FIXREVERTER: A Realistic Bug Injection Framework for Benchmarking Fuzz Testing

Anonymous Author(s)

***Abstract*—Fuzz testing is an active area of research with proposed improvements published at a rapid pace. Such proposals are judged *empirically*: Can they be shown to perform better than the status quo? Unfortunately, recent work has found that fuzz testing evaluations, broadly speaking, often fall short of best scientific practice, resulting in misleading or incorrect results. There is a real need for a solid fuzz testing benchmark.**

This paper makes three contributions on this front.

- **First, it identifies four goals for such a benchmark: (1) it should have relevant, real-world target programs; (2) these programs should have realistic, relevant bugs; (3) when a bug is triggered by a fuzzer it should be evident which one; and (4) the benchmark should not be prone to overfitting.**
- **Second, it presents FIXREVERTER, a bug injection framework that can be used to produce a compliant benchmark. Our key idea is to automatically (re)introduce bugs in a program: FIXREVERTER identifies code patterns that match fixes of previous CVEs, and reverses those patterns. FIXREVERTER is built with a series of steps, involving static and dynamic analyses, which increase the confidence that reversing a fix will result in interesting, triggerable fault.**
- **Last, it presents BUGBIN, a benchmark built by using FIXREVERTER to inject bugs in existing binutils programs (a common target of fuzzing evaluations).**

Experiments demonstrate that FIXREVERTER's individual components are useful, and that overall it is capable of injecting realistic bugs useful for fuzz testing evaluation.

I. INTRODUCTION

Fuzz testing (a.k.a. *fuzzing*) has proved to be surprisingly successful at discovering security vulnerabilities. For example, AFL, one of the most mature and widely used fuzzers, has an extensive trophy case [1]. 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/paths in a target program than the baseline B, then one could argue A will find more bugs. However, while studies show that greater test suite code coverage does seem to correlate with finding more bugs [4], [5], the correlation is weak [6]. Another popular measure is to count the number of distinct, crash-inducing tests 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 [7]. Unfortunately, a study by Klees et al [8] showed that both heuristics could still yield very 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.

A. Developing a Fuzzing Benchmark

Klees et al recommended developing a *benchmark* of buggy programs with which to empirically compare fuzzer implementations. 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);
- **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.

To the best of our knowledge, there exist five fuzzing benchmarks: LAVA-M [9], the DARPA Cyber Grand Challenge (CGC) [10], Google fuzzer-test-suite [11], UNIFUZZ [12], and Magma [13]. LAVA-M is a set of four small programs with bugs injected automatically using a technique called LAVA. While LAVA-M has proved a popular target in the fuzzing literature, it arguably fails goals **G2** and **G4**. In particular, LAVA injects "bugs" that are triggered just by exercising a branch that contains an unusual condition. Such patterns do not match realistic bugs, and they are easily gamed, as one of the LAVA authors observed [14]. CGC has more programs, and its bugs are well-identified (**G3**) and more realistic (**G2**). However, its programs are synthetic (violating **G1**).

More recently, Hazimeh et al. [13] have proposed Magma, which comprises a set of real-world programs with relevant bugs injected by hand; when a bug is triggered, it gives a telltale sign. Magma satisfies goals **G1**, **G2**, and **G3**. Its main issue is that it is *fixed*. If a benchmark comes into common use, tools may start to employ heuristics and strategies that are not fundamental, but apply disproportionately to particular benchmark programs [8]. While fuzzing tools are fast evolving, benchmarks need to evolve, too. Magma (and CGC, too) have manually injected bugs; the significant effort to update them may result in stagnation and overfitting.

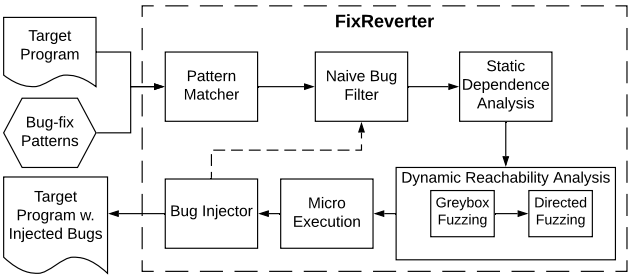

Fig. 1: The architecture of FIXREVERTER.

Google fuzzer-test-suite and UNIFUZZ both contain dozens of real-world buggy programs, satisfying **G1** and **G2**. But both benchmarks do not provide clear indication when a bug is found, using inaccurate deduplication heuristics (violating **G3**) and require significant manual efforts to update (violating **G4**).

In sum, no existing benchmark satisfies all four goals.

B. *Our proposal:* FIXREVERTER

This paper presents FIXREVERTER, a tool at the heart of a fuzzing benchmark-producing methodology, and BUGBIN, an initial benchmark we have produced using it. FIXREVERTER is a novel *fault injection* tool. Our observation is that a benchmark built around fault injection is useful because (1) bugs can be injected into real programs (**G1**) in a way that the bug signals when triggered (**G3**), 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 (**G4**). On the other hand, this approach only works if *realistic* faults are injected and if the methodology for doing so is not easy to game (goals **G2** and **G4**). These are the problems with LAVA, and ones we attempt to address with FIXREVERTER.

To inject bugs that are realistic, our key idea is to employ code patterns that match fixes of previous Common Vulnerabilities and Exposures (CVEs). 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 illustrates the architecture of FIXREVERTER. It takes as inputs a program into which to inject bugs, and descriptions of bugfix patterns. Its first step is to match the *syntax* of the pattern in the target program according to a grammar, thus identifying candidate injection sites (Section II). For example, a fix pattern FIXREVERTER supports is a simple NULL check that immediately returns, which is based on fixes like the one for CVE-2017-8395 [15], shown in Figure 2 (under the comment *PR 21431*). Many patterns are possible. Based on a study of CVE fixes, we also have explored patterns involving pointer range checks and numeric checks, which are used to fix out-of-bounds errors.

For each candidate injection site that matches this code pattern, FIXREVERTER carries out a series of steps to increase confidence that reversing the fix (in this case, deleting the

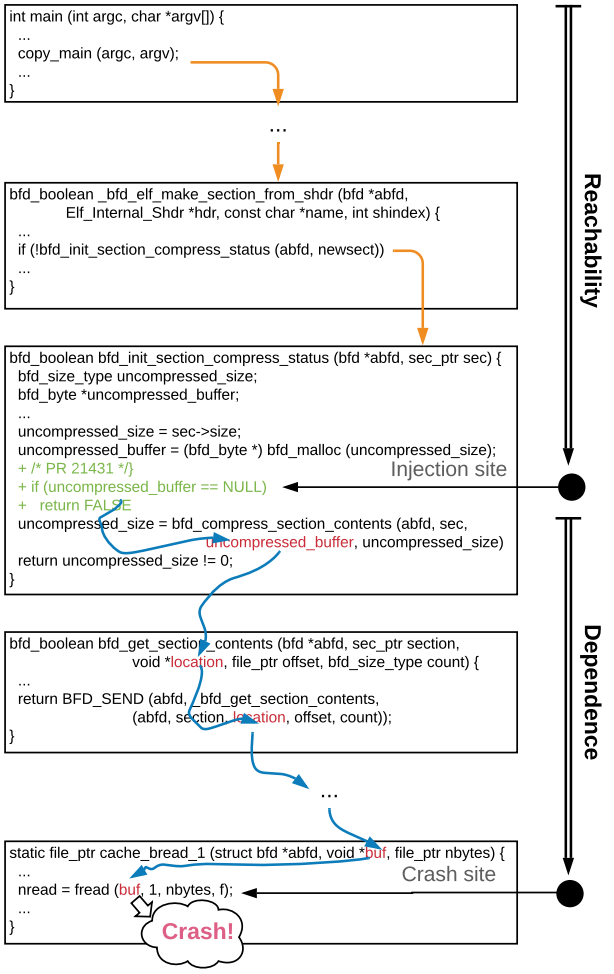

Fig. 2: Illustration of FIXREVERTER: CVE-2017-8395

check and return) will indeed result in an interesting, triggerable crash. The first of these steps is the *naive bug filter*. An injected bug could fail too easily, in which case we should not include it. One way to drop such bugs is to inject each candidate and then run a regression test suite—any injected bug that fails the regression tests should be discarded. Another strategy, which we used to develop BUGBIN, is to run the program with an injected bug using a default input, and if it fails, we don't inject.

