; instead, UNIFUZZ distinguishes bugs based on stack traces, which is unreliable per Klees et al. [9], thus violating **G3**. Google FuzzBench also comprises dozens of real-world programs, but many of them do not have known bugs; FuzzBench focuses on measuring code coverage, not bugs triggered. For those programs that have bugs, it uses unreliable means (stack traces) to distinguish them. The DARPA Cyber Grand Challenge (CGC) [11] has well-identified (**G3**) and realistic (**G2**) bugs, but its programs are synthetic, violating **G1**. Magma [12] satisfies **G1**, **G2**, and **G3**: it comprises real-world programs with relevant bugs injected by hand so that when a bug is triggered, it gives a telltale sign.

An issue with any benchmark, and with each of these in particular, is that it can be prone to overfitting (**G4**). Fuzzer developers may start to employ heuristics and strategies that do not serve the general goal of finding more vulnerabilities, but instead simply aim to perform better on a particular benchmark [9]. A benchmark built around automated *fault injection* can help avoid the overfitting problem and satisfy the other goals, too: (1) bugs can be automatically injected into real programs in a way that the bug signals when triggered, and (2) the tool can be used to produce new, fault-injected programs as often as needed, to prevent fuzzers from overfitting to a fixed set of programs.

LAVA [13] is an approach for adding faults to a program automatically, and LAVA-M is a collection of four programs with LAVA-injected bugs, which has proved to be a popular evaluation target in the fuzzing literature [14–18]. LAVA works by injecting code snippets that each consist of a branch with an unusual condition; if the condition is true, the program faults (with a telltale sign). Apocalypse [19] generalizes this approach by injecting conditions that implement an *error transition system*, with a fault induced when the final state is reached. Both LAVA and Apocalypse arguably fail **G2**, since the injected patterns do not resemble realistic bugs. For LAVA there is also concern (noted by the LAVA authors [20]) that the injected pattern is easily gamed, reducing the benefit of automating fault injection for evaluation purposes. EvilCoder [21] uses static dataflow analysis as the basis for injecting realistic source/sink bugs, but as far as we can tell has never been used at scale.

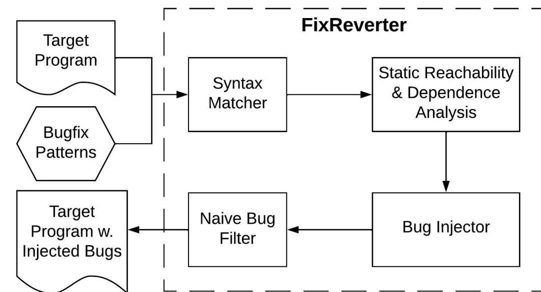

Figure 1: The architecture of FIXREVERTER.

1.2 Our proposal: FIXREVERTER

This paper presents FIXREVERTER, a new fault injection tool, and REVBUGBENCH, a benchmark we have produced using it. To increase the realism of its injected faults, FIXREVERTER employs patterns that match fixes of previous CVEs [22]. In particular, FIXREVERTER is instructed to find a pattern that matches an observed fix, and then *reverses* that pattern, aiming to undo the "fix" and thereby (re)introduce a bug.

Figure 1 depicts FIXREVERTER. It takes as inputs a program into which to inject bugs, and descriptions of bugfix patterns. Bugfix patterns have two components: a *syntactic pattern* that identifies, via a grammar, code at a potential injection site, and a *semantic condition* that indicates whether reverting the matched code could lead to a crash.

Based on a study of 814 CVEs (Section 2) we have developed three bugfix patterns, which we call *conditional-abort* (`ABORT`), *conditional-execute* (`EXEC`), and *conditional-assign* (`ASSIGN`). The `ABORT` pattern matches a fix that aborts continued forward execution if the condition is true. One instance of this pattern is CVE-2017-8395 [23], shown in Figure 2 (under the comment *PR 21431*), which returns early if a pointer variable is NULL, in order to avoid subsequently dereferencing it. The `EXEC` pattern matches a fix that adds additional constraints to a conditional to prevent erroneous execution of its body. The `ASSIGN` pattern matches a fix that introduces a conditional assignment to a variable, to prevent erroneous execution involving that variable in the code that follows it.

For each pattern, FIXREVERTER first matches the syntax of the pattern in the target program according to a grammar (Section 3.1), thus identifying candidate injection sites (Section 3.2). Then, for each candidate injection site that matches this code pattern, FIXREVERTER checks the semantic condition, which if satisfied means that reversing the fix will likely result in a triggerable crash. It does this via a *static reachability & dependence analysis* (Section 3.3). This analysis first checks whether the injection site may be *reachable* via a feasible program execution starting at an entry point (e.g., `main`). For example, the NULL-check site in Figure 2 is

reachable from main through the orange call edges. Second, it (inter-procedurally) checks whether there is dataflow from a relevant variable or field to a possible dereference site. For the example in Figure 2, the variable is uncompressed_buffer and the analysis would detect that this variable flows to a dereference within the fread call (through the blue dataflow edges). If the analysis finds no dereference site to which a variable flows, FixReverter will not reverse the code pattern, since it has no specific evidence that doing so will induce a crash.

After deciding the final set of bugs, FixReverter injects them (Section 3.4). For *ABORT* pattern, this means deleting the check and return (or dropping a disjunctive clause). For *EXEC* pattern, this means loosening the condition. For *ASSIGN* pattern, this means removing the conditional assignment. FixReverter's injector avoids introducing new branches that may be instrumented by the fuzzers and affect their feedback, similar to Magma's by-hand injections [12].

To avoid injecting uninteresting bugs which could fail too easily, FixReverter runs a *naive bug filter*. One possible filter is the target program's regression test suite—any injected bug that fails the regression tests should be discarded. Another strategy, which we used to develop RevBugBench, is to run the program with injected bugs using a fuzzing seed, and we do not inject those triggered by the seed.

To report the specific cause(s) of a crash due to some input I, we apply a novel triage procedure (Section 3.5). To have clear indicators which injected bugs are triggered in a crashing input, FixReverter logs when the guards from the removed and updated conditions hold. The triage procedure identifies two kinds of cause: (1) an *individual cause* is a single bug that is sufficient to produce the crash with I, on its own; (2) a *combination cause* is a set of bugs that must all be present for I to crash. Sometimes a single input can induce multiple crashes with distinct causes; e.g., input I produced by a fuzzer when bugs A, B, and C are injected could triage to having both A and B, individually, as causes (*both* need not be present, but if either is, the program crashes).

1.3 A New Benchmark: RevBugBench

We construct RevBugBench (Section 4) by running FixReverter on 8 programs in FuzzBench and 2 Binutils programs, all used in prior fuzzing evaluations (**G1**). In total, nearly 8000 bugs are injected. We integrated RevBugBench into Google's FuzzBench service to simplify its use in evaluating fuzzers at scale.

We evaluated 5 fuzzers on RevBugBench (Section 5): AFL [1], libFuzzer [24], AFL++ [25], Eclipser [16], and FairFuzz [26]. In total, the fuzzers triggered 219 individual-bug causes, 19% of which were detected by at most one fuzzer, and 221 additional combination causes. Performance varied by fuzzer and target program, with AFL++ doing the best overall.

In summary, RevBugBench contains 10 real-world pro-

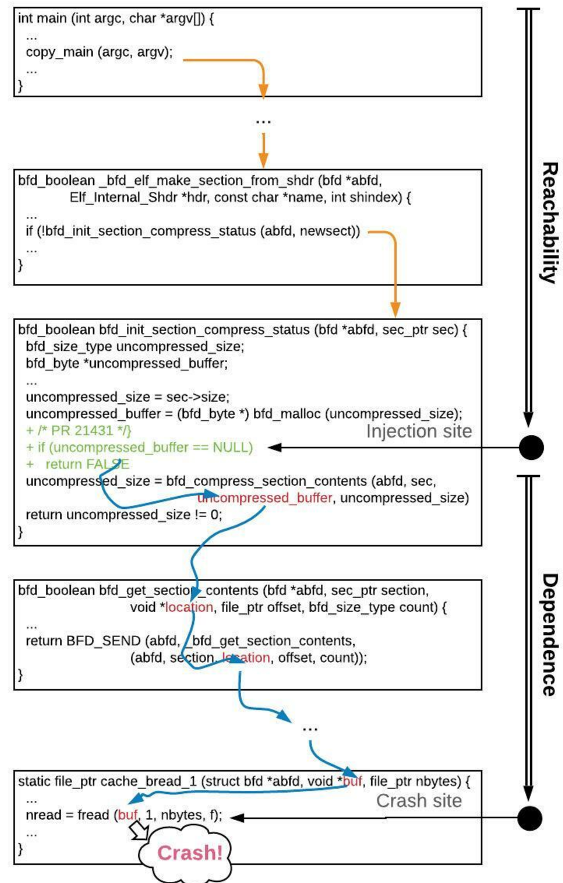

Figure 2: Illustration of FixReverter: CVE-2017-8395

grams from prior fuzzing evaluations, satisfying **G1**. **G2** follows from the reversion of code patterns based on real CVE bugfixes, and **G3** follows from the injection logic that identifies when a bug is triggered. There is the caveat that a reverted fix is not provably triggerable because FixReverter's static analysis is necessarily overapproximate, and moreover injected bugs may interact in ways that the analysis did not anticipate. Nevertheless, as a practical matter we find that many injected bugs *are* triggerable by modern fuzzers, and many more are not; moreover, our triage procedure allows the root causes of a crash to be discerned. Finally, **G4** is satisfied because while fuzzers may overfit to a static set of injected bugfix patterns, the use of FixReverter allows RevBugBench to be expanded with new patterns and programs, or by creating benchmark variants using a sample of injected bugs. We hope RevBugBench's integration into FuzzBench could ensure its continued use and evolution.

