Automated Customized Bug-Benchmark Generation

Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords,
Shih An Pan, Julien Henry, David Melski, and Eric Schulte
GrammaTech, Inc., Ithaca, NY 14850
{vkashyap,jruchti,lkot,turetsky,rswords,span,jhenry,melski,eschulte}@grammatech.com

Abstract—We introduce BUG-INJECTOR, a system that auto-matically creates benchmarks for customized evaluation of static analysis tools. We share a benchmark generated using BUG-INJECTOR and illustrate its efficacy by using it to evaluate the recall of two leading open-source static analysis tools: Clang Static Analyzer and Infer.

BUG-INJECTOR works by inserting bugs based on bug templates into real-world host programs. It runs tests on the host program to collect dynamic traces, searches the traces for a point where the state satisfies the preconditions for some bug template, then modifies the host program to "inject" a bug based on that template. Injected bugs are used as test cases in a static analysis tool evaluation benchmark. Every test case is accompanied by a program input that exercises the injected bug. We have identified a broad range of requirements and desiderata for bug benchmarks; our approach generates on-demand test benchmarks that meet these requirements. It also allows us to create customized benchmarks suitable for evaluating tools for a specific use case (e.g., a given codebase and set of bug types).

Our experimental evaluation demonstrates the suitability of our generated benchmark for evaluating static bug-detection tools and for comparing the performance of different tools.

Index Terms—Bug Benchmarks; Static Analysis Evaluation

I. INTRODUCTION

Several static analysis tools for finding bugs in programs exist today. Researchers in academia and industry are constantly working on creating new tools and sophisticated techniques for static bug finding. However, evaluating static analysis tools remains a challenge. A good evaluation system will guide impactful improvement in bug-finding tools by identifying their blind spots, furthering their adoption and effective use.

There are multiple aspects of static analysis tools that are important to evaluate. In this paper, however, we mainly focus on one key evaluation metric for static analysis tools: *recall*. Virtually all static analysis tools used widely for bug detection on C/C++ programs are unsound [1]. Measuring the recall of a tool helps understand the degree to which it is unsound. Answering "how well can a tool find all the bugs in a program", i.e., recall, in a convincing manner is difficult. It is hard—if not impossible—to enumerate all bugs in any non-trivial program. However, we can estimate the recall of a tool by counting how many *previously-known bugs* in a given set of programs are found by the tool. Such estimated recall rates can be particularly useful for *comparing* different tools or tool configurations. There is a large body of previous work [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] on creating benchmarks containing known bugs. Despite this significant progress, a recent study by Delaitre et al. [18]

found that there is still a shortage of test cases for evaluating static analysis tools and thus a need for real-world software with ground-truth information about known bugs.

To address this need, we first discuss some desirable properties for a benchmark suite that contains known bugs and is targeted towards evaluating static analysis tools.

Real-world-like. The benchmark's programs should be representative of real-world programs (e.g., in size and complexity).

Reliable ground truth. Each known bug in the benchmarks should manifest on at least one execution of the program. If they do not, any recall estimate based on the benchmarks is not meaningful. Ideally, each known bug should come with a proof-of-existence, such as an input that can trigger the bug.

Automated generation. To eliminate staleness, the tests in the benchmark suite should be generated automatically: on demand, without manual effort, in the quantity desired for statistical significance.

Customizable. Users should be able to generate customized benchmarks that are tailored to their codebases and bug distribution expectations: one fixed benchmark does not suit all. For example, Herter et al. [19] suggest that certain sectors (such as the aerospace and automotive industries) deem recursive function calls inappropriate. Similarly, not all bug classes are equally important to all users of static analysis tools.

Broad coverage of bug types. Benchmarks should include a broad variety of bugs: for example, bugs corresponding to a large set of different Common Weakness Enumeration entries (CWEs) [20]. Users can choose to customize their evaluation by disregarding certain types of bugs.

Suitable for usefully evaluating and comparing the recall of static analysis tools. Comparing the recall on the benchmarks should discriminate between static analysis tools. The benchmark tests should provide guidance to further improve the recall of a given tool, e.g., by including bugs which are within scope for the tool, but which the tool is unable to detect.