After filtering naive bug injections, the next step is *static dependence analysis* (Section III). It serves two purposes. First, it determines that the injection site (i.e., NULL-check site in Figure 2) is *reachable* via a feasible program execution starting at main. Second, it (inter-procedurally) analyzes dataflow from a checked variable at that site (uncompressed_buffer in the example) to a possible dereference site (the fread call in the example); if such a *dependence* is present, the chances of a crash are much more likely.

The static analysis should winnow the candidate sites to those that can induce real crashes, but static analysis is over-

approximate and thus may leave some non-bug sites behind. To winnow these, FIXREVERTER performs *dynamic reachability analysis* (Section IV-A). This phase employs a run-time analysis to see whether a particular injection site can be reached by an *actual execution* from main. We can use any of various greybox and/or directed fuzzers (e.g., AFL [1], Angora [16], AFLGo [17]) to decide this question; using multiple tools can help overcome limitations of any single one.

Even if an injection can be reached, that does not mean it will trigger a crash. To add confidence that it will, FIXREVERTER's next step is to use *micro execution* [18] to directly run each function containing an injection site in an attempt to induce a crash (Section IV-B). Micro execution fills in function arguments lazily, based on how they are used, and thereby explores many possible behaviors. It is under-approximate, and thus serves to complement the static analysis.

The last step of FIXREVERTER is to inject the bugs at the target sites that remain after the prior steps (Section V). To have clear indicators when an injected bug is triggered, FIXREVERTER logs when the invariants from the removed checks hold.

While we have developed FIXREVERTER using particular patterns and analyses, these can all be adjusted as desired to improve the range and quality of injected bugs.

C. A New Benchmark: BUGBIN

We construct BUGBIN (Section VI) by running FIXREVERTER on four binutils programs, which are common in prior fuzzing evaluations (**G1**).

In our experiments, we first show that FIXREVERTER is capable of reintroducing previous bugfixes in the CVEs, illustrating that the bugs injected by FIXREVERTER are realistic. We then evaluate the effectiveness of each component of FIXREVERTER by comparing to a baseline (i.e., injecting all matched pattern instances) and through a manual analysis. We show that the static dependence analysis and dynamic reachability analysis effectively increase the confidence that the injected bugs are exploitable. Last but not least, we compare the performance of AFL and Angora on BUGBIN. Using the performance metrics built in our benchmark, Angora outperformed AFL and these tools detected very different bugs in BUGBIN. (Section VII)

D. Contributions

The contributions of this paper are the following.

- A framework, FIXREVERTER, that automatically injects realistic, exploitable bugs in real programs, by using static and dynamic analyses to identify reversible instances of bugfix patterns.
- A benchmark, BUGBIN, that meets all 4 requirements, useful for evaluating fuzzing tools.
- An evaluation that assesses the effectiveness of FIXREVERTER's components and compares state-of-the-art fuzzers using BUGBIN.

II. BUGFIX PATTERN MATCHING

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. FIXREVERTER's grammar-based *pattern matcher* analyzes all files in the target program to find the code regions that match a given syntactic bugfix pattern. Each matching region is a candidate injection site, which is then vetted by subsequent analysis phases. To assist this analysis, the matcher can return the variable declarations that are used in the matched pattern instances, i.e. the so-called *traced variables*.

We inspect past bugfixes in the Common Vulnerabilities and Exposures (CVEs) to extract and define common syntactic bugfix patterns. We discuss these first, and then how FIXREVERTER matches the identified patterns.

A. Sources of Past Bugfixes

We performed a study of the 100 most recent CVEs of binutils as of July 2019. We chose binutils for three reasons. First, binutils is a well-studied library in the fuzzing literature [8], [16], [17], [19]–[22] that contains well-documented CVEs. Second, most of binutils' CVE records contain direct links to the source code diff that fixed the bugs. Third, most of binutils' CVE records are reproducible and come with call stacks to help understand the origins of the bugs and how the bugfixes work.

For each CVE, we spent 10 minutes with the provided test input, call stack, and code diffs in order 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 10 CVEs among the 100 as a basis for a general bugfix pattern, discussed below. We call these CVEs our *bugfix regression dataset*.

B. Bugfix Pattern Definitions

We call the syntactic pattern we abstracted from our bugfix regression dataset *small-if-check*: for each fix, the developer adds a simple if-statement to check that a variable (or path) satisfies an invariant, and breaks the flow of control (e.g., returns from the function) if it does not. Removing such an if-statement would reverse the fix and thus reintroduce the bug. Our key insight is that if this code-fix pattern is found elsewhere in the program, then removing it could introduce a bug similar to the one the same pattern was used to fix.

We can express the small-if-check pattern using a context-free grammar (CFG), shown in Listing 1. By convention, non-terminals are in uppercase and terminals are in lowercase. This CFG specifies that an if-statement falls into the pattern when (1) it has no else branch; (2) it has a small body with less than 3 statements; and (3) the body ends with a jump instruction (i.e., **`return`**, **`break`**, **`goto`** or **`continue`**).

The production rules for *CONDS* further refine three patterns—*NULL_CHECK*, *PTR_RANGE*, and *NUM_CMP*—shown in Listing 2, which are based on the bugfixes we observed. We match against these patterns in the guard directly, or when in sub-expressions of logical operations (see lines 1

```
1  SMALLIF → CONDS BODY
2  BODY → | JUMP | less3 JUMP
3  JUMP → break | goto | return | contiune
```

Listing 1: Grammar of the small-if-check pattern.

```
1  CONDS → COND LOGICOP CONDS
2  LOGICOP → ||:logic[] | &&:logic[]
3  COND → NULL_CHECK | PTR_RANGE | NUM_CMP
4  NULL_CHECK → ptrVar:traceVar[] RELOP ZEROVAL
5             | ZEROVAL RELOP ptrVar:traceVar[]
6  RELOP → == | !=
7  ZEROVAL → null | cast null
8  PTR_RANGE → PTR_OP CMP PTR_OP
9  PTR_OP → ptrVar:traceVarPtrRange[] ARITH NUM_VAL
10        | NUM_VAL ARITH ptrVar:traceVarPtrRange[]
11        | ptrVar:traceVarPtrRange[]
12 NUM_CMP → NUM_OP CMP NUM_OP | NUM_OP REL_OP NUM_OP
13 NUM_OP → NUM_VAL ARITH NUM_VAL | NUM_VAL
14 NUM_VAL → NUM | num:traceVarNumCmp[]
15         | SINGLECALLER ptrVar
16         | SINGLECALLER ptrVar ( NUM )
17         | SINGLECALLER var
18         | SINGLECALLER num
19 NUM → lit_num | null
20 SINGLECALLER → sizeof | function
21 ARITH → + | - | * | /
22 CMP → < | > | <= | >=
```