Contributions This paper made the following contributions:

- The identification of the *conditional-abort*, *conditional-*

execute, and *conditional-assign* bugfix patterns, defined by a combination of code syntax and a semantic condition, discerned from a study of bugfixes of CVEs.

- A framework, FixReverter, that automatically injects bugs by reversing instances of the three patterns, detected by the use of syntax matching and static analysis.
- A fuzzing benchmark RevBugBench based on running FixReverter on 10 programs taken from Binutils and FuzzBench, and integrated into FuzzBench along with a novel triaging algorithm.
- An evaluation that compares 5 state-of-the-art fuzzers using RevBugBench, with results that suggest its usefulness as a benchmark.

2 Bugfix Patterns

We inspected past bugfixes in the Common Vulnerabilities and Exposures reports (CVEs) to find relevant bugfix patterns. In particular, we studied 693 CVEs from six open source programs (Binutils [27], Tcpdump [28], libxml2 [29], FFmpeg [30], libarchive [31] and systemd [32]). We chose these programs for three reasons. First, they are well studied in the fuzzing literature [9, 15, 26, 33–37], and contain well-documented CVEs. Second, most of their CVE records contain direct links to the source code diff of the bugfix. Third, most of their CVE records are reproducible and come with call stacks to help understand the origins of the bugs and how the bugfixes work. In addition, we also studied the 121 CVEs and bugfixes that were used to construct the Magma benchmark [12].

For each CVE, we spent 10 minutes with the provided test input, call stack, and code diffs to understand the cause of the bug and the developer's intention with the bugfix. For all of the CVEs together, we tried to identify common patterns.

In the end, we used 170 CVEs from the 814 as a basis for three general bugfix patterns we call *conditional-abort* (`ABORT`), *conditional-execute* (`EXEC`), and *conditional-assign* (`ASSIGN`), which we discuss in detail shortly. Among them, 10 CVEs were used as basis of two patterns. While other patterns are possible, `ABORT`, `EXEC`, and `ASSIGN` represent general, intuitive patterns to demonstrate the effectiveness of FixReverter. Indeed, bugfixes similar to them were considered in previous studies of bug repositories [38–40] and automatic bugfixing tools [41–43]. We call these CVEs that are the basis of the three patterns our *bugfix dataset*.

ABORT An `ABORT` fix is characterized by the addition of an if-statement that checks that a variable (or path) involved in a downstream dereference satisfies an invariant, and breaks the flow of control (e.g., returns from the function) if it does not. Such a bugfix *prevents subsequent program execution* from dereferencing a pointer whose value is dependent on the checked variable. Figure 2 shows an `ABORT` bugfix, as

```
1  bfd_boolean _bfd_dwarf2_find_nearest_line
2  (asymbol **symbols,  /*other parameters*/)
3  {
4    ...
5    - if ((section->flags & SEC_CODE) == 0)
6    + if (symbols != NULL && (section->flags & SEC_CODE) == 0)
7    {
8      asymbol **tmp;
9      for (tmp = symbols; (*tmp) != NULL; ++tmp)
10     ...
11   }
12 }
```

Figure 3: CVE-2017-8392 bugfix.

explained in Section 1.2. `ABORT` is the most common pattern in our dataset, matching 155 CVEs. A generalization of this pattern is the addition of a disjunctive clause to an already-present aborting conditional block, e.g., `if(p) return;` becomes `if (q || p) return;`.

EXEC An `EXEC` fix is characterized by the addition of a conjunctive boolean expression to an existing conditional statement (*if*, *while* and *for*) to check that a variable (or path) involved in a dereference within the conditional's body satisfies an invariant. Such a bugfix *tightens* the condition of executing the true branch of the conditional statement which dereferences a pointer whose value is dependent on the checked variable. Figure 3 shows the bugfix of CVE-2017-8392 [44] with the `EXEC` pattern. In line 9, `symbols` is copied to `tmp` which is dereferenced. The developer added a check of `symbols` against NULL in line 6 to avoid the NULL dereference. 9 CVEs in our dataset match the `EXEC` pattern.

ASSIGN An `ASSIGN` fix is characterized by the addition of a new if-statement whose body is the assignment to a variable that is involved in a downstream dereference; the conditional guard may or may not involve the same variable. Such a bugfix changes the value of the assigned variable which is used in a dereference in the subsequent program execution. Figure 4 is an example of the `ASSIGN` pattern. In CVE-2013-0211 [45], parameter `s` may cause an overwrite error in line 8 if its value exceeds the maximum value to safely write to the buffer. To avoid such a bug, the developer constrained the value of `s` by adding the if-statement in lines 5 and 6. In our dataset 16 CVEs match the `ASSIGN` pattern.

All of these patterns involve conditional statements, but the pattern of fix reversion is different for each, as is the code affected by that reversion (e.g., either within the conditional itself, or in subsequent program execution), which affects the semantic conditions under which that pattern applies.

```
1 static ssize_t
2 _archive_write_data(size_t s,  /*other parameters*/)
3 {
4   ...
5   + if (s > max_write)
6   +    s = max_write;
7   archive_clear_error(&a−>archive);
8   return ((a−>format_write_data)(a, buff, s));
9 }
```

Figure 4: CVE-2013-0211 bugfix.

```
1  ABORT → if ABORT_CONDS ABORT_BODY
2  ABORT_CONDS → ABORT_COND
3               | ABORT_COND || ABORT_CONDS
4  ABORT_BODY → JUMP | upto3 JUMP
5  JUMP → break | goto | return | continue
6
7  EXEC → IF_WHILE_FOR EXEC_CONDS EXEC_BODY
8        | if EXEC_CONDS EXEC_BODY else EXEC_BODY
9  EXEC_CONDS → EXEC_COND && EXEC_COND
10              | EXEC_COND && EXEC_CONDS
11 EXEC_BODY → no_jump
12 IF_WHILE_FOR → if | while | for
13
14 ASSIGN → if ASSIGN_COND ASSIGN_BODY
15 ASSIGN_BODY → TRACER = rhs_assign
16
17 TRACER → PTR_TRACER | NUM_TRACER
18 PTR_TRACER → ptrVar:traceVar[]
19            | ptrVar:traceBase[] -> ptrVar:traceField[]
20            | var:traceBase[] . ptrVar:traceField[]
21
22 NUM_TRACER → num:traceVar[]
23            | ptrVar:traceBase[] -> num:traceField[]
24            | var:traceBase[] . num:traceField[]
```

Figure 5: Part of grammar for the *ABORT*, *EXEC*, and *ASSIGN* patterns.

3 FIXREVERTER

The first step of FIXREVERTER (Figure 1) is to perform a syntactic search for a *fix pattern* that could be reversed in order to inject a bug (Section 3.1). FIXREVERTER's grammar-based *syntax matcher* analyzes all files in the target program to find the code regions that match each given syntactic bugfix pattern (Section 3.2). Each matching region is a candidate injection site. Next, using information returned from the syntax matcher, the *static reachability and dependence analysis* confirms that *traced variables* in the matched pattern may reach a dereference site that could result in a crash (Section 3.3). FIXREVERTER then *injects bugs* in the target program by reverting the patterns confirmed by the static analysis (Section 3.4), but filters uninteresting bugs through a *naive bug filter*. Finally, after a fuzzing campaign, we must triage which injected bugs were the cause of an observed failure in the target program (Section 3.5).

3.1 Bugfix Pattern Grammar

We express the bugfix patterns using a context-free grammar (CFG), shown in Figure 5. By convention, non-terminals are in uppercase and terminals are in lowercase.