Implemented independently of evaluated techniques. To avoid circularity, the techniques used to create the benchmark suite should be independent of the techniques the benchmark suite will be used to evaluate. Otherwise evaluations using the benchmark suite will be biased by hiding shared limitations.

We address all of the above desired properties through BUG-INJECTOR, a system that automatically generates benchmarks containing known bugs. BUG-INJECTOR-generated benchmarks have a broad range of applications, but the one we present in

1

this paper is particularly suited to estimating and comparing the recall rates of static analysis tools, such as the open-source tools Clang Static Analyzer and Infer.

BUG-INJECTOR starts from (i) a set of *bug templates* (§III-B) that represent known bugs, (ii) a *host program*, i.e., an existing real-world software application, and (iii) a set of tests to exercise the host program. It searches dynamic traces of the host program to identify *injection points* where the state satisfies a bug template's preconditions. Using dynamic state to identify bug injection locations, rather than using information from static analyses, provides independence from bias and from the limitations of static analysis techniques (such as pointer analysis imprecision or SMT solver weaknesses). For each of the identified injection points (or a random subset thereof), BUG-INJECTOR creates a new variant of the host program by inserting a bug based on the bug template, integrating with existing data and control flow. BUG-INJECTOR relies on existing data and control flow complexity in the host programs to generate realistic contexts for injected bugs. BUG-INJECTOR outputs multiple versions of each host program, each containing one injected bug and identifying a concrete program input that will trigger the injected bug.

BUG-INJECTOR can inject bugs from a broad range of bug classes into programs of various sizes, functionality types, and complexity levels (§V). This customizability allows for additional uses of BUG-INJECTOR beyond tool evaluation. For instance, a tool developer who creates analysis checkers for a new kind of bug can use BUG-INJECTOR to generate test cases containing bugs of that kind to quickly evaluate the checker's recall against real-world software (§ III-D). This usage of BUG-INJECTOR can complement the typical method of testing analysis checkers with small, hand-crafted tests.

The specific contributions of this paper are:

1) The BUG-INJECTOR system, a novel technique to automatically generate customized, realistic benchmarks with known bugs that can be triggered using accompanying inputs. We describe BUG-INJECTOR's architecture, functionality, and underlying algorithms in § III. As far as we know, BUG-INJECTOR is the only existing system (Table I) capable of producing benchmarks that meets all desired properties discussed above.

2) Openly-available benchmark suites (§ V) generated using BUG-INJECTOR. We have created bug templates (both manually and automatically) from different sources corresponding to a wide variety of CWE [20] entries and injected them into open-source real-world programs.

3) An extensive evaluation of two leading open-source static analysis tools for C/C++ programs—Clang Static Analyzer (CSA) [21] and Infer [22]—on our generated benchmarks. Our results (§VI), show that: (a) both of these tools fail to detect bugs that are seemingly in scope for them, (b) our benchmarks can be used to compare the recall of the two tools, and (c) our benchmarks can contrast two analysis configurations of CSA, showing that BUG-INJECTOR can be used to automatically tune analysis configurations customized to a codebase. We also filed bug reports for

CSA and Infer on certain missed warnings. Additionally, we show that a closely related work, LAVA [23], is not suitable for comparing static analysis tools.

In the remainder of the paper, we compare to related work (§II), describe challenges in estimating tool recall (§IV), discuss limitations and future work (§ VII), and conclude (§ VIII).

II. RELATED WORK

Creating bug-containing benchmarks for testing and evaluating bug-finding tools has attracted significant research attention in recent years. In this section, we compare BUG-INJECTOR to the closest related work, summarized in Table I.

TABLE I: Summary comparing BUG-INJECTOR (**BI**) with other closely related work across the different properties outlined in I. Columns: **EC**=EvilCoder, **Synth**=Synthetic benchmarks, Wild=wild caught bugs. Values: Ltd.=Limited, Yes*=subject to some errors.

Property	BI	LAVA	EC	Synth	Wild
Real-world-like	Yes	Yes	Yes	No	Yes
Reliable ground truth	Yes	Yes	No	Yes*	Ltd.
Automated, not fixed	Yes	Yes	Yes	No	No
Customizable	Yes	Yes	Yes	No	No
Wide coverage of CWEs	Yes	No	No	Yes	No
Evaluate static tools?	Yes	No	No	Ltd.	Ltd.
Independent?	Yes	Ltd.	No	Yes	Yes

Synthetic benchmarks. Several efforts have targeted manual creation of artificial test programs containing bugs. Some prominent examples are: Juliet tests [2], the IARPA STONESOUP snippets [3], Toyota ITC benchmarks [4], OWASP WebGoat [5], Wilander et al., [6], [7], and ABM [8]. However, synthetic benchmarks have limited applicability in identifying how tools perform on real-world code.