Listing 2: Grammar of the conditional expression patterns.

```
1  static bfd_boolean apply_relocations (/*parameters*/)
2  {
3    unsigned char * rloc;
4    ...
5  - if ((rloc + reloc_size) > end || (rloc < start))
6  + if (rloc>= end || (rloc + reloc_size) > end || (rloc < start))
7    {
8      warn (_("skipping invalid relocation offset 0x%lx in section %s\n
           "),
9        (unsigned long) rp->r_offset,
10       printable_section_name (filedata, section));
11     continue;
12   }
13   ...
14   byte_put_little_endian (rloc, addend + sym->st_value, reloc_size
         );
15   ...
16 }
17
18 void byte_put_little_endian (unsigned char * field, elf_vma value,
       int size)
19 {
20   switch (size)
21   {
22     ...
23     case 2:
24       field[1] = (value >> 8) & 0xff;
25     ...
26   }
27 }
```

Listing 3: CVE-2019-14444 bugfix.

and 2). A terminal may be associated with a *tracer*, which is an annotation following a colon, e.g, `logic[]` in `||:logic[]` on line 2. Tracers are used by FIXREVERTER to identify the symbols for tracking data dependence and/or performing bug injections, as detailed in Section II-C.

NULL_CHECK: We define *NULL_CHECK* as an expression that matches a terminal **ptrVar** checked as equal or not equal to **null** (lines 4-7 in Listing 2). Such a check is used to avoid downstream NULL dereferences. The bugfix for CVE-2017-8395 [15], shown in Figure 2, is an example of *NULL_CHECK*.

The developer's fix is shown as the green-colored lines prepended with + in the figure. Originally, a NULL dereference error was reported on the variable buf in the fread call in function cache_bread_1. This bugfix allows the function bfd_init_section_compress_status to return early if uncompressed_buffer is NULL, thus no longer reaching the crash site. Through the inspection of source code and call stack of this CVE, we notice that there are six calls in between these two functions in the call stack while the variables buf and uncompressed_buffer point to the same address. This example illustrates the potentially long code distance that can exist between the error location and the bugfix location while the variables are still related through data propagation, motivating our solution to use a static data dependence analysis in Section III. Two CVEs in our regression dataset match *NULL_CHECK*.

PTR_RANGE: Checking that a pointer variable's address is within a range can fix bugs such as buffer overflows and overreads. Listing 3 shows the *PTR_RANGE* bugfix for CVE-2019-14444 [23] in `readelf`. The original check at line 5 is insufficient to avoid the integer overflow at line 24 that allows an attacker to trigger a write access violation [23]. To fix this bug, the developer added another conditional expression rloc >= end (line 6) to skip the invalid relocation offset. Four CVEs in our regression dataset use *PTR_RANGE*.

For FIXREVERTER, we define *PTR_RANGE* as the expressions that compare $(<, \leq, >, \geq)$ two pointer operations (line 8 in Listing 2). A pointer operation is a pointer variable (line 11) or the arithmetic operation $(+, -, \times, /)$ between a pointer variable and a number (lines 9-10). Note that the pointer variables used in *PTR_RANGE* may affect the existence of the bug as shown in Listing 3.

NUM_CMP: Checking that a numeric variable falls within a range can avoid using it in an out-of-bounds access, as in a buffer overread or overwrite. For example, Listing 4 shows the bugfix for CVE-2017-9756 [24]. The aarch64_ext_ldst_reglist function in binutils 2.28 allows remote attackers to cause a denial of service (buffer overflow and application crash) [24]. The fix includes a check on value before using it as the index to access data at line 9, to make sure it is within the bounds of data; otherwise, the function returns early (lines 6-8). Four CVEs in our regression dataset use *NUM_CMP*.

At lines 12-13 in Listing 2, we define *NUM_CMP* 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 variable, or the result of evaluating a function call or array access that returns a numeric type (lines 14-18).

```
1  int aarch64_ext_ldst_reglist (/*parameters*/)
2  {
3    aarch64_insn value;
4    ...
5    struct {/*struct definition*/} data [];
6  + /* PR 21595: Check for a bogus value. */
7  + if (value >= ARRAY_SIZE (data))
8  +   return 0;
9    if (expected_num != data[value].num_elements || data[value].
         is_reserved)
10     return 0;
11   ...
12 }
```

Listing 4: CVE-2017-9756 bugfix.

We note that CVE-2017-9745 [25] uses both *NUM_CMP* and *PTR_RANGE* in its bugfix, motivating us to detect *small-if-check* that contains more than one kind of conditional expression pattern.

Small-if-check represents a general, intuitive pattern to demonstrate the effectiveness of FIXREVERTER. Indeed, bug-fixes similar to it were considered in previous studies of bug repositories [26] and automatic bugfixing tools [27]. We expect FIXREVERTER can be extended to support many other patterns, too.

C. Pattern Matcher

FIXREVERTER's pattern matcher is implemented as LibTooling library based on Clang 8 [28]. It works in two phases. In the first, it parses in a grammar file (i.e., Listings 1 and 2) and converts it to a *state machine*. Phase 2 traverses the Clang abstract syntax tree (AST) using the visitor pattern. As it encounters if-statements, it finds tokens in each if-statement and feeds them into the state machine to see if it matches a defined pattern. When the if-statements are matched, the tokens that were gathered are placed into a JSON file to be used in the later stages of FIXREVERTER.

Phase 1: When running the pattern matcher, a pattern file is specified as a command-line argument. This file takes the following format:

$$NT \rightarrow Token1\ Token2\ ...\ TokenN$$

where NT is a non-terminal, and a token can either be another non-terminal, or a terminal. In our grammar, a terminal may be associated with a *tracer*. Information about the traced terminals is important for the later stages of FIXREVERTER. For example, the name and declaration of a variable that appear in the conditional expression of an if-statement are used by the static dependence 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, line 4 in Listing 2 ensures all variables used in *NULL_CHECK* are traced.

In phase 1, the grammar file is first parsed into a list of productions; then, a rudimentary LR(0) parser uses the list of productions to create a state machine. Each state represents a possible position in the provided grammar. As tokens are sent in during phase 2, the token will determine what state we transition to.

Phase 2: After creating the state machine, we traverse the AST using the visitor pattern to perform semantic actions. 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 node, 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. If the state machine enters a valid state after parsing the *BODY* (so there is no else branch) then the match is confirmed and all the traced terminals are output by the pattern matcher.

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 example, terminal **`less3`** in line 2 of Listing 1 represents any compound statement body which has less than 3 internal statements. In addition, we can also 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 19 in Listing 2). We also distinguish some terminals based on the type of a variable, or whether that variable is a pointer. For example, in *NULL_CHECK*, we require a pointer variable to be compared to the value 0 at lines 4 and 5 in Listing 2.

Limitations of pattern matcher: First, if an **`#include`** appears in the middle of a function, the pattern matcher currently matches the patterns in the included code (we never saw this happen in our benchmarks). Second, while all terminals in our test cases are accounted for, we may not have handled all possible C terminals in the pattern matcher.

III. STATIC DEPENDENCE ANALYSIS

We only want to consider injections that are likely to result in interesting and exploitable bugs. The first step after pattern matcher is the *naive bug filter*. As discussed in Section I, we filter the uninteresting bugfix patterns that will fail too easily if injected. FIXREVERTER's *static dependence analysis* is then used to provide additional evidence that candidate injections will indeed produce viable bugs.