The CFG specifies that an **if** statement falls into the *ABORT* pattern (lines 1-5) when (1) it has no else branch; (2) it has a small body with up to 3 statements;[1] (3) the body ends with a jump instruction (i.e., **return**, **break**, **goto** or **continue**); and (4) when an **if** guard has multiple conditions, they are connected by the logical OR operator.

The *EXEC* pattern accepts both **while** and **for** statements; it also accepts **if** statements with or without **else** (lines 7 and 8). The condition requires at least two conditional expressions, connected by the logical AND operator (lines 9 and 10), and the body of the *EXEC* pattern cannot contain a jump instruction (line 11).[2]

Lines 14 and 15 specify that an **if** statement falls into the *ASSIGN* pattern if (1) it only has a single condition, and (2) its body is a single assignment statement.

[1]We express this as a custom terminal **upto3** in the grammar.

[2]Expressed as a custom terminal **no_jump**.

In our grammar, a terminal may be associated with a *tracer*, which is an annotation following a colon, e.g, `traceVar[]` in **ptrVar**:`traceVar[]` on line 18. Tracers are used by FIXREVERTER to identify the symbols whose flow should be followed to dereference sites, thus enforcing the semantic condition of the bugfix patterns, as detailed in Section 3.2. For example, the left-hand side of the assignment statement of the *ASSIGN* pattern is traced (line 15).

In Figure 6, we show the grammar rules that define the different conditional expressions for each pattern (lines 1-3). We define *EQ_NULL* (lines 4 and 5) and *NOT_EQ_NULL* (lines 6 and 7) as the expressions that match a pointer tracer that is checked to be equal and not equal to **null**, respectively. We define *PTR_CMP* (line 9) as the expressions that compare $(<, \leq, >, \geq)$ two pointer operations, and *PTR_REL* (line 10) as the expressions that check the equality (==, !=) of two pointer operations. A pointer operation is a pointer tracer or the arithmetic operation $(+, -, \times, /)$ between a pointer tracer and a number (lines 11-13). We define *NUM_CMP* (lines 14-15) as the expressions that check between two numeric operations (i.e., a numeric value or the arithmetic operation between two numeric values). A numeric value is a literal number, a numeric tracer (i.e., a numeric variable or field), or the result of evaluating a function call or array access that returns a numeric type (lines 17-21).

```
1  ABORT_COND → EQ_NULL | PTR_CMP | NUM_CMP
2  EXEC_COND → NOT_EQ_NULL | PTR_CMP | NUM_CMP
3  ASSIGN_COND → NOT_EQ_NULL | EQ_NULL | PTR_CMP |
       PTR_REL | NUM_CMP
4  EQ_NULL → PTR_TRACER == ZEROVAL
5            | ZEROVAL == PTR_TRACER
6  NOT_EQ_NULL → PTR_TRACER != ZEROVAL
7                | ZEROVAL != PTR_TRACER
8  ZEROVAL → null | cast null
9  PTR_CMP → PTR_OP CMP PTR_OP
10 PTR_REL → PTR_OP REL_OP PTR_OP
11 PTR_OP → PTR_TRACER ARITH NUM_VAL
12          | NUM_VAL ARITH PTR_TRACER
13          | PTR_TRACER
14 NUM_CMP → NUM_OP CMP NUM_OP
15          | NUM_OP REL_OP NUM_OP
16 NUM_OP → NUM_VAL ARITH NUM_VAL | NUM_VAL
17 NUM_VAL → NUM | NUM_TRACER
18           | SINGLECALLER ( ptrVar )
19           | SINGLECALLER ( ptrVar [ NUM ] )
20           | SINGLECALLER ( var )
21           | SINGLECALLER ( NUM )
22 NUM → lit_num | null
23 SINGLECALLER → sizeof | functionCall
24 ARITH → + | - | * | /
25 CMP → < | > | <= | >=
26 REL_OP → == | !=
```

Figure 6: Grammar of the conditional expressions in the *ABORT*, *EXEC*, and *ASSIGN* patterns.

3.2 Syntax Matcher

FIXREVERTER's syntax matcher is implemented as a Clang LibTool [46] (version 12). It works in two phases. First, it reads in a grammar file just discussed, converting it into a *state machine*. Phase 2 traverses the Clang abstract syntax tree (AST) using the visitor pattern. As it encounters statements, it finds tokens in each statement and feeds them into the state machine to see if it matches a defined pattern. For the statements matched, gathered tokens are placed into a JSON file to be used in the later stages of FIXREVERTER.

Phase 1 When running the syntax matcher, a pattern file is specified as a command-line argument. This file takes a CFG with lists of productions $NT \rightarrow Token_1\ Token_2\ ...\ Token_n$, where NT is a non-terminal, and a token can be either a non-terminal or a terminal. In the grammar, a terminal may be associated with a *tracer*. Information about the traced terminals is important for the next stage of FIXREVERTER to satisfy the semantic condition in the bugfix patterns. For example, the name and declaration of a variable that appears in the conditional expression of an *ABORT* pattern's if-statement are used by the static analysis. Traced terminals are written to a JSON file with their source location and string value.

Tracers can be specified in two ways in the input grammar. (1) **terminal**:tracer traces the same terminal in different places in the grammar. For example, *NT* → **var**:t1 **< num** && **var**:t1**> num** requires the same variable to appear in both locations of this expression. (2) **terminal**:tracer[] traces multiple terminals in the grammar. For example, the pointer tracers in lines 18-20 in Figure 5 ensure all variables, base pointers, and fields that match *PTR_TRACER* are traced and kept in traceVar[], traceBase[], and traceField[], respectively.

In phase 1, the grammar file is first parsed into a list of productions; then, we use a generalized LR(0) parser [47] to create a state machine. Each state represents a possible position in the grammar. State transitions are determined by what the possible tokens are. In this generalized parser, if there are two or more state transitions that could be taken, it forks into multiple parsers where each takes one of the actions. This allows temporary ambiguity. Since each of the three patterns we defined is mutually exclusive, this will eventually result in one match; when conflicts are found in any forked parser, it terminates.

Phase 2 After creating the state machine, the syntax matcher traverses the AST using the visitor pattern provided by Clang LibTool. For every function-statement node, if the function exists in the original C file, we traverse its body; otherwise (e.g., a function defined in a header file), we skip over it. For each if-statement, for-loop, or while-loop, we start executing the state machine. For each subsequent AST node visited, we extract terminals from the AST node depending on the node type, and continue executing the state machine using the terminals. Once the visitor pattern returns to the node which initiated the state machine, each fork of the parser is checked. If its state machine is in an accepting state, then the traced terminals fed into that branch are output by the syntax matcher. For example, according to Figure 5, when the visitor leaves an if-body, if the state machine has parsed **if** *ABORT_CONDS ABORT_BODY*, it will move into an accepting state, and a match for the *ABORT* pattern will be output.

While most of the terminals we process are individual operators or variables, our custom terminals allow us to treat combinations of statements or expressions differently for better expressiveness and flexibility. For example, terminal **upto3** in line 4 of Figure 5 represents any compound statement body which has fewer than or equal to 3 internal statements; terminal **no_jump** in line 11 of Figure 5 represents a single or compound statement that does not contain a *JUMP* statement. In addition, we can differentiate tokens by passing a different terminal based on a value. For an integer literal, we process **null** if the value is 0, and **lit_num** in any other case (line 22 in Figure 6). We also distinguish some terminals based on the type of a variable, or whether that variable is a pointer. For example, in *EQ_NULL*, we require a pointer tracer to be compared to the value 0 at lines 4-5 in Figure 6.

Running example We illustrate FIXREVERTER's workflow with an example. Figure 7a shows the bugfix of CVE-2017-7303 [48] in Binutils. It checks whether the variable

```
1  for(...)
2  + if (oheader == NULL)
3  +   continue;
4    if (section_match (oheader, iheader))
5        return i;
```

(a) CVE-2017-7303 bugfix.

```
1  for(...)
2    #ifdef FRCOV
3    if (injectFlag[529]) {
4      if (oheader == NULL && !(0))
5        log("trigger bug index 529");
6      else
7        log("reach bug index 529");
8    }
9    if ((injectFlag[529] && 0)
10     || (!injectFlag[529] && (oheader == NULL)))
11   #else
12   if (0)
13   #endif
14     continue;
15   if (section_match (oheader, iheader))
16     return i;
```

(b) Injected code. Injection index is 529.

Figure 7: Running example.

oheader is NULL, and skips the rest of the loop if so, to avoid a subsequent NULL dereference. In REVBUGBENCH (Section 4), FIXREVERTER reverted this bugfix, (re)injecting a bug in the target program *disassemble*. FIXREVERTER's syntax matcher finds the if-statement in lines 2-3 because it matches the grammar of the *ABORT* pattern in Figure 5; the variable oheader is considered a traced variable.