Wild. Bugs may be mined and curated from real-world software. Some prominent examples of such curated bug collections are: BugZoo [9], BugSwarm [24], ManyBugs [25], Defects4J [26], [27], BugBench [10], BugBox [11], SecuriBench [12], and Zitser et al., [13]. While they have the advantage of being real-world-like, they have varying degrees of ground truth, and not all of them come with proof-of-existence. There is also very little benchmark-user customizability with respect to bug type coverage and distribution.

The curation of both wild and synthetic benchmarks requires substantial manual effort and is prone to errors (e.g., both the Juliet test cases and the Toyota ITC benchmarks have required corrections [2], [19]). They are fixed and not customizable, with pre-determined target code constructs and bug types. They therefore have limited applicability for evaluating and comparing the recall of static analysis tools. SARD [14] is perhaps the largest openly available collection of known buggy test programs, put together by the SAMATE group at NIST. It contains both synthetic and wild benchmarks.

EvilCoder. This system [15] uses static analysis to find sensitive sinks in a host program and connects them to a user-controlled source to inject taint-based bugs. A significant

disadvantage is that there is no guarantee that inserted bugs are true positives—which makes it unsuitable for estimating recall. Indeed, the paper does not evaluate bug-finding tools on EvilCoder test cases. EvilCoder injected bugs inherit the limitations of the static analysis tools used as a part of the injection pipeline, and therefore may bias evaluation of other static analysis tools. EvilCoder is limited to taint-based bugs.

LAVA. This system [16], [23] inserts bugs into host programs by identifying situations where user-controlled input can trigger an out-of-bounds read or write. LAVA bugs come with an input to trigger the bug and are validated to check that they return exit codes associated with buffer overflows. However, this approach is limited to inserting buffer overruns, and other kinds of bugs are left as future work. More recently [28], LAVA has been extended to a small number of additional bug types. LAVA test cases are generated to satisfy an additional goal: the bugs must manifest only on a small fraction of all possible inputs. This requirement seems targeted towards testing fuzzing tools; we do not think it is necessarily applicable in the context of testing static analysis tools.[1] To satisfy the requirement, LAVA injects bugs with certain patterns (the "knob and trigger" pattern which relies on magic values). It is unclear how realistic this bug pattern is with respect to bugs found in production software. A more detailed discussion of the suitability of LAVA benchmarks for static analysis evaluation is provided in § VI-E. Another closely related technique is Apocalypse [17], which is similarly targeted towards creating challenging benchmarks for fuzzing and concolic execution tools.

As opposed to the synthetic and wild benchmarks, EvilCoder, LAVA, and BUG-INJECTOR are automated and can create large number of bugs in custom real-world programs.

BUG-INJECTOR uses bug templates and a host program to produce a suite of programs containing one known bug apiece, along with an input that can trigger each bug. The available bug templates cover a large number of CWEs, and new bug templates are easy to create. Through empirical evaluation (§ VI), we show that BUG-INJECTOR generated benchmarks are suitable for evaluating and comparing the estimated recall of static analysis tools.

Other related techniques are mutation testing [30], [31] and fault injection [32], [33], [34]. Mutation testing is used to evaluate the quality of a test suite, and is different from our work because the mutations are much simpler, are not dynamically targeted, and are not guaranteed to introduce real bugs. Compared to our work, fault injection techniques serve a different purpose: they aim to evaluate the robustness of software in the presence of various kinds of faults, e.g., data corruption, errors returned by library functions. Typically, these techniques inject or emulate faults in software at runtime, and then compare the dynamic behavior of software during normal and fault-induced runs. Faults injected by these techniques are fairly simple [32], and in contrast to BUG-INJECTOR, faults are not integrated with the host program.

---

[1]Many famous bugs, e.g. HeartBleed [29], execute on the majority of possible inputs.