The static dependence analysis takes as input the set of traced variables output by the pattern matcher. Each traced variable is considered as a *source*. The static dependence analysis does two things. First, it decides whether this source could be *reached* via an execution from a designated entry point. Second, it determines if the source trace traced variable can influence a subsequent pointer dereference, which acts

as a *sink*. (These are the two vertical lines on the right in Figure 2.) Such sinks are chosen because the memory bugs we want to inject (i.e., NULL and out-of-bounds dereferences) can be exploited when a pointer is dereferenced. The analysis determines whether a sink's dereference may be affected by if-checked source variable(s); if so, reversing the if-statement check may introduce an exploitable memory error.

We implemented the analysis using the Phasar C-code static analysis framework [29]. Phasar provides an efficient, summary-based solver of Inter-procedural Finite Distributive Subset (IFDS) [30] dataflow problems, including taint analysis. IFDS solves a dataflow problem by constructing an exploded super-graph (ESG), replacing every node of its inter-procedural control-flow graph (ICFG) with the bipartite graph representation of the respective flow function [30]. Our implementation is based on Phasar release v0619 [31] using LLVM 8.0 [32].

FIXREVERTER's analysis extends Phasar's existing IFDS-TaintAnalysis 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-insensitive [33]. When constructing the ICFG, it uses a pointer analysis [33] to resolve the targets for function pointers. The analysis operates on the LLVM Intermediate Representation (IR) [32]. The whole-program IR for the target program is generated using WLLVM [34], which takes the `makefile` of a program and produces object files with LLVM bitcode (IR) stored in a dedicated section. A whole-program bitcode file is then extracted with a WLLVM utility `extract-bc`, which is further disassembled with the `llvm-dis` utility into a whole-program bitcode file.

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 traced variable may lead to exploitable memory error, we extend the IFDSTaintAnalysis module with a global map to keep track of the sources of tainted variables. Each key—a tainted variable—maps to the list of variables that its taintedness originates from. The map is updated every time a variable gets tainted. When the IFDS analysis completes, we use the map to output the pairs of sources/sinks that have data dependence.

To identify the sources (i.e., declarations of traced variables) in the IR, we use the line and column numbers of all traced variable declarations produced by the pattern matcher, and match against the debug information in *llvm.dbg.declare* instructions [35] to retrieve the corresponding IR registers as the sources. We identify the sinks (i.e., pointer dereferences) as instructions that perform a dereference in the IR.

Limitations of static dependence analysis: Our static dependence analysis inherits several limitations of Phasar's IFDSTaintAnalysis. First, the pointer analysis used for resolving function pointers is unsound, making the analysis unable to reach some functions that are reachable during actual execution. Second, the analysis is field-insensitive, making it imprecise to track the tainted flow among struct fields. Third, 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.

IV. DYNAMIC ANALYSIS

The static dependence analysis may over-approximate which injection sites are reachable, and may likewise over-approximate which of these will, if an injection takes place, lead to a fault. FIXREVERTER employs *dynamic analysis* to increase the confidence that we inject exploitable bugs. A *dynamic reachability analysis* determines whether an injection site can be reached. *Micro execution* determines whether an injected bug could be exploited if execution were to start from the bug site's surrounding function.

A. Dynamic Reachability Analysis

The dynamic reachability analysis takes in the matched pattern instances confirmed by the static dependence analysis. Because greybox fuzzers typically aim to achieve high code coverage in the target program [19], they can satisfy our goal of confirming if the code injection sites can be reached during execution. FIXREVERTER uses state-of-the-art greybox fuzzers AFL [1] and Angora [16].

Some injection sites may not be easy to reach, even by greybox fuzzing. Such *hard-to-reach* instances are particularly valuable for a benchmark, to promote the future development of fuzzers. For these injection sites we use *directed* fuzzing, which aims to generate tests that will reach a set of target locations [36]. FIXREVERTER uses the directed fuzzer AFLGo [17], supplying it with the injection sites not reached by AFL and Angora. All injection sites reached by any tool we use are returned as the output of the dynamic reachability analysis.

Limitations of dynamic reachability analysis: Determining whether an arbitrary location in a program is reachable is an undecidable problem. Dynamic tools reflect this undecidability by failing to reach in-fact reachable locations (whereas static analyses will often err in the reverse direction). We apply multiple tools with different mechanisms to improve our overall chances of success. Other dynamic input generation tools can be used as well.

B. Micro Execution

In addition to code reachability, static dependence analysis also over-approximates the data dependence between a source (the traced variable) and a sink (the pointer deference). To increase confidence on the exploitability of the injections by FIXREVERTER, we use a dynamic analysis to confirm if the program will crash when the condition of the matched small-if-check instance is satisfied.

Generating a concrete input that satisfies an injection site's **if**-condition is not necessarily straightforward. As such, we execute the injection site's function using micro execution [18]. This technique allows users to execute any code fragment

```
1 static bfd_boolean getvalue (char **srcp, bfd_vma *valuep, char *
      endp)
2 {
3   char *src = *srcp;
4   ...
5 + if (src >= endp)
6 +  return FALSE;
7   if (!ISHEX (*src))
8     ...
9 }
```

Listing 5: CVE-2017-9954

```
1 /* variable p is a traced variable*/
2 +if (p == NULL)
3 +  log("triggered bug index 1!");
4 +else
5 +  logf("reached bug index 1!");
6 - if (p == NULL || a > 10)
7 +if (a > 10)
8 -  return 0;
9 +  return 0;}
```

Listing 6: Bug injection example.

without a user-provided test driver or input data, via a customized Virtual Machine (VM). This VM catches all memory operations before they occur and allocates memory as needed, on the fly.

FIXREVERTER uses the BAP micro execution framework [37]. We supply BAP with each function that contains a injection site. Prior to starting, we modify the function to ensure the bugfix pattern is reversed and its condition is satisfied. A run-time error at a pointer dereference during the micro execution confirms that reversing the bugfix can crash the program *if* the condition is satisfied in an ordinary execution. For *NULL_CHECK*, the complete if-statement is replaced with an assignment that assigns the traced variable to NULL, as shown in the example below.

```
- if (p == NULL)
-   return 0;
+ p = NULL;
```

We then run the BAP micro executor to check if a NULL dereference is reported. We use the micro executor's symbolic execution mode (which tracks logical path conditions during micro execution) to help avoid infeasible paths. By default, the BAP micro executor does not crash when a NULL pointer is dereferenced. Instead, we write a plugin to instruct BAP to throw an error when dereferencing an address that falls within the zero page (0x0..0x1000) and filter out non error-producing injection sites.

Limitations of micro execution: First, the assumption we make to run the micro execution, i.e., that the condition of the small-if-check can be satisfied in an ordinary execution, may not be true, leaving some infeasible bugs. This can happen if the program contains multiple checks of the same pointer in different locations or simply the invariant cannot be satisfied from a specific entry point.

Second, we currently only support micro execution on *NULL_CHECK*. Supporting *PTR_RANGE* and *NUM_CMP* would require supplying BAP with bounds information, which is not readily available. To see why bounds are needed, consider Listing 5. The function getvalue may crash with a buffer overread at line 7 without the fix at lines 5 and 6 which checks whether src exceeds its bound (endp). Though BAP can start micro executing function getvalue, it needs to know that endp bounds srcp. Without this constraint, BAP cannot accurately flag the out-of-bounds access.

V. BUG INJECTION

We implement the FIXREVERTER bug injector as a Clang LibTooling [28]. The injected code allows a developer to distinguish the following four states for each injected bug after fuzzing a target.