3.3 Reachability & Dependence Analysis

FIXREVERTER's *static reachability & dependence analysis* takes as input the set of tracers output by the syntax matcher. For *ABORT* and *EXEC*, the tracers occur in the conditional guard; for *ASSIGN*, the tracer is the assigned-to variable.

The static analysis does two things (visualized by the two vertical lines on the right in Figure 2). First, it decides whether this tracer *may be reached* via an execution from a designated entry point. Second, it determines if a subsequent pointer dereference *may be dependent* on the tracer, i.e., whether the latter can influence it. In sum, the tracer is a *source* and the subsequent dereference is a *sink*. Some examples of the sinks are reading/writing of a traced pointer, offsetting a pointer using a traced number, and calling a library function within which a traced argument is dereferenced. For *ABORT* and *ASSIGN*, the dereference must occur *after* the matched conditional. For *ABORT*, if we remove the matched `if`, thereby reverting the "fix", execution will reach the sink when it would not have before, creating the potential to produce a crash. For *ASSIGN*, it will reach the sink without having executed the reverted assignment. The semantic condition of *EXEC* is different: the subsequent dereference must occur *within* the body of the if/while/for whose guard contains the tracer. Fix reversion drops a clause in the guard, making it possible for execution to reach a sink it would not have, producing a crash.

Running example Analyzing the example in Figure 7a, FIXREVERTER's static reachability analysis finds that this *ABORT* pattern appears in the function find_link, which is reachable from the entry function of disassemble. The dependence analysis tracks that the variable oheader is the argument passed to section_match, which dereferences it. Thus there is a source-sink flow between oheader and the dereference site, confirming the semantic condition of this pattern.

Implementation We built the analysis on top of the Phasar C-code static analysis framework [49]. Phasar provides a summary-based solver of Inter-procedural Finite Distributive Subset (IFDS) [50] problems, including taint analysis. IFDS solves a dataflow problem by constructing an exploded supergraph (ESG), replacing every node of its inter-procedural control-flow graph (ICFG) with the bipartite graph representation of the respective flow function [50]. Our implementation is based on Phasar release v0521 [51] using LLVM 12.0 [52].

FIXREVERTER's analysis extends Phasar's existing IFDSTaintAnalysis module. It takes a set of entry functions to start solving the dataflow problem. It uses the ICFG to decide which sources are reachable from these entry functions, and then uses the taint analysis to identify dependences between these reachable sources and possible sinks. The analysis is context-sensitive and field-based [53]. When constructing the ICFG, it uses a pointer analysis [53] to resolve the targets for function pointers. The analysis operates on the whole-program LLVM Intermediate Representation (IR) [52], generated using WLLVM [54] and LLVM disassembler `llvm-dis`.

Our static dependence analysis addressed several limitations of Phasar's IFDSTaintAnalysis module. First, Phasar's IFDSTaintAnalysis module ignores fields when propagating taintedness, which makes the taint tracking unsound. We implemented a field-based [53] analysis by extending the IFDSTaintAnalysis module with a global set. This set stores the tuple <*type of its base pointer, field offset*>, called the *field tuple*. When a field is tainted through propagation, its field tuple is added to the set. When there is a field access, we look up the set. If its field tuple exists in the set, the field is tainted.

Second, Phasar's IFDSTaintAnalysis module does not keep track of the origin of the taintedness of a variable. Thus, it cannot report data-dependent source/sink pairs, only that a sink is tainted. To tell which tracer may lead to an exploitable memory error, we extend the IFDSTaintAnalysis module with a global map to keep track of the sources of tainted variables. The key of the map is called the *taint-track key*, which is either a tainted variable or a field tuple. Each taint-track key maps to the list of taint-track keys that its taintedness originates from.

The map is updated each time a taint-track key is tainted. When the analysis completes, we use the map to output the pairs of sources/sinks that have a data dependence.

To identify the sources of the static dependence analysis, we use line and column numbers of all traced variable and base pointer declarations produced by the syntax matcher, and match against the debug information in *llvm.dbg.declare* instructions [55] (target programs are compiled with no optimization to generate these instructions). For each traced variable, we retrieve the corresponding IR register as the source. For each traced field, we add to a global data structure for our field-based analysis the field tuple <*type-of-traceBase, traceField-offset*>. This tuple is also treated as the source of the dependence analysis. We identify the sinks (i.e., pointer dereferences) as IR instructions that perform a dereference.

Limitations The analysis inherits two limitations of Phasar's IFDSTaintAnalysis. First, Phasar's control-flow analysis is unsound, in part due to an imprecise treatment of function pointers, meaning it may incorrectly claim that some functions are unreachable. As such, some injection sites may be unnecessarily ruled out. To get a sense of the impact of this unsoundness, we ran an experiment in which we ignored the static analysis and injected *all* `ABORT` bugs at candidate injection sites in the 10 programs in Table 1 and we fuzzed those programs with AFL++ for 24 hours. We found that in addition to the 102 bugs found by AFL++ that were approved by the static analysis, there were 14 more that were incorrectly filtered out by it. We are working with the Phasar authors to fix this unsoundness. Second, sinks that are not in the application code are specified as a list of library-function parameters; this by-hand specification runs the risk of missing some sinks.

3.4 Bug Injection

We implement the FIXREVERTER bug injector as a Clang LibTool [46] (version 12). The injected code allows a developer to distinguish the following three states for each injected bug after fuzzing a target.

- *Not reached*: the fuzzer never reaches the conditional statement;
- *Reached*: some inputs generated by the fuzzer reach the conditional statement;
- *Triggered*: the reversed condition(s) with tracer is satisfied, while other condition(s) in the same statement is not satisfied.

Similar bug states are used in Magma [12].

We take output from the previous step of FIXREVERTER to perform an injection. For `ABORT` and `ASSIGN`, if there is a single guard expression, then the entire conditional and its body are skipped (it becomes `if (0)...`), mimicking the reversion of fixes as in Figure 2. For `ABORT`, if the guard expression is a disjunction (e.g., `if (p || q)...`) and just one of the subexpressions has a traced variable, then that subexpression is skipped (e.g., it becomes `if (p)...`); if all disjunctive clauses are traced, the entire `if` is skipped. For `EXEC`, the `if` body is retained, but the last-seen, relevant conditional expression is removed; e.g., `if (p && q)...` becomes `if (q)...`, as in Figure 3. When multiple injection candidates are syntactically nested, we inject the innermost candidate in the nested structure.

We use a static conditional preprocessing block to distinguish the injections when *running the fuzzer* and when *triaging* a crash (see Section 3.5), controlled by a macro, FRCOV. We can see this in Figure 7b. When the macro is undefined, the program can be used for fuzzing—in the running example, we have effectively replaced `if(oheader==NULL) continue` (lines 2-3 of Figure 7a) with `if(0) continue` (lines 12-14 of Figure 7b). This way no new branch is introduced so that it will not be biased towards fuzzers that are sensitive to control flows.

We define FRCOV to compile the programs used for triaging. This enables extra logic to log that a bug is *triggered* if the subexpression(s) with tracers is satisfied and other subexpression(s) is not satisfied. Otherwise, *reached* is logged. FIXREVERTER controls if an injection is turned on or off using an environment variable (which will cause `injectFlag[529]` to be true or false, in Figure 7b). Turning off an injection means that the original code logic at the injection site is preserved.

Before fuzzing a program, we compile it with options of address sanitizer (ASan) [56] and undefined behavior sanitizer(UBSan) [57]. This way, both temporal and spatial memory safety violations (out-of-bounds reads/writes, and uses-after-free) will reliably trigger a crash when they occur, in both the FRCOV-enabled/disabled versions of the program.

After bug injection, we perform the *naive bug filter*. As discussed in Section 1, we filter the uninteresting bugs that will fail too easily if injected.

3.5 Bug Triage

When a fuzzer generates an input that causes a FIXREVERTER-processed program to crash, we want to report the specific cause. This is not as simple as it might seem. Just because a condition is triggered does not mean that a dependent variable *must* then be dereferenced, only that it *could* be—the program may sometimes take an execution path that avoids that dereference. As such, running an input could trigger several injections, and a *triage* procedure must determine which ones are material in producing the crash.