III. BUG INJECTOR

In this section, we describe the tooling used for BUG-INJECTOR, introduce bug templates, and describe how BUG-INJECTOR works. We illustrate the injection of a bug template into a host program, and discuss potential applications.

A. Tooling

BUG-INJECTOR is implemented using the Software Evolution Library (SEL) [35], an open-source toolchain that provides a uniform interface for instrumenting, tracing, and modifying software. SEL supports multiple programming languages. Currently, BUG-INJECTOR works on C/C++, Java, and JavaScript[2] software. In this paper, we focus on BUG-INJECTOR as applied to C/C++ software. C/C++ software modifications are implemented via Clang's libtooling API. Clang's libtooling provides a solid foundation for parsing and program modification in the presence of the latest C/C++ syntactic features, making BUG-INJECTOR applicable to a wide range of C/C++ software.

B. Bug templates

BUG-INJECTOR is able to inject a wide range of bug types, based on the provided bug templates. A bug template is defined in Common Lisp, and it specifies: (a) the dynamic and static requirements for a successful bug injection, (b) the code snippets constituting the bug itself, and (c) how these code snippets should be integrated into the program. A bug template consists of one or more *patches*.[3] An example bug template consisting of a single patch is provided in Figure 1b. Each patch has the following fields.

code. The buggy code that will be inserted into the host program. In Figure 1b, the buggy code is a call to `memcpy`. The buggy code can contain references to free variables.

free-variables. A list of type-qualified free variables in the buggy code. These are matched to type-compatible in-scope variables at the injection location in the host program. Occurrences of the free variables in **code** are replaced with the matched host program variables before injection. In Figure 1b, the free variable listing implies that `$dst` and `$src` should be bound to host program variables with type `char*`.

precondition. Any boolean predicate constructed using the following primitives defined over the in-scope variables (`$v`) at a program point (`p`) in the dynamic trace:

- `value($v, p)`: the value of `$v` at `p`
- `size($v, p)`: the dynamically allocated size of memory pointed to by `$v` at `p`
- `ast(p)`: the abstract syntax tree at `p`
- `name($v)`: the name of `$v`
- `type($v)`: the static type of `$v`

The primitives value and size allow matching on dynamic conditions, whereas ast, name, and type allow matching on static conditions. BUG-INJECTOR uses the precondition

---

[2]Java and JavaScript support is experimental, under heavy development.
[3]A successful bug injection applies all the patches in a bug template.

3

```
1  void f1(char *src) {
2    char *dst = 0; // 'dst' initialized to a null ptr
3    memcpy(dst + 0, src, 10); // expected warning: null
4    // ptr argument in call to memory copy function
5  }
```

(a) Clang Static Analyzer (CSA) regression test, annotated with expected tool behavior. This test program contains a bug: the first argument to the memory copy function is null.

code	`memcpy($dst + 0, $src, $num);`
free-variables	`$dst`: pointer to char `$src`: pointer to char `$num`: integer
precondition	`value($dst, p) = 0 ∧ value($num, p) > 1`

(b) A bug template containing a single patch corresponding to the bug in Figure 1a. For presentation, the patch has been abstracted from its Common Lisp definition.

```
1   /* global variable declarations */
2   static char *lastout;
3   static char *prog;
4   static int out_byte;
5   /* ... lots of code, some interact with globals */
6   static int grep(int fd) {
7     /* ... more code */
8  +  /* from input (./harness.sh test BIN 1) */
9  +  /* POTENTIAL FLAW */
10 +  memcpy(lastout + 0, prog, out_byte);
11    reset(fd);
12    lastout=0;
```

(c) The diff resulting from injecting the bug template in 1b into grep. Here, `$dst → lastout`, `$src → prog`, `$num → out_byte`.

Fig. 1: A CSA regression test (1a), a corresponding bug template (1b) created manually, and the diff resulting from injection of this bug template into the grep program (1c).

predicate to search the dynamic traces for suitable injection locations: points in the trace that meet the precondition. The input that gives rise to a trace is called the "witness" of that trace. The buggy code injected into the source at the precondition-matching location will be executed when run with the witness input. In Figure 1b, the precondition specifies that at an injection point p, (a) an in-scope variable bound to `$dst` is a null pointer, and (b) another in-scope variable bound to `$num` has a value > 1.