- *Not reached*: the fuzzer never reaches the if-statement;
- *Reached*: some inputs generated by the fuzzzer reach the if-statement;
- *Triggered*: the condition over the traced variables in the reversed if-statement's guard is satisfied.
- *Crashed*: an injection is triggered *and* the program crashes because of it.

Similar bug states are used in Magma [13].

To perform an injection, we take each if-statement output from the previous step of FIXREVERTER. Based on the grammar in Listing 2, the conditional expression of the if-statement is a sequence of conditional sub-expressions separated by logical operators. For each sub-expression that contains a traced variable, we revert this sub-expression to inject a bug. To indicate when the injected bug is reached or triggered, we insert a check to the sub-expression to log *triggered* if the condition is satisfied and log *reached* if the condition is not satisfied. There may be other sub-expressions in the if-statement that do not contain a traced variables, we retain the original program's flow of control for these sub-expressions.

We illustrate the results of bug injection in Listing 6. The if-statement prior to injection is shown in lines 6 and 8. Assuming p is a traced variable, the condition p == NULL is rewritten (lines 2–5) so that the program no longer returns when the condition is triggered, and also logs whether the condition is triggered or is merely reached. The index in the log message is an ID maintained by the previous step of FIXREVERTER. The remaining sub-expression in the if-statement, a > 10, still returns 0 when satisfied, as shown in lines 7 and 9.

When using FIXREVERTER to benchmark a fuzzer, it is possible that multiple injected bugs could be triggered prior to a benchmark program crashing. To determine which injection is the root cause, we perform a triaging step. In particular, we examine the log to determine which bugs are triggered; for each bug i (if there is more than one), we produce a version of the target program with just that bug injected (we make this easier using preprocessor directives also inserted by FIXREVERTER). Then we re-run using the crashing input. For

TABLE I: BUGBIN.

Program	Size (LLVM bitcode)	Parameter	# of injections
cxxfilt	7.9 MB	-	29
nm	8.0 MB	-C	102
objdump	11.5 MB	-D	59
readelf	3.9 MB	-a	63

each bug index i for which the target crashes, we designate i as *Crashed*.

Note that it is possible that multiple bugs can individually cause a crash on the same input. For example, imagine that one part of an input file triggers a bug in an initial stage of input processing, but if that bug is removed, a different part of the same input file triggers a bug in a later stage. It is also possible that some combination of injected bugs is required to cause a crash, i.e., running with injected bugs individually will not cause a crash. In that case, no individual bug will be blamed. We observe both cases in our evaluation (see details in Section VII).

VI. GENERATING BUGBIN

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

A. Target programs

We chose the utility programs `cxxfilt`, `nm`, `objdump`, and `readelf` in the binutils 2.34b package to create BUGBIN. These programs are frequently used in the literature to evaluate fuzz testing [8], [16], [16], [17], [19]–[22]. Columns 2 and 3 in Table I show the size of each program (its program-specific LLVM bitcode), and the command-line options used to run them (which have been used in past fuzzing evaluations [16], [19], [21], [22]).

B. FIXREVERTER *usage*

FIXREVERTER ultimately injected 29, 102, 59, and 63 bugs in `cxxfilt`, `nm`, `objdump`, and `readelf`, respectively, to form BUGBIN, per the last column in Table I. Because some code of these programs is shared in the binutils codebase, there are overlapped bugs between programs. Among the 253 injected bugs, there are 179 unique ones.

Pattern matcher: We ran FIXREVERTER's pattern matcher on each C source file in binutils. Because each binutils program only reaches a portion of the overall source code, we use WLLVM [34] to build the program and link only the necessary files to generate the IR. The pattern matcher result of each program thus only contains the result in the functions that appear in that program's IR.

Naive bug filter: Next, for each target program and each potential injection site we inject the bug at that site and run with the default fuzzing seed (discussed in the dynamic reachability analysis part below). We drop from consideration any injections that cause the program to (immediately) fail.

Static dependence analysis: To run the static dependence analysis requires specifying a set of program entry points. The natural approach is to select a program's main function as its entry point, with the aim of considering all possible pattern-matched locations reachable from main as candidate injection sites. We call this the *main-entry* approach. Another approach, which we call *all-entry*, is to use both main and those functions in which a matched pattern appears as entry points. Doing so essentially takes as given that these functions are reachable, and thus focuses only on using the dependence analysis to determine the likelihood of a fault if a bug is injected. Taking reachability as a given avoids possible soundness problems in the analysis' determination of reachability (as discussed in Section III), but also will cause the taint analysis to miss dereference sites in callers between main and the injected function.

Based on our study of this tradeoff for these programs (Section VII), we chose the *all-entry* mode to generate BUGBIN.

Dynamic reachability analysis: We ran the dynamic reachability analysis using AFL (version 2.57b), Angora (version 1.2.2), and AFLGo (version 2.52b). We used a small seed string *hello* in the input file for `cxxfilt`, `nm` and `objdump`. Our preference would have been to use the empty seed (i.e., an empty file) [8], but Angora rejected the use of this seed. For `nm` and `objdump`, we considered using a small executable file (a valid input) but in preliminary experiments found that the small string actually achieved more interesting coverage (it may be that the small executable drove the target programs into system-specific logic and precluded a broader exploration). For `readelf`, however, fuzzers struggled with the small string input, so we used a small ELF file instead.

We ran AFL five times, each with a 24-hour timeout on the target programs compiled with Address Sanitizer (ASAN) [38]. Angora was also run 5 times using its default settings [39], with a 24-hour timeout. AFLGo was supplied with the **if**-statement line numbers returned by the static dependence analysis but not reached by AFL and Angora as its targets. It was then run 5 times, each with a 24-hour timeout.

Micro execution: We did not end up using micro execution to create BUGBIN because we found it exhibited too many false positives and false negatives (see Section VII).

VII. EVALUATION

This section presents experimental evaluation of FIXREVERTER, which aims to answer the following research questions (RQs).

- **RQ1:** how effective is each component of FIXREVERTER?
- **RQ2:** how do fuzzers perform on BUGBIN?
- **RQ3:** are the injections by FIXREVERTER realistic?

To answer **RQ1**, we must assess how well FIXREVERTER filters out possible injection sites that are not (or are unlikely to be) triggerable bugs. If we had ground truth—i.e., we knew which pattern-matched sites constitute actual, triggerable bugs—we could determine, for each component, how many non-bugs are retained and how many actual bugs are filtered.

As a first approximation, we can simply count how many injection sites are filtered by each step, to get a sense of how well (potential) non-bugs are being removed. If each component was entirely over-approximate, we would be sure that no real bugs are getting filtered; however, both the static analysis and the dynamic analyses may fail to exercise feasible paths (they are under-approximate), which means they may incorrectly filter real bugs. Since we do not have ground truth, we cannot be sure how often that happens. However, we can make an empirical assessment. In particular, we can confirm that a *baseline* subset of the pattern-matched bugs are indeed triggerable (e.g., by using fuzzing), and then consider whether (and how often) triggerable bugs get filtered, post-matching, and why.

To answer **RQ2**, we run AFL and Angora on BUGBIN. We compare the number of bugs reached/triggered/crashed by each tool.

To answer **RQ3**, we try to reintroduce bugs in *bugfix dataset* with FIXREVERTER. We run FIXREVERTER on the commits where the bugs were fixed and evaluate each stage of FIXREVERTER on reintroducing the bug.