Our triage procedure works by re-running the triage version of the target program (i.e., with the marcro FRCOV defined) on the generated input with subsets of the triggered bugs injected, to see under which circumstances the program still crashes. At the end, triage will identify two kinds of causes of failure:

- When input I produces a failure when *a single bug is injected*, call it A, then we say I triggering A is an *indi-*

Input: I: crashing input; ts: set of triggered injections
Output: bs: the set of I's failure causes

```
 1: bs ← ∅
 2: for i from 1 to ‖ts‖ do
 3:     for each set s ∈ Powerset(ts) where ‖s‖ = i do
 4:         if ∄s′ ∈ bs. s′ ⊂ s then
 5:             Run I on inject(s)
 6:             if I crashes then
 7:                 bs ← bs ∪ {s}
 8:             end if
 9:         end if
10:     end for
11: end for
```

Figure 8: Bug triage algorithm.

vidual cause. Even though other bug sites were triggered during the run, A is enough to cause the crash on its own.

- When input I produces a failure *only* when *multiple bugs are injected*, say A and B, then we say the failure cause is a *combination* of A and B. Neither A nor B individually is enough—both must present for the crash to occur.

Interestingly, it is possible that the same input can cause multiple individual- or combination-bug crashes. For example, a bug (or combination of bugs) in an initial stage of input processing may cause a crash on I, but if that bug(s) was/were removed, another bug (or combination) may cause a crash when the program processes a different part of I.

We use an algorithm to triage a crash into a set of causes. Intuitively, this algorithm first finds any individual causes, and then finds combination ones, if they exist. For example, if an input I triggers injections $ts = \{A,B,C,D\}$, a set $bs = \{\{A\},\{C,D\}\}$ produced by the algorithm means that $\{A\}$ is an individual cause (injecting only A causes the crash), and $\{C,D\}$ is a combination cause (injecting both C and D causes the crash but injecting them individually does not).

This algorithm is shown in Figure 8. It takes a crashing input I, and ts, the set of injections that were triggered when executing I. For each subset s in the powerset of ts, starting from the subsets whose size is 1 (lines 2-3), if s is not a superset of any element in the set bs (line 4), line 5 injects only the bugs in s and runs I. If I crashes, s is added as a cause to set bs (lines 6-7).

We report both individual and combination causes in our evaluation (see details in Section 5).

4 REVBUGBENCH

In this section, we discuss how we used FIXREVERTER to create REVBUGBENCH.

4.1 Target programs

REVBUGBENCH consists of 10 programs chosen from two sources. First, we use 8 programs from FuzzBench. FuzzBench currently consists of 47 real-world programs. We chose these 8 programs because the majority of their source code is written in C which our tool supports and the syntax matcher returns more than 100 candidate injection sites on these programs. Second, we chose two utilities, cxxfilt and disassemble, in the Binutils package. These utilities are frequently used in the literature to evaluate fuzz testing [9, 15, 15, 26, 33–36]. Column 1 in Table 1 shows the program names and column 2 shows the size (in MB) of each program (its program-specific LLVM bitcode). All 10 programs run from the libFuzzer [24] fuzzing entrypoints which allowed us to integrate REVBUGBENCH into the FuzzBench service (Section 4.3). The versions of these programs are the same as those used in FuzzBench if they are specified, or the most recent version at the time REVBUGBENCH was created.

4.2 FIXREVERTER usage

Columns 5, 8, and 11 in Table 1 show the number of injected bugs for each program, grouped by each bugfix pattern. In total, we injected over 7900 bugs; the `ABORT` pattern had the most injections (6742).

Syntax matcher We ran FIXREVERTER's syntax matcher on C source files in the target programs. First, for each program, we use Bear [64] to generate a compilation database that records compile commands needed to build C files. The syntax matcher is then run on the source files in the compilation database; other C source files not in the compilation database are omitted.

Static reachability & dependence analysis To run the static analysis requires specifying a set of program entrypoints. Because every program in REVBUGBENCH runs from a libFuzzer harness, we use each program's LLVMTestOneInput function as its entry point. Our implementation based on Phasar release v0521 experienced out of memory crashes on disassemble, curl, libxml2_reader, libxml2_xml and usrsctp. This Phasar release improved from earlier versions the soundness of its dataflow analysis, but also introduced performance overhead. For these five program, we used an earlier version, Phasar v1220 [65], to avoid the memory error. On average, static analysis dropped 71% of the injection sites returned by the syntax matcher.

Naive bug filter For each target program, we inject each potential injection site returned by the static analysis and execute the program with its fuzzing seeds. For each REVBUGBENCH program, we use the seeds provided by its maintainers if available (curl, libpcap, libxml2_reader, proj4, usrsctp, and zstd). Otherwise, we use a default seed "hi" (consistent with FuzzBench's practice of seed selection) and a small valid seed

Table 1: REVBUGBENCH. The *syntax*, *semantic*, and *final* columns show the numbers of injection sites returned by the syntax matcher, static analysis, and naive bug filter, respectively.

Program	Size (MB bitcode)	Injection sites								
		ABORT			EXEC			ASSIGN		
		# syntax	# semantic	# final	# syntax	# semantic	# final	# syntax	# semantic	# final
cxxfilt [27]	40	5784	88	86	360	1	1	1404	7	7
disassemble [27]	67	5784	1263	1262	360	56	55	1404	181	181
curl [58]	22	304	159	155	13	1	1	57	21	19
lcms [59]	2	587	365	360	15	6	5	40	11	10
libpcap [60]	1	252	92	87	11	0	0	19	8	7
libxml2_reader [29]	8	3992	2142	2122	186	112	112	564	340	335
libxml2_xml [29]	8	4442	1624	1615	206	91	91	594	285	278
proj4 [61]	3	246	223	222	10	2	2	21	8	8
usrsctp [62]	5	732	580	572	51	36	35	14	7	7
zstd [63]	5	488	285	261	5	0	0	60	18	14
Total	161	22611	6821	6742	1217	305	302	4177	886	866

taken from AFL's repository based on file format. We drop from consideration any injections that crash any seed. The naive bug filter dropped 102 injections.

4.3 FuzzBench Service Integration

FuzzBench provides service to evaluate fuzzers on real-world benchmarks, at scale [3]. It has integrated multiple state-of-the-art fuzzers and provides build scripts that make it easier to replicate fuzzing evaluations. However, FuzzBench focuses on comparing fuzzers via code coverage. While it also supports measuring fuzzer performance via bugs found, its method to distinguish bugs (stack traces) is unreliable, as discussed in Section 1. We integrated REVBUGBENCH into FuzzBench to allow replicable and large-scale evaluations of fuzzers that measure their ability to find crashing bugs in real-world programs.

We extended three key components of FuzzBench: *benchmarks* (target programs), *measurer* (on-the-fly result analyzer), and *reporter* (statistical analysis of results).

Each program in FuzzBench is automatically compiled on a Docker image [66], so called base-image, with a build script. We build FIXREVERTER in the base-image of each target program and modify its build script to run FIXREVERTER. Specifically, we generate the compilation database to run the syntax matcher, create the intermediate representation to perform the static analysis, and then inject the bugs. The bug-injected programs are given to FuzzBench for fuzzing (i.e., added to its *benchmarks* component). The programs are compiled with options of address sanitizer (ASan) [56] and undefined behavior sanitizer (UBSan) [57] set up by FuzzBench.

FuzzBench's *measurer* runs inputs generated by fuzzers on target programs, and measures some characteristics of the execution such as code coverage. FuzzBench packs inputs generated every 15 minutes into a cycle. The *measurer* is invoked in every cycle to process the inputs. We extended the measurer with our bug triage algorithm (Section 3.4) to measure which injections (or their combination) are reached, triggered and/or crashed. We also extended the measurer to store the collected results in a database for further analysis after each fuzzing campaign completes. We implemented such an analysis to speed up the bug triage with multi-processing.

FuzzBench's *reporter* performs statistical analysis on data collected from each fuzzer, such as how code coverage grows over time. We reuse its interface to run the same analysis to report the bugs found by each fuzzer over time.

5 Experiments

This section presents the experimental results of running five different fuzzers on REVBUGBENCH, to assess its utility as a fuzzing evaluation benchmark, and FIXREVERTER's effectiveness at bug injection. Our assessment considers the following three questions:

(1) Does FIXREVERTER inject bugs that can trigger actual crashes (Section 5.1)? FIXREVERTER's static analysis is conservative, so it could retain injections that it thinks *may* lead to a crash, but do not in actual fact. The experiments show that hundreds of injections are triggered in REVBUGBENCH, and many of these lead to crashes.

(2) Does FIXREVERTER inject bugs that are hard to find, at least for some fuzzers (Section 5.2)? If REVBUGBENCH only contains bugs that all fuzzers can find, it fails to distinguish one fuzzer from another. The experiments show that only a fraction of the injected bugs trigger crashes (i.e., they are not too easy); some bugs are only triggered by some fuzzers; and fuzzer performance varies overall.