The example bug template in Figure 1b was manually created based on an existing regression test (Figure 1a) for CSA. This regression test contains a bug at the call to `memcpy`: that its first argument is a null pointer. A successful injection of the bug template in Figure 1b will insert a call to `memcpy`, where (a) `$src`, `$dst`, and `$num` are replaced with host program variables, and (b) the variable bound to `$dst` is null and the variable bound to `$num` is > 1, before the call to `memcpy`. Thus, the bug injection attempts to create the same kind of bug, but embedded and integrated with the host program's data and control flow complexity. The null value of the host program variable bound to `$dst` comes from an existing sequence of host program events (i.e., it is not artificially generated): e.g., the pointer might have been set to null at a distant program

point in a different function, or its value copied from some other pointer which happens to be null under certain conditions triggered by the witness input.

Typically, creating a bug template from an example bug requires identifying (a) the relevant buggy code, (b) the free variables in the buggy code that must be re-bound in the host program, and (c) preconditions to ensure the bug is successfully transferred to the host program.
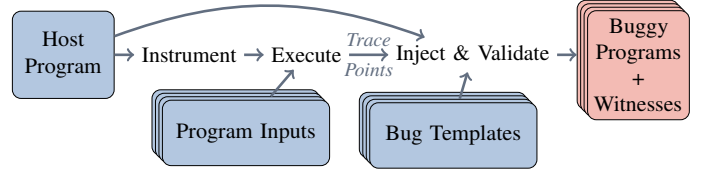
C. Technique

Fig. 2: BUG-INJECTOR pipeline.

The BUG-INJECTOR pipeline of *instrument*, *execute*, and *inject* is shown in Figure 2 and described in the algorithm in Figure 3. BUG-INJECTOR takes three inputs: (1) a host program, (2) a set of tests for this program, and (3) a set of bug templates. It attempts to inject bugs from the set of bug templates into the host program, and returns multiple different buggy versions of the host program. Each returned buggy program variant has at least one known bug (the one that was injected), and is associated with a *witness*—a test input which is known to exercise the injected bug.

The BUG-INJECTOR algorithm begins by instrumenting the host program (Figure 3, line 2). The Instrument method rewrites the source code of the host program, inserting code to emit dynamic trace output. Traces include the values of all in-scope variables (currently limited to primitive types and pointers) at every program statement. The algorithm then runs the instrumented program with test inputs. The collected

Input: Host Program, *Host*: *Program*
Input: Program Inputs, *Suite*: {*Test*}
Input: Bug Templates, *Templates*: {*BugTemplate*}
Parameters: *NumInjections*, *MaxTrace*
Output: Buggy Program Versions, *Bench*: {⟨*Program*, *Test*⟩}
 1: **let** *Bench* ← ∅, *TraceDB* ← ∅
 2: **let** *Inst* ← Instrument(*Host*)   ▷ Instrument
 3: **for** *Input* ∈ *Suite* **do**   ▷ Execute
 4:     *TraceDB* ← *TraceDB* ∪ Collect(*Inst*, *Input*, *MaxTrace*)
 5: **end for**
 6: **for** *Template* ∈ *Templates* **do**
 7:     *Candidates* ← Match(*Template.precondition*, *TraceDB*)
 8:     *Sampled* ← RandomSample(*Candidates*, *NumInjections*)
 9:     **for** ⟨*Points*, *Witness*⟩ ∈ *Sampled* **do**
10:         *Bugged* ← Inject(*Host*, *Points*, *Template*)   ▷ Inject
11:         **if** Validate(*Bugged*, *Witness*) **then**   ▷ Validate
12:             *Bench* ← *Bench* ∪ {(*Bugged*, *Witness*)}
13:         **end if**
14:     **end for**
15: **end for**
16: **return** *Bench*

Fig. 3: BUG-INJECTOR algorithm.

4

traces (capped by size *MaxTrace*) and the test inputs that produced them are stored efficiently in a persistent binary-format database, *TraceDB* (line 4).

BUG-INJECTOR then attempts to inject each of the bug templates *NumInjection* times. For every bug template, it uses Match to search *TraceDB* for candidate program point sets that correspondingly match the preconditions for all the patches in that bug template. Match returns a list of candidates: each candidate is a tuple of points—one program point per patch in the bug template—and a witness input. BUG-INJECTOR randomly samples *NumInjection* candidates for injection. The candidates picked for injection are then used by Inject (line 10), which takes the code in the patches of the bug template and rewrites the source code locations associated with each of the *Points*. Source rewriting involves inserting the associated code snippet into the host program, then renaming all the free variable names with the precondition-matching and type-compatible in-scope variables of the host program.