All the experiments were run on a server with 2 Xeon(R) Silver 4116 CPUs, 192GB RAM and Ubuntu 16.04. AFLgo and Angora were executed through Docker due to conflicts of setting up different LLVM versions from the static dependence analysis.

A. RQ1: how effective is each component of FIXREVERTER?

A breakdown of the impact of each step of running FIXREVERTER to produce BUGBIN (Section VI) is shown in Table II. The numbers in Columns 3–5 to the left of the "/" show the number of remaining injection sites after running each component of FIXREVERTER. For example, the row *NULL_CHECK* of `cxxfilt` shows that 619 *NULL_CHECK* injection sites are identified by the pattern matcher. The static dependence analysis filters out 97 and the dynamic reachability analysis further filters out 494 to result in 26 remaining sites.

The numbers to the right of the "/" in column 3 indicate the number of sites that are part of the *baseline* set of confirmed-triggerable (*crash*) bugs. We produced this set as follows. For each program, we injected all but the naive bugfix patterns identified by the pattern matcher, and then ran AFL and Angora (five 24-hour trials for each) on the program. For any crashes that occurred, we triaged which bug was responsible (per Section V) and added it to the baseline. Doing so reinforced the value of knowing ground truth: 29980 unique, crash-inducing inputs generated by fuzzing amounted to only 63 individual bugs (last row in the table). Triaging was done per Section V: We produced versions of each program with just a single bug injected, one version per bug. Then we reran each version on all crashing inputs for that program, keeping track of which versions crashed (meaning that the single bug in that version was the reason for the crash). 12 inputs managed to individually trigger multiple bugs; 17739 inputs triggered no bug individually—multiple bugs needed to be present for the crash. Such *interference* [40] between multiple injections is one reason we prefer to have fewer overall injections, since it simplifies benchmarking.

TABLE II: FIXREVERTER results. The static dependence analysis uses *all-entry* mode, and the dynamic reachability column shows the union of AFL, Angora and AFLGo results.

Program	Pattern	Pattern matcher	Static dependence	Dynamic reachability
cxxfilt	*NULL_CHECK*	619 / 9	522 / 6 (-3)	26 / 6
	NUM_CMP	243 / 2	94 / 2	3 / 2
	PTR_RANGE	22 / 0	22 / 0	0 / 0
nm	*NULL_CHECK*	624 / 13	525 / 10 (-3)	88 / 10
	NUM_CMP	243 / 4	94 / 3 (-1)	12 / 3
	PTR_RANGE	22 / 0	22 / 0	2 / 0
objdump	*NULL_CHECK*	826 / 4	670 / 4	52 / 4
	NUM_CMP	390 / 4	165 / 3 (-1)	5 / 2 (-1)
	PTR_RANGE	66 / 1	54 / 0 (-1)	1 / 0
readelf	*NULL_CHECK*	152 / 16	115 / 14 (-2)	49 / 14
	NUM_CMP	191 / 10	69 / 3 (-7)	11 / 2 (-1)
	PTR_RANGE	34 / 0	34 / 0	3 / 0
total		3432 / 63	2386 / 45 (-18)	253 / 43 (-2)

Returning to the table: The negative numbers after the "/" in columns 4 and 5 (if any) show the number of crash bugs in the baseline filtered out by each component. For example, the row *NULL_CHECK* of `cxxfilt` shows that the baseline contains 9 crash bugs from the 619 injection sites. Static dependence analysis (imprecisely) filters out 3 of these bugs and dynamic reachability analysis does not further filter out any more, leaving 6 confirmed *NULL_CHECK* crash bugs in `cxxfilt` from FIXREVERTER's results.

Per the last row in the table, FIXREVERTER ultimately injects 253 bugs in the four target programs. Among them, 43 bugs are in the confirmed baseline while 20 other bugs in the baseline are not injected. 49 out of 63 confirmed baseline bugs are unique across the entire binutils codebase (some code is shared) and `cxxfilt`, `nm`, `objdump`, and `readelf` have 1, 3, 5, 26 unique bugs respectively.

The next few subsections investigate the impact of each component. Overall, we observe that *static dependence analysis (all-entry) and dynamic reachability analysis both have a positive impact on injecting exploitable bugs, while micro execution is limited by its false positives and false negatives.*

B. Impact of static dependence analysis

The goal of the static dependence analysis is to retain those potential injection sites that are *reachable*, and for which a traced variable induces a dependence on a downstream dereference, which could lead to a crash. Filtering unreachable, non-dependent sites avoids uninteresting injections and reduces the chances of interference.

Column 4 in Table II shows that static dependence analysis using all-entry mode retains 2386 out of 3432 results produced by pattern matcher. Among the 63 crashed bugs found by the baseline, static dependence analysis filters out 18. This is unfortunate; ideally, the static analysis would filter only non-viable bug injections. We identified the following reasons for missing these crashed bugs.

- 6 crash bugs (2 in `cxxfilt`, 3 in `nm`, and 1 in `objdump`) are dropped due to the unsoundness of

```
1 if (remaining < 2)
2     break;
3 *buf++ = '^';
4 *buf++ = c + 0x40;
```

Listing 7: Control dependence example.

Phasar's handling of function pointers. While all-entry mode is intended to mitigate Phasar's unsoundness, it only sidesteps modeling calls via function pointer that reach the injection site; it does not sidestep that calls via function pointer may be needed to go from that site to a dereference, causing 2 missed crash bugs. All-entry mode also introduces a new problem: a crash might occur at a dereference in a function that called the one containing an injection. If this call is not reachable from the main function due to the handling of function pointers, it still cannot be modeled in all-entry mode, because the all-entry mode starts the analysis from the function containing the injection without knowing its caller. This problem causes 4 missed crash bugs. We experimentally study these two modes in greater depth, below.

- 5 crash bugs (1 in `cxxfilt`, 1 in `nm`, 1 in `objdump`, 2 in `readelf`) are dropped due to what amounts to misconfigurations of Phasar. Two are dropped because the crashing pointer dereference is in the library code which the static dependence analysis does not analyze directly. The library specification we use missed these sinks. The remaining three bugs arose from other issues involving use of proper flags in Phasar and our implementation.
- 7 crash bugs (all in `readelf`) are dropped because the analysis does not track *control dependence* [41] from the traced variable to pointer dereference. In these cases, the developer's intention is to use a small-if-statement to guard the bounds of a pointer, skipping the dereference if the check fails. For example, Listing 7 shows an example code from `readelf`. remaining is a local variable that counts the remaining number of words toward the bound of the pointer *buf*. Thus, the small-if-statement guards buf from out-of-bounds dereference. However, the traced variable remaining does not have a *data* dependence on buf. We could hope that the static dependence analysis could track control dependences too, but past work [42] has shown that such analyses often claim many spurious dependences, which would reduce the overall utility of the analysis step. In any case, Phasar does not support tracking of control dependences but if/when it does, this is something to consider.

In sum, 11 of the bugs are due to unsoundness or misconfiguration; we have been working with the Phasar developers to fix this soundness problem, and improve the default configuration. The remaining 7 bugs relate to the non-analysis of control dependences; precise analysis of control dependencies may require new research.

TABLE III: All-entry vs. main-entry approaches in static dependence analysis.