(3) Do fuzzers trigger crashes owing to *combination* causes (see Section 3.5), i.e., where multiple bugs must be present for the crash to occur (Section 5.3)? Triggering an injected bug (or bugs) may not lead to a crash, on its own, but doing so may perturb control flow toward another bug that triggers one. The experiments reveal many combination causes, which constitute an additional, interesting performance measure. In most cases a fuzzer's performance is consistent when measuring either individual causes or combination causes.

Experimental setup The five fuzzers we ran are AFL [1], libFuzzer [24], AFL++ [25], Eclipser [16], and FairFuzz [26].

Table 2: Fuzzer performance on REVBUGBENCH. *Reach*, *Trigger*, *Individual*, and *All Causes* columns show median numbers of reached injections, triggered injections, individual causes, and distinct injections that caused any crashes (individual or combination). Each cell first shows the results of all patterns, and then *ABORT*, *EXEC* and *ASSIGN* in the parentheses.

Program	AFL [1]				AFL++ [25]			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	49 (46 0 3)	35 (33 0 2)	8 (8 0 0)	26 (26 0 0)	51 (48 0 3)	36 (34 0 2)	9 (9 0 0)	27 (27 0 0)
disassemble	68 (61 2 5)	14 (8 2 4)	11 (8 2 1)	11 (8 2 1)	110 (103 2 5)	18 (12 2 4)	14 (11 2 1)	14 (11 2 1)
curl	59 (46 0 13)	25 (20 0 5)	6 (6 0 0)	10 (10 0 0)	58 (46 0 12)	20 (17 0 4)	5 (5 0 0)	6 (6 0 0)
lcms	139 (131 4 4)	26 (24 0 2)	7 (7 0 0)	20 (18 0 2)	152 (143 4 5)	33 (31 0 3)	11 (11 0 0)	27 (25 0 2)
libpcap	31 (30 0 1)	14 (13 0 1)	3 (3 0 0)	9 (8 0 1)	43 (42 0 1)	19 (18 0 1)	4 (4 0 0)	10 (9 0 1)
libxml2_reader	413 (327 18 68)	108 (48 11 51)	19 (10 4 6)	47 (22 7 19)	482 (374 22 86)	141 (61 13 70)	23 (13 4 6)	66 (35 8 23)
libxml2_xml	257 (205 7 45)	65 (25 4 37)	8 (5 2 1)	38 (13 3 23)	304 (250 7 47)	83 (38 4 41)	10 (6 1 3)	29 (17 2 10)
proj4	78 (72 1 5)	35 (33 0 2)	18 (16 0 2)	20 (18 0 2)	174 (167 1 6)	74 (72 0 2)	31 (29 0 2)	32 (30 0 2)
usrsctp	226 (214 11 1)	15 (12 2 0)	6 (4 1 0)	7 (6 1 0)	238 (224 13 1)	20 (17 3 0)	8 (6 2 0)	14 (12 2 0)
zstd	152 (147 0 5)	100 (95 0 5)	34(34 0 0)	71 (70 0 0)	151(146 0 5)	101 (96 0 5)	31 (31 0 0)	63 (63 0 0)
Total	1472 (1279 43 150)	437 (311 19 109)	120 (101 9 10)	259 (199 13 48)	1763 (1543 49 171)	545 (396 22 132)	146 (125 9 12)	288 (235 14 39)

Program	Eclipser [16]				FairFuzz [26]			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	50 (47 0 3)	37 (35 0 2)	9 (9 0 0)	28 (28 0 0)	44 (42 0 2)	31 (30 0 1)	7 (7 0 0)	24 (24 0 0)
disassemble	73 (66 2 5)	14 (8 2 4)	11 (8 2 1)	11 (8 2 1)	34 (27 2 5)	13 (9 2 4)	11 (9 2 1)	11 (9 2 1)
curl	59 (46 0 13)	28 (23 0 5)	6 (6 0 0)	9 (9 0 0)	58 (46 0 12)	26 (22 0 3)	6 (6 0 0)	8 (8 0 0)
lcms	144 (135 4 4)	24 (22 0 2)	8 (7 0 0)	18 (17 0 1)	143 (135 4 4)	26 (23 0 2)	9 (7 0 0)	17 (16 0 1)
libpcap	28 (27 0 1)	13 (12 0 1)	3 (3 0 0)	8 (7 0 1)	17 (16 0 1)	7 (6 0 1)	1 (1 0 0)	3 (3 0 0)
libxml2_reader	411 (325 17 68)	106 (44 10 52)	18 (10 3 5)	49 (23 6 20)	375 (298 17 60)	98 (42 9 46)	15 (8 2 3)	42 (21 5 16)
libxml2_xml	262 (210 7 45)	67 (25 4 38)	7 (4 0 1)	31 (12 3 17)	269 (217 6 46)	72 (29 4 39)	5 (4 0 1)	20 (11 1 8)
proj4	199 (192 1 6)	81 (79 0 2)	33 (31 0 2)	33 (31 0 2)	75 (69 1 5)	33 (31 0 2)	17 (15 0 2)	19 (17 0 2)
usrsctp	218 (206 11 1)	15 (13 2 0)	7 (6 1 0)	9 (8 1 0)	192 (180 11 1)	10 (8 2 0)	3 (2 1 0)	3 (2 1 0)
zstd	151 (146 0 5)	101 (96 0 5)	37 (37 0 0)	70 (70 0 0)	149 (144 0 5)	93 (88 0 5)	34 (34 0 0)	61 (61 0 0)
Total	1595 (1400 42 151)	486 (357 18 111)	139 (121 6 9)	266 (213 12 42)	1356 (1174 41 141)	409 (288 17 103)	108 (93 5 7)	208 (172 9 28)

Program	LibFuzzer [24]				MetaFuzzer			
	Reach	Trigger	Individual	All Causes	Reach	Trigger	Individual	All Causes
cxxfilt	24 (24 0 1)	16 (15 0 1)	3 (3 0 0)	14 (14 0 0)	53 (50 0 3)	42 (39 0 3)	11 (11 0 0)	35 (35 0 0)
disassemble	38 (28 2 8)	16 (9 2 6)	11 (8 2 2)	12 (9 2 2)	139 (129 2 8)	22 (14 2 6)	16 (12 2 2)	17 (13 2 2)
curl	49 (40 0 9)	19 (16 0 3)	4 (4 0 0)	6 (6 0 0)	59 (46 0 13)	31 (26 0 5)	7 (7 0 0)	13 (12 0 1)
lcms	135 (128 4 3)	25 (24 0 1)	13 (13 0 0)	22 (21 0 1)	161 (152 4 5)	41 (38 0 3)	21 (19 0 2)	33 (31 0 2)
libpcap	30 (29 0 1)	12 (11 0 1)	3 (3 0 0)	9 (8 0 1)	48 (47 0 1)	24 (23 0 1)	7 (7 0 0)	16 (15 0 1)
libxml2_reader	396 (314 18 64)	92 (36 9 47)	16 (9 3 3)	64 (25 8 31)	545 (422 26 97)	175 (79 15 81)	33 (16 4 13)	109 (49 11 49)
libxml2_xml	262 (212 6 44)	67 (29 4 34)	4 (4 0 0)	15 (10 1 5)	317 (258 7 52)	90 (40 4 46)	32 (10 3 19)	75 (33 3 39)
proj4	163 (158 0 5)	73 (72 0 1)	30 (29 0 1)	31 (30 0 1)	204 (197 1 6)	86 (83 0 3)	36 (34 0 2)	38 (36 0 2)
usrsctp	199 (187 11 1)	11 (9 2 0)	3 (2 1 0)	7 (6 1 0)	249 (235 13 1)	20 (17 3 0)	9 (7 2 0)	15 (12 3 0)
zstd	140 (135 0 5)	79 (74 0 5)	28 (28 0 0)	52 (52 0 0)	152(147 0 5)	107(102 0 5)	47 (47 0 0)	89 (88 0 1)
Total	1436 (1255 41 141)	410 (295 17 99)	115 (103 6 6)	232 (181 12 41)	1927 (1683 53 191)	638 (461 24 153)	219 (170 11 38)	440 (324 19 97)

AFL and libFuzzer are both classic and popular fuzzers. AFL++, Eclipser, and FairFuzz were all recently developed. AFL++ is the best fuzzer using FuzzBench's coverage metric in a sample report [67]. Eclipser's evaluation shows that it significantly outperforms other fuzzers in terms of bugs found on LAVA-M [16]. FairFuzz is the most referenced recent fuzzer that has been integrated into FuzzBench.