To validate an injection, BUG-INJECTOR adds instrumentation[4] to the modified program to dynamically check that the pre-conditions hold before the injected bug upon re-execution against *Witness* (line 11). The buggy program *Bugged* and its associated *Witness* are added to the output *Bench* (line 12). After exhausting the given number of injections, or when no more candidate injection points are available, *Bench* is returned.

As an example, consider the injection of bug template given in Figure 1b into the C program grep, resulting in the buggy variant of grep shown in Figure 1c. In this instance, BUG-INJECTOR uses host program's global static variables, lastout, prog, and out_byte, in the call to memcpy. For diagnostic purposes, the injected buggy code is optionally preceded by a comment that includes the input witness for the injected bug. When buggy grep is run with this input, the value of lastout is null before the call to memcpy, at least once during program execution. This injection successfully violates the "memcpy should not be called with its first argument being a null pointer" rule, but in a different code context.

CSA emits a warning about the bug in their regression test Figure 1a. However, CSA fails to report a warning for the similar injected bug in this buggy version of grep. CSA has "lost" this bug due to its injection into a more complex context.

D. Uses of BUG-INJECTOR

One of the applications of BUG-INJECTOR is to provide feedback to static analysis tool developers regarding the false-negative rates of their "checkers" on real-world programs. A typical workflow for building static analysis checkers[5] is an iterative process: (1) develop a checker to detect violations of a program property, (2) test the static analysis checker on some manually crafted test programs, (3) deploy the checker into production, (4) identify failures and false-negative corner

---

[4]The validation code is removed before adding it to the benchmark.

[5]This workflow is informed by the author's discussions with static analysis tool developers and by the static analysis checker development tutorial for Phasar [36].

cases for the checker, (5) iterate and improve the checker. BUG-INJECTOR can improve and accelerate this process. Instead of manually crafting test cases, we can craft relevant bug templates. BUG-INJECTOR can then generate checker benchmarks by injecting these bug templates into real-world programs. The static analysis checker can then be tested on the generated benchmarks to obtain early feedback regarding the checker's performance (such as expected false-negative rate, scalability), before deploying the checker into production.

Another application of BUG-INJECTOR is customized evaluation of static analysis tools, as we have done in §VI. We also provided the SAMATE group at NIST with BUG-INJECTOR. This group is conducting SATE VI [37]: the sixth iteration of Static Analysis Tool Exposition. SATE is a non-competitive study of static analysis tool effectiveness, aiming at improving tools and increasing public awareness and adoption. SATE VI is already making use of BUG-INJECTOR generated test programs, in addition to manually crafted test programs. Further, NIST is expecting to make extensive use of BUG-INJECTOR for SATE VII, the next iteration of SATE. To quote the initial experience of the NIST team with BUG-INJECTOR: "using BUG-INJECTOR to generate benchmarks is much faster (at least five times as fast) than using our current manual benchmark generation process." For SATE VI, the participating static analysis vendors can compare how well they perform on BUG-INJECTOR generated benchmarks vs. the manually created benchmarks, which will be a useful broader study regarding the effectiveness of BUG-INJECTOR. NIST also plans to add BUG-INJECTOR generated tests to the SARD dataset [14].

IV. ESTIMATING STATIC ANALYSIS RECALL

As previously discussed in §I, it is difficult to compute the exact recall of a tool. Thus, BUG-INJECTOR (as well as all other related work) estimates the recall of a static analysis tool using the set of known bugs in a given benchmark, which is a subset (possibly strict) of all the bugs actually present in that benchmark. The set of known bugs in a given benchmark is referred to as the *ground truth* for the benchmark. In this section, we discuss some practical issues in representing ground truth for the purposes of evaluating static analysis tools.

Ground truth accuracy. That is, each bug in the provided list must manifest in at least one execution of the program. LAVA [16] provides backtraces for each test case showing that the bugs included are real. EvilCoder [15], however, provides no such guarantees. BUG-INJECTOR benchmarks come with inputs which can generate dynamically-observed program states where the bug template preconditions are met. Hence, the guarantees provided by BUG-INJECTOR are relative to the correctness of the bug template specification.