Program	all-entry		main-entry	
	# of results	Runtime (m)	# of results	Runtime (m)
cxxfilt	638	388	45	6
nm	641	399	348 / -3	178
objdump	889	3726	554 / -2	1427
readelf	218	579	187	362

Finally, we study the tradeoffs in between the *all-entry* and *main-entry* approaches of the static dependence analysis. With a sound analysis, main-entry would be preferred, since it would capture reachability and dependence properly; the all-entry approach uses main but also the injection sites' functions as entry points. Table III shows the different results these two modes produce on the same pattern matcher results as input. Similar to Table II, we use the numbers after "/" in column 4 to show the additional number of crashed inputs missed by main-entry mode. We observe that the main-entry mode runs 1.6 to 65 times faster, and retains 31 to 593 fewer patterns than the all-entry mode. That said, main-entry mode also discards more viable crash sites, due the unsoundness of handling function pointers.

C. The impact of dynamic reachability analysis

Because static analysis can be over-approximate, it may mistakenly decide that injection sites are reachable. Dynamic analysis can add confidence that injection sites are actually reachable. The last column in Table II shows that FIXREVERTER's dynamic reachability analysis retains 253 out of 2386 patterns and only misses 2 more baseline crash bugs from the static dependence analysis results. That it retains most baseline bugs should be unsurprising since those bugs were produced using these fuzzers in the first place. More trials and/or longer timeouts might retain more injection sites.

Figure 3 shows the effectiveness of each tool used in the dynamic reachability analysis. Each bar counts the total number of injection sites reached by the dynamic reachability analysis as a whole, with each colored or patterned bar indicating which tool(s) reach how many bugs. We observe that *none of AFL, Angora or AFLGo was able to reach all bugs other tools can reach in most cases*, demonstrating the benefits of using multiple tools.

D. The impact of micro execution

The goal of micro execution is to give confidence that reachable injection sites will actually produce faults if triggered. In preliminary experiments we evaluate BAP's micro executor's effectiveness at this task on `cxxfilt`. Based on manual inspection of 37 injection sites, we determine that 10 injections can lead to crashes while 27 will not. The expectation is for BAP to correctly report crashes for these 10 injections while filtering at least some of the remaining 27. BAP successfully filtered 9 of these 27, but failed to retain 6 out of the 10 buggy injections. The reason seems to be a bug in BAP that causes a fault when reaching certain sorts of branches. BAP is thus, for now, sufficiently ineffective that we did not evaluate it further.

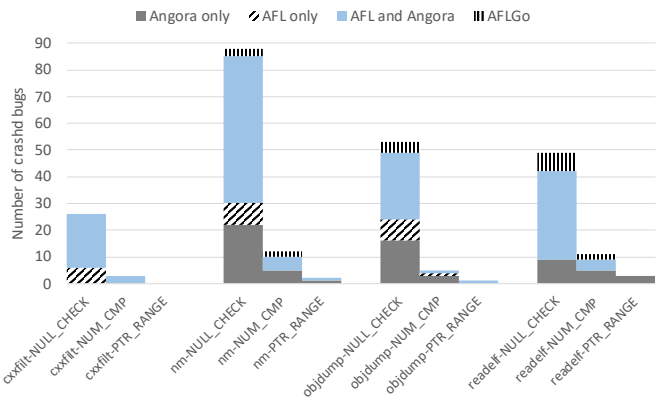

Fig. 3: Results of AFL, Angora, and AFLGo used as dynamic reachability analysis in FIXREVERTER.

E. RQ2: how do the fuzzers perform on BUGBIN*?*

As a test of the feasibility of using BUGBIN for fuzzing evaluation, we ran AFL and Angora on it. Each fuzzer was run 5 times with 24-hour timeout. Even though both AFL and Angora were used as part of FIXREVERTER to generate BUGBIN, they were only used to determine which injection sites were reachable. This means that the injected bugs in BUGBIN are likely to be reached by these tools, but the tools' capabilities to trigger and crash these bugs still worth comparing.

The results of AFL and Angora using BUGBIN's performance metrics are shown in Table IV. Each cell shows the median number of reached/triggered/crashed bugs and the Semi-Interquartile Range (in smaller font). Overall, *Angora outperformed AFL when evaluated on* BUGBIN*, while each tool detected different bugs in each program.* Angora outperformed AFL in 3 out of 4 benchmark programs. Yet, AFL detected about three times more crashed bugs than Angora did in `cxxfilt`. This result illustrates the significant different performances of these tools on different programs. In addition, both AFL and Angora detected more crashed bugs than the baseline in RQ1. AFL and Angora found 4 and 1 additional crashed bugs in `cxxfilt`, respectively, and Angora also found 1 more crashed bugs in `readelf`.

Among the 23,443 crashing inputs generated in this experiment, 6667 triggered no bug individually. Comparing to the baseline result in RQ1 (17739 out of 29980 crashing inputs triggered no bug individually), a significantly lower percentage of the crashing inputs are caused by the interference between bugs, demonstrating the benefits of injecting fewer but exploitable bugs in a benchmark.

F. RQ3: are the injections by FIXREVERTER *realistic?*

As a self-check, we would expect FIXREVERTER to reintroduce the bugs its patterns were based on. To test this, we investigated how FIXREVERTER behaved on the CVE bugs in the *bugfix regression dataset* (Section II-A). Each bug in the bugfix dataset is based on a different version of binutils and uses a different configuration to build and run. This makes it challenging to run FIXREVERTER as a whole because some components of the tool may not run because the program version cannot be built or run. We thus run each individual component of FIXREVERTER on the binutils version with the corresponding bugfix commit and show if the expected injection site is reported (by pattern matcher, static dependence analysis, or dynamic reachability analysis), or is correctly reversed (by bug injector). Table V shows the results of each step of FIXREVERTER. Overall, 2 CVEs were successfully reintroduced throughout all the components of FIXREVERTER.

As expected, FIXREVERTER's pattern matcher is able to match all of regression bugfixes in the source code. Two binutils versions containing CVE-2017-9745 [25] and CVE-2017-9756 [24] cannot be configured (using the configurations recorded in the CVEs), compiled and linked successfully to produce executables on the machine we ran the experiments. Thus, both static dependence analysis and dynamic reachability analysis cannot be run. Among the 8 left, the static analysis, however, drops 2 of them. CVE-2018-6759 [43] and CVE-2018-6872 [44] are filtered because they involve control dependences the static dependence analysis do not track, as discussed in Section VII-B.

It is only sensible to run the dynamic reachability analysis if the proper configuration and environment can be set up to reproduce a bug. To know this, we tried to reproduce the bugs on the original program versions where the bugs were report, using the provided configurations. However, we were not able to reproduce the bugs in 5 CVEs. Among the rest 5 CVEs, 3 of them were reached by our dynamic reachability analysis.

TABLE IV: AFL and Angora performance on BUGBIN.

Program	AFL			Angora		
	Reach	Trigger	Crash	Reach	Trigger	Crash
cxxfilt	27 0.5	18 0.5	11 1.0	17 2.5	9 2.0	4 1.5
nm	40 0.5	6 0.5	2 0.5	38 2.5	6 1.0	4 1.0
objdump	29 0.0	6 0.0	2 0.0	30 1.0	7 0.5	5 1.0
readelf	30 1.0	10 1.0	5 1.0	53 1.5	27 2.0	16 1.0
total	126	40	20	138	49	29

TABLE V: Reintroduce CVE bugs in bugfix regression dataset. "1" (or "0") in Columns 2-4 means the component successfully (or fails to) detect the actual bugfix site in the CVE. "1" (or "0") in Column 5 means the bug injector introduces a bug that is semantically equivalent (or inequivalent) to the actual bug. "-" means the component fails to run on the program because of misconfiguration.