Each fuzzer was run on each benchmark program with 3

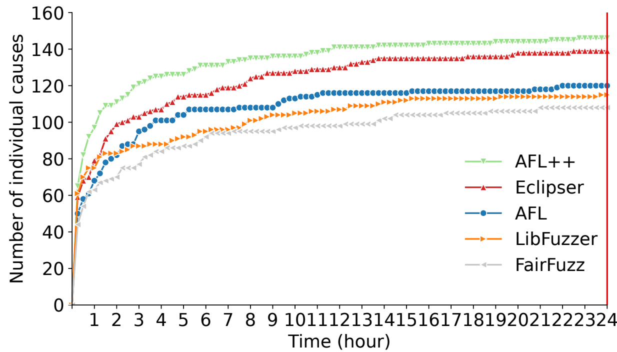

Figure 9: Individual causes detected over time.

trials and 24-hour timeout. We ran bug triage for all crashing inputs generated by each fuzzer, except for LibFuzzer's results on libxml2_reader and libxml2_xml. These two runs generated orders of magnitude more crashing inputs than other runs (over 150,000 compared to a few hundred), making it infeasible to triage all of them. We thus took a random sample of 664 crashing inputs, which ensures the results are within a 5% margin of error and 1% confidence level. We also limited the triage of each combination cause to include at most 3 injections. Bug triage took between 3 and 240 minutes for each fuzzer on each benchmark program.

The experiments were run on 2 servers. Server A has 2 Intel(R) Xeon(R) Silver 4116 CPUs with 192GB RAM running Ubuntu 16.04. Server B has 2 Intel(R) Xeon(R) Gold 5218 CPUs with 384GB RAM running Ubuntu 18.04. All experiments of each target program were conducted on the same server to ensure that the fuzzer performance can be compared.

5.1 Does FIXREVERTER inject bugs that fuzzers can actually find?

Table 2 tabulates overall fuzzer performance. Each row is a REVBUGBENCH program, and each group of columns identifies a particular fuzzer. Fuzzers that detected the most individual causes for a program are highlighted in gray. The **MetaFuzzer** columns show the aggregated results over *all trials of all fuzzers in each program*, an upper bound when comparing fuzzers. We use this combined performance measure to illustrate the properties of REVBUGBENCH. Overall, using the number of individual causes as the performance metric, the MetaFuzzer found a total of 219 individual causes in REVBUGBENCH, out of thousands of possible injections (Table 1); most of these individual causes (170) were due to the injected bugs that reversed the *ABORT* pattern.

AFL++ was the best performing fuzzer, detecting the most individual causes overall (146), and the most in 6 programs (tied with Eclipser in 1 program). Eclipser was a close second, finding 139 individual causes, and the most in 4 programs (tied with other fuzzers in 1 program). While AFL, FairFuzz, and LibFuzzer found fewer individual causes, each of them has 1 program in which it detected the most individual causes.

Table 2 shows fuzzer performance by the end of the 24-hour timeout; Figure 9 shows the median number of individual causes detected over time, for each fuzzer for all programs. Many bugs were found quickly, with a long tail. All fuzzers were able to detect some individual causes even after 12 hours of the fuzzing campaign. This result illustrates the challenge of detecting some injected bugs in REVBUGBENCH. AFL++ consistently performed better than other fuzzers over time. On the other hand, although libFuzzer started by finding many individual causes (75) in the first 45 minutes, it was eventually surpassed by AFL and Eclipser over 24 hours.

5.2 Does FIXREVERTER inject bugs that are hard to find?

Overall, fuzzers found 219 distinct individual bugs during our experiments, out of thousands injected. We do not know how many of those injected *can* lead to a crash (due to the overapproximate nature of the static analysis), but our experiments show that a sufficient number can be used to distinguish different fuzzers' performance.

Indeed, with more time and trials, we expect more will be detected. For example, we observed that different bugs were detected by the same fuzzer in different trials: although LibFuzzer found the most individual causes in lcms (13); in total there were 21 individual causes found in all trials by all 5 fuzzers (captured in the MetaFuzzer result). Moreover, we can see that Table 2's *Reach* and *Trigger* columns, which count the number of injection sites' reverted conditions that fuzzers reached and triggered, respectively, are a fair bit higher than the number of causes. 1927 (24%) and 638 (8%) of all 7910 injected bugs were reached and triggered, respectively. We expect that for at least some of these, different inputs would drive the program from the trigger to a crash.

The Venn diagram in Figure 10 examines which fuzzers found which bugs (in any trial). Over 40% (91 out of 219) individual bugs were detected by all fuzzers, and 16% (35) bugs were detected by four fuzzers. This shows a large number of FIXREVERTER injected bugs can be found by most fuzzers. On the other hand, each fuzzer detected unique bugs that other fuzzers did not find. Overall, about 19% (42) of all individual bugs were only found by 1 fuzzer. FairFuzz, the fuzzer that had the worst performance in terms of detecting individual causes, found 2 unique bugs. AFL++, as expected, found the most unique bugs (28). This result also shows that FIXREVERTER injected bugs that do not overfit a single approach in the evaluated fuzzers.

5.3 Do fuzzers find combination causes in REVBUGBENCH?

Our idea in developing REVBUGBENCH was to assess a fuzzer's performance according to individual bugs found.

Figure 10: Venn diagram of all individual causes.

However, we discovered via our triage procedure (Section 3.5) that many crashes only occurred if more than one bug is injected (i.e., the combination cause); these are tabulated (summed with individual causes) in the *All Causes* columns in Table 2.

For a crash with an individual cause, a reverted condition is triggered and subsequent execution dereferences a dependent variable that the check would have prevented, producing a crash. But not every individual trigger necessarily leads to a crash, as discussed above; additional criteria may also need to be met which are outside the triggered condition. When multiple bugs are injected, it is possible that these criteria could be met by triggering another bug (or bugs). For example, suppose that for a particular input I, triggering bug A violates an invariant about a data structure, but then triggering bug B drives control flow to a point that accesses that data structure. Both bugs are needed to cause the crash, for I.

When assessing fuzzer performance, counting individual bugs is simplest, but doing so fails to account for bug interactions observed, and potentially exploited, by fuzzers during a campaign. As shown in Table 2, combination causes were detected in all programs; sometimes individual causes were more prevalent (e.g, disassemble and proj4), and sometimes combination ones were (e.g., libxml2_reader and cxxfilt). Nevertheless, relative performance among fuzzers is fairly consistent whether using “individual causes" and “all causes" as metrics. AFL++ was the best fuzzer using both metrics. Overall, fuzzers performed consistently on the metrics of reach, trigger and cause. A better fuzzer (e.g., AFL++) tends to perform better using all three metrics on each target program.

6 Related Work

As discussed in Section 1.1, prior fuzzing benchmarks have limitations that FIXREVERTER can help address. Google fuzzer-test-suite [68], recently superseded by FuzzBench [3], and UNIFUZZ [10] are platforms for evaluating fuzz testers at scale, but neither provides a distinctive indication of when a bug is triggered; doing so has been shown to yield highly unreliable results [9]. FIXREVERTER's automatically injected bugs produce crashes that can be definitively characterized, using our novel triage algorithm.

Magma [12] and DARPA Cyber Grand Challenge (CGC) [11] comprise programs with manually injected bugs that give a telltale sign when triggered. Magma's programs are real, and the injected bugs are based on a variety of actual prior bugs; CGC's programs and bugs are synthetic, with many small programs and one bug per program. FIXREVERTER's injected bugs are realistic, based on the reversal of common bugfix patterns we found in a corpus of CVEs.

LAVA is another bug injection framework for benchmarking fuzz testing [13]. It relies on a dynamic taint analysis to identify program locations where some input bytes `b` are able to influence program behavior. Near these program locations, LAVA injects out-of-bounds accesses predicated on comparisons to “magic values," e.g., `b==0x6c617661`. There is concern that the injected bugs in LAVA-M [13] are biased, as noted by LAVA authors [20] and Magma's evaluation [12]. Such bias was not observed in our evaluation on REVBUGBENCH.

Apocalypse [19] is a bug injection methodology that adds code implementing an *error transition system* (ETS) along a judiciously chosen program path; when the final state is reached, a fault is induced. The current state of ETS is kept in an injected global variable, and transitions are guarded by predicates over program variables; such predicates are synthesized from symbolic execution path conditions (for feasibility) and model counts (for rarity). Apocalypse was used to inject bugs in 30 Coreutils programs, and it was evaluated using KLEE [69] and AFL [1]. One nice feature of Apocalypse is that, like LAVA, injected bugs are always triggerable—they are injected along what is known to be a feasible path. Having to witness such a path in advance, however, is arguably a limitation, and one that FIXREVERTER does not have. Also like LAVA, Apocalypse-injected bugs are synthetic, whereas FIXREVERTER-injected bugs are more realistic. We view Apocalypse and FIXREVERTER as complementary, and indeed both could be used to inject bugs in the same benchmark.