Matching ground truth to tool output. Ground truth must include information such as location and bug type for each listed bug. This information allows automated or semi-automated matching of a tool's output with the ground truth. There are various pitfalls in providing this information: there may be multiple locations associated with a bug, multiple bug types

associated with the same bug, multiple bugs in the same location (depending on the granularity of the location), or lexically distinct languages used by tools to warn about the same bug type. Several recent studies [19], [18] elaborate on these problems.

Real-world bug distribution. BUG-INJECTOR gives us control over how many of each type of bug we inject. By injecting bugs of a type that are harder or easier for a given tool to detect, one can influence the measured recall of the tool on the generated benchmark. Unfortunately, it is difficult to know the real-world distribution of different bug types. This does not prevent the use of BUG-INJECTOR for comparing the relative recall of two tools on particular bug types of interest or between different settings of the same tool.

LAVA [16] injects only buffer overflows, so the bug type is known up front. Every test case includes a backtrace that showcases the bug. While this may be sufficient for evaluating fuzzers or manually inspecting static analysis results, it can be difficult to automate. For example, do you credit a tool with finding a bug only if it warns about the location at the top of the backtrace, or is it sufficient for it to warn about any location in the backtrace? Are there other relevant locations in the program that can be justifiably reported by static analysis tools? For the LAVA-1 dataset, we found empirically that key locations in the backtraces can be matched to invocations of the synthetic method `lava_get()` in the source code. Consequently, we interpret the ground truth to be the set of these locations.

For our BUG-INJECTOR benchmark, such additional ground truth information is implicit in the bug templates (which specify the bug type) and the locations where the injection was performed. As shown in the example in Figure 1, the injection location can be determined by examining the source code difference between the original and injected program.

A further hurdle to automation is that there is no standardized format for the output of a static analysis tool that all tools adhere to, and often no direct way to determine which specific bug a tool is reporting. In practice, the evaluator must typically rely on manually created heuristics that match the tool's reports with ground truth based on location and warning type. This approach has some limitations, notably the possibility of mistakenly failing to credit the tool with a true positive because it reports a slightly different but related bug, or because it reports the correct bug at a slightly different location. Adding some "tolerances" to the location heuristics, such as allowing a neighborhood of several lines of code around the expected bug location, can mitigate this problem but may cause its own issues if the tool detects unrelated bugs in the neighborhood. In our experimental evaluation § VI-B, we explicitly discuss how we credit tools for finding appropriate bugs in our benchmarks.

V. OUR BENCHMARK SUITES

In this section, we describe two BUG-INJECTOR-generated benchmark suites. Both these benchmark suites, and the bug templates used to generate them, are available online for use by the community [38]. We plan to maintain a library of bug templates that can be used for different user-chosen evaluations.

A. Selection of host programs

TABLE II: Host programs used for evaluation. LOC gives the lines of code in the programs. The rest of the columns are described in § V-C.

Project	Version	LOC	Prep Time	Query Time	Sites/ KLOC
grep [39]	2.0	12K	66	1.76	372.76
nginx [40]	1.13.0	178K	766	5.03	7.62

We use the open-source projects listed in Table II as the host programs for generating our benchmark suites. We have successfully injected bugs into other C/C++ host programs (total of 15 real-world programs to date), but we have not included them in this paper due to resource constraints in running experiments (§ VI). One such excluded program is WireShark version 1.12.9, which has 2.3 million lines of code: it is the largest program we have successfully injected bugs into. This demonstrates BUG-INJECTOR's ability to inject into a variety of real-world projects. An important criteria for picking host programs is the availability of test suites with good code coverage: they provide a large number of distinct trace points for BUG-INJECTOR, improving the chances of finding many suitable injection points by matching preconditions.

B. Selection of bug templates

Bug Template Source	No. of Templates	mean counts		
		LOC	FVars	CF Stmts
CSA [41]	10	3.1	0.8	0.2
Infer [42], [43]	6	4.2	0.8	0.8
Juliet tests [2]	55	7.8	1.3	0.9

TABLE III: The number of bug templates from each source. The last three columns provide the means over each set of templates for: (a) the number of lines of code to be injected, (b) the number of free variables to be rebound, and (c) the number of control-flow statements in the injected code, respectively.