CVE	Pattern matcher	Static dependence	Dynamic reachability	Bug injection
2019-14444	1	1	-	0
2018-7643	1	1	0	1
2018-6872	1	0	-	1
2018-6759	1	0	-	1
2018-10535	1	1	1	0
2017-9954	1	1	1	1
2017-9756	1	-	-	1
2017-9745	1	-	-	1
2017-8395	1	1	1	1
2017-7303	1	1	0	0

```
1  static unsigned int find_link (const bfd * obfd, const
       Elf_Internal_Shdr * iheader, const unsigned int hint) {
2    ...
3    Elf_Internal_Shdr ** oheaders = elf_elfsections (obfd);
4    unsigned int i;
5  - if (section_match (oheaders[hint], iheader))
6  + BFD_ASSERT (iheader != NULL);
7
8  + /* See PR 20922 for a reproducer of the NULL test. */
9  + if (oheaders[hint] != NULL
10 +   && section_match (oheaders[hint], iheader))
11       return hint;
12
13   for (i = 1; i < elf_numsections (obfd); i++)
14     {
15       Elf_Internal_Shdr * oheader = oheaders[i];
16 +   if (oheader == NULL)
17 +     continue;
18   ...
19 }
```

Listing 8: CVE-2019-7303 bugfix.

Finally, we use the bug injector to reverse the corresponding fix in each CVE. We say our bug injector reintroduces a semantically equivalent bug if the same conditional expression as in the bugfix is reversed. 7 out of the 10 injections are semantically equivalent. Our bug injector only partly reverted the fix for CVE-2017-7303 and CVE-2018-10535. The bugfix of CVE-2017-7303 is shown in Listing 8, consisting of 3 parts (line 6, lines 8-10, and lines 16-17). Only the fix at lines 16-17 matches our *NULL_CHECK* pattern and is reserved; thus, this bug is partly reintroduced. The developer fixed CVE-2019-14444 [23] by inserting an additional conditional sub-expression into a small-if-statement, as shown at line 6 in Listing 3. Because rloc at line 6 is considered as a traced variable and it appears in all three conditional sub-expressions, the injector removes the whole if-statement, rather than just the added condition. This will therefore reintroduce the bug, but perhaps additional faulty executions besides.

VIII. Related Work

We have discussed the limitations of the most relevant fuzzing benchmarks in Section I. In this section, we focus on how these and other works relate to FixReverter and BugBin.

LAVA is the only other bug injection framework for benchmarking fuzz testing [9]. 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 memory out-of-bound errors triggered through the comparisons of the input bytes to "magic values" (e.g., b==0x6c617661). This simple bug pattern does not reflect the actual bugs in real-world programs [14]. In addition, LAVA is constrained by the capability of the dynamic taint analysis, an expensive step that may not yield high code coverage. In contrast, FixReverter injections amend actual developer-written code, based on real-world bug fix patterns; static and dynamic analyses are used to confirm the injection's reachability.

Magma and FixReverter share the same goal of introducing realistic bugs in fuzzing benchmarks. Magma and FixReverter also use similar metrics to indicate when an injected bug is detected. However, Magma is a static benchmark whose bugs are manually injected from past CVEs, making it (eventually) susceptible to overfitting [13]. FixReverter addresses this limitation by automatically injecting bugs according to realistic bugfix patterns. Nevertheless, the bugs injected in Magma are interesting; ideally they could serve as a basis for automated injection, perhaps as part of future versions of FixReverter.

The DARPA Cyber Grand Challenge (CGC) [10] dataset is another static benchmark of manually crafted bugs, where the buggy programs themselves are synthetic, i.e., purpose-built as fuzzing/attack targets. Although its sample set has a variety of bugs, each program contains only a single bug. This setting is unrealistic for evaluating fuzzing because most real-world programs contain many bugs. FixReverter injects multiple bugs into the same program and uses real programs, not synthetic ones.

Google fuzzer-test-suite [11], recently superseded by FuzzBench [3], contains a set of bugs from real-world libraries. However, fuzzer-test-suite does not provide a clear indication when a bug is triggered, relying on the approximate stack hashes to triage crashes. FuzzBench focuses on the automated process and platform for evaluating fuzz testing, currently relying on the code coverage as the performance metric. Our work complements FuzzBench. Using FixReverter to automatically inject realistic bugs with a clear indication when a bug is found, the service provided by FuzzBench will be able to perform large-scale fuzzing evaluation using ground truth targets (bugs), rather than only a proxy (coverage).

UniFuzz [12] presents a platform that integrates 35 fuzzers and a benchmark of 20 real-world programs for evaluating fuzzers. The benchmark, however, does not incorporate knowledge of ground truth and relies on approximated approaches to triage crashes. Nevertheless, the performance metrics used by UniFuzz (e.g., speed and stability of finding bugs) are still useful for assessing fuzzers on other benchmarks.

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 the 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 [45]. For example, Getafix [27] 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., [26], [46], [47]) may be adapted to allow FixReverter to support other bug injections in the benchmark.

IX. 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 several stages of static and dynamic analysis, code patterns that match fixes of previous CVEs; reversing those patterns should (re)introduce self-signaling bugs. FixReverter-based benchmarks can easily evolve, since FixReverter can be used to inject bugs into new programs, and inject them according to 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 BugBin by using FixReverter on four binutils programs that are common targets of fuzzing tools. Our evaluation shows that FixReverter is capable of reintroducing bugs in previously-fixed CVEs. We also studied the effectiveness of each component of FixReverter and observed that both its static dependence analysis and dynamic reachability analysis increased the confidence that we inject exploitable bugs. Finally, we evaluated fuzzers AFL and Angora using BugBin and its built-in performance metrics. We observed that Angora outperformed AFL, and these tools detected very different bugs in BugBin.

We see two important future directions. First, FixReverter's analysis components have 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, and bugfixes and improvements to BAP micro execution would make it more useful. Second, FixReverter's bugfix patterns can be extended according to further study of existing fixes in bug databases [26], [27].

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] 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.

[8] 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

[9] 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.

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

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

[12] 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.

[13] 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

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

[15] "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

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

[17] 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

[18] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 539–549.

[19] 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

[20] 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.

[21] 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

[22] 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

[23] "CVE-2019-14444." Available from MITRE, CVE-ID CVE-2019-14444., 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14444

[24] "CVE-2017-9756." Available from MITRE, CVE-ID CVE-2017-9756., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9756

[25] "CVE-2017-9745." Available from MITRE, CVE-ID CVE-2017-9745., 2017. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9745

[26] 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

[27] 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

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

[29] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural 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.

[30] 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

[31] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 393–410.

[32] 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.

[33] 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

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

[35] "Source level debugging with llvm," https://releases.llvm.org/8.0.1/docs/SourceLevelDebugging.html.

[36] P. Wang, X. Zhou, K. Lu, Y. Liu, and T. Yue, "Sok: The progress, challenges, and perspectives of directed greybox fuzzing," 2020.

[37] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[38] "Addresssanitizer," https://releases.llvm.org/8.0.0/tools/clang/docs/AddressSanitizer.html.

[39] "Angora," https://github.com/AngoraFuzzer/Angora/.

[40] V. Debroy and W. E. Wong, "Insights on fault interference for programs with multiple bugs," in *2009 20th International Symposium on Software Reliability Engineering*, 2009, pp. 165–174.

[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, Jul. 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[42] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Information Systems Security*, R. Sekar and A. K. Pujari, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 56–70.

[43] "CVE-2018-6759." Available from MITRE, CVE-ID CVE-2018-6759., 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6759

[44] "CVE-2018-6872." Available from MITRE, CVE-ID CVE-2018-6872., 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6872

[45] 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.

[46] 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.

[47] 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.