EvilCoder [21] is also a vulnerability injection methodology that relies on static dataflow analysis. It identifies *sensitive sinks*, such as calls to `system()`, and user-controlled sources, such as returns from `fread()`, and aims to discover dataflow paths from the former to the latter. To inject a bug, it tries to find “security mechanisms” that are used to affect variables

that carry data along the flow, e.g., by sanitizing them, and then "instruments" the mechanism so as to bypass it. While EvilCoder's source-sink discovery component has been implemented, its bug injection component seems to not have been automated, and just a proof-of-concept example was developed. As such, the utility of the approach was never fully evaluated. Different from EvilCoder, FIXREVERTER focuses on injecting vulnerabilities based on bugfix patterns that are defined both syntactically and semantically. Dataflow analysis in FIXREVERTER (i.e., the static reachability and dependence analysis) can also be viewed as a static taint analysis, but its primary goal is to match the semantic pattern, as described in Section 3.3. We conjecture that FIXREVERTER could be extended to implement EvilCoder-style bugs.

FIXREVERTER is a kind of fault injector, an antithesis to tools that aim to automatically fix faults. While no past work has used bugfix patterns to introduce triggerable bugs for the purpose of benchmarking fuzz testing as we present in this paper, work on automatic program repair has seen successful applications of bugfix patterns [70]. For example, Getafix [41] applies a hierarchical clustering algorithm that summarizes fix patterns into a hierarchy and a ranking technique that uses the context of a code change to select the most appropriate fix for a given bug. Getafix shows that real-world bugfixes often use patterns similar to what we revert in FIXREVERTER, further evidencing the feasibility of using bugfix patterns to (re-)introduce realistic bugs. The bugfix patterns discovered in automatic program repair and other works that study the bugfix repositories (e.g., [38, 71, 72]) may be adapted to allow FIXREVERTER to support other bug injections.

7 Conclusions and Future Work

This paper has presented FIXREVERTER, a novel bug injection framework at the heart of a fuzzing benchmark-producing methodology. It aims to inject realistic bugs that give a unique indication when triggered. It does so by finding, through syntactic matching and static analysis, code patterns that match fixes of previous CVEs; reversing those patterns should (re)introduce self-signaling bugs. FIXREVERTER-based benchmarks can evolve, since FIXREVERTER can be used to inject bugs into new programs and with new patterns. Doing so ensures the benchmark is relevant and fresh, so tools do not overfit to it. FIXREVERTER serves as a useful complement to frameworks such as Google's FuzzBench [3].

We constructed REVBUGBENCH by using FIXREVERTER on 8 FuzzBench and 2 Binutils programs that are common targets of fuzzing tools. We evaluated 5 fuzzers using REVBUGBENCH and its built-in performance metrics. We observed that FIXREVERTER injected many triggerable bugs, and these fuzzers detected very different bugs in REVBUGBENCH. Overall, the fuzzer performance varied by target program, with AFL++ doing the best.

We see two important future directions. First, FIXREVERTER's static analysis has several shortcomings that, if fixed, should improve the quality of its results. In particular, sound handling of function pointers would make the static analysis more useful. Second, FIXREVERTER's bugfix patterns can be extended according to further study of existing fixes in bug databases [38, 41]. We plan to include patterns that do not involve changes to the control flow as in the three current patterns but require new definitions of the semantic conditions.

Acknowledgements

This work was partly supported by NSF grants CCF-1816951 and CCF-1955610. We would also like to thank Google's FuzzBench team for their help on our experiments and the anonymous reviewers for their detailed comments that improved the paper.

References

[1] "american fuzzy lop," https://lcamtuf.coredump.cx/afl/.

[2] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[3] "Fuzzbench: Fuzzer benchmarking as a service," https://google.github.io/fuzzbench/.

[4] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering (ICSE)*, 2014.

[5] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[6] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *International Conference on Software Engineering (ICSE)*, 2014.

[7] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.

[8] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Security Symposium*, 2009.

[9] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[10] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," 2020.

[11] "Darpa cyber grand challenge (cgc) binaries," https://github.com/CyberGrandChallenge/, 2018.

[12] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[13] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: large-scale automated vulnerability addition," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[14] "Rode0day: A continuous bug finding competition," https://rode0day.mit.edu/.

[15] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[16] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 736–747. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00082

[17] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 627–637. [Online]. Available: https://doi.org/10.1145/3106237.3106295

[18] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[19] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 224–234.

[20] B. Dolan-Gavitt, "Of bugs and baselines," http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html, 2018.

[21] J. Pewny and T. Holz, "EvilCoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 214–225. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991103

[22] "Common vulnerabilities and exposures (cve)," https://cve.mitre.org/.

[23] "CVE-2017-8395." Available from MITRE, CVE-ID CVE-2017-8395., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8395

[24] "Libfuzzer," https://llvm.org/docs/LibFuzzer.html.

[25] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[26] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

[27] "Gnu binutils," https://github.com/google/oss-fuzz/tree/master/projects/binutils.

[28] "Tcpdump 4.x.y by the tcpdump group," https://github.com/the-tcpdump-group/tcpdump.

[29] "The xml c parser and toolkit of gnome," https://github.com/google/fuzzbench/tree/master/benchmarks/libxml2_libxml2_xml_reader_for_file_fuzzer.

[30] "A complete, cross-platform solution to record, convert and stream audio and video." https://github.com/FFmpeg/FFmpeg.

[31] "libarchive - multi-format archive and compression library," https://github.com/libarchive/libarchive.

[32] "The systemd system and service manager," https://github.com/systemd/systemd.

[33] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[34] B. Zhang, J. Ye, C. Feng, and C. Tang, "S2f: Discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, 2017, pp. 548–552.

[35] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2329–2344. [Online]. Available: https://doi.org/10.1145/3133956.3134020

[36] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 803–817. [Online]. Available: https://doi.org/10.1109/SP.2019.00052

[37] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 787–802.

[38] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 339–349. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00048

[39] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Trans. Software Eng.*, vol. 47, no. 1, pp. 165–188, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2018.2884955

[40] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni, "Learning quick fixes from code repositories," *CoRR*, vol. abs/1803.03806, 2018. [Online]. Available: http://arxiv.org/abs/1803.03806

[41] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585

[42] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 802–811.

[43] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, *TBar: Revisiting Template-Based Automated Program Repair*. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[44] "CVE-2017-8392." Available from MITRE, CVE-ID CVE-2017-8392., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8392

[45] "CVE-2013-0211." Available from MITRE, CVE-ID CVE-2013-0211., 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0211

[46] "Libtooling," https://releases.llvm.org/8.0.0/tools/clang/docs/LibTooling.html.

[47] S. G. McPeak, "Elkhound: A fast, practical glr parser generator," USA, Tech. Rep., 2003.

[48] "CVE-2017-7303." Available from MITRE, CVE-ID CVE-2017-7303., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7303

[49] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[50] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[51] "Phasar v0521: a llvm-based static analysis framework," https://github.com/secure-software-engineering/phasar/tree/aed66c04e6dbb3f6ef1bed4ad69b29aa0017bd9a.

[52] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[53] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, p. 1–69, Apr. 2015. [Online]. Available: https://doi.org/10.1561/2500000014

[54] "Whole program llvm," https://github.com/travitch/whole-program-llvm/.

[55] "Source level debugging with llvm," https://releases.llvm.org/10.0.0/docs/SourceLevelDebugging.html.

[56] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[57] "Undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[58] "Quality assurance testing for the curl project," https://github.com/curl/curl-fuzzer.

[59] "Little-cms," https://github.com/google/fuzzbench/tree/master/benchmarks/lcms-2017-03-21.

[60] "A portable c/c++ library for network traffic capture," https://github.com/google/fuzzbench/tree/master/benchmarks/libpcap_fuzz_both.

[61] "Proj - cartographic projections and coordinate transformations library," https://github.com/OSGeo/PROJ.

[62] "Sctp user-land implementation," https://github.com/weinrank/usrsctp.

[63] "Zstandard - fast real-time compression algorithm," https://github.com/facebook/zstd.

[64] "Build ear," https://github.com/rizsotto/Bear.

[65] "Phasar v1220: a llvm-based static analysis framework," https://github.com/secure-software-engineering/phasar/tree/febddffe9e2ca98b4587e3ed4298dd02c1adda0e.

[66] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[67] "Fuzzbench report," https://google.github.io/fuzzbench/reference/report/.

[68] "Fuzzer test suite," https://github.com/google/fuzzer-test-suite/.

[69] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[70] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[71] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 913–923.

[72] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.