We create bug templates from three sources (shown in Table III) to satisfy two different goals. First, we want our benchmark suite to allow a fair evaluation of CSA [21] and Infer [22], and inject bug types that these tools care about and are expected to find. Both tools support the detection of buffer overflows (BO) and null pointer dereferences (NPD). Therefore, we collect examples of BO and NPD bugs that appear in these tool's documentation [41], [42] and regression test suites [43]. We manually verified that each example contains the bug they claim to contain, and then converted the example to a bug template. We also checked that at least one tool warns on each example bug snippet. The manual conversion of a bug example to a bug template is fairly straightforward (described in § III-B), and only took on the order of few minutes per example. Each of the 16 bug templates collected from CSA

and Infer are injected upto 30 times into each of the two host programs in Table II, to create benchmark suite **B1**, with a total of 591 program variants. Note that a bug template may have been injected fewer than 30 times into a host program because of insufficient number of precondition-matching locations or failed validation. In **B1**, each of the 16 bug templates has been injected at least once. **B1** is used for answering the research questions (§ VI-A) **RQ1**, **RQ2**, and **RQ3**.

Second, we want to demonstrate that BUG-INJECTOR can inject a wide variety of bug types and CWE categories [20]. To this end, we *automatically* converted 55 bug examples from the Juliet test suite (version 1.3. [2]) into bug templates; these bug examples span 55 unique CWE types, from stack-based buffer overflows (CWE-121) to type confusion (CWE-843). We exploited the uniform structure of Juliet tests to automatically create these bug templates: we extract free variables, preconditions, and code to inject from the Juliet test suite using both static and dynamic information from each bug example. We created the benchmark suite **B2**, which contains 2,492 program variants, by injecting each of the 55 bug templates sourced from Juliet tests upto 30 times into each of the host programs. Each of the 55 bug templates has been injected at least once. Many bug types in Juliet tests are out of scope for CSA and Infer, therefore we do not evaluate these tools on **B2** in this paper. **B2** serves to answer **RQ4**.

The program variants with bugs are uniformly formatted using a code beautification tool, ensuring the injection does not stand out due to code-style differences. As shown in Table III, the bug templates typically include a small amount of code. These characteristics, along with the use of existing program variables (through free variable rebinding), allow the injections to meld with the existing code and look realistic (e.g., see Figure 1c, or examine any of the benchmark programs).

C. Performance of BUG-INJECTOR

As discussed in § III, BUG-INJECTOR operates in a pipeline of several stages. Performance in these stages depends on characteristics of the host program, its tests, and the bug template set. Table II summarizes the key characteristics and performance data for the host programs. Timing experiments were performed on an Intel(R) Xeon(R) 2.10 GHz machine with 72 cores and 128 GB of RAM.

In the *instrument* and *execute* stages, BUG-INJECTOR parses the host program, adds instrumentation, and runs the program with test inputs to collect traces. The time required for this stage depends on the size of the program, the number of variables it contains, and the number of input tests to run; the "Prep Time" column in Table II, given in seconds, provides this information for each host program. This provided prep time is a one time cost, which gets amortized over the number of bugs to be injected into the same host program.

The *inject* stage involves searching the trace database for points satisfying the bug template preconditions. The time required per injection depends on the number of points collected in the trace, the percentage of points which satisfy the precondition and free variable requirements, as well as the

complexity of the precondition. The "Query Time" column gives the median time (in seconds) per query. The "Sites/KLOC" column in Table II provides the number of matching host-program sites that are suited for injection based on our bug templates, per 1000 lines of code. The `grep` program contained a large number of string and integer variables, and therefore showed higher density of potential injection sites; conversely, `nginx`, with few integer variables, had lower density of injection sites.

Lastly, BUG-INJECTOR edits the program, applies code formatting to the buggy software, and writes it out to disk. The time required to apply code formatting and printing the buggy program is directly proportional to the program size. Overall, the prep time dominates the pipeline as the most expensive stage. Given the offline and automatic nature of benchmark creation, we believe the performance of BUG-INJECTOR is reasonable.

VI. EVALUATION

In this section, we outline the research questions that direct our evaluation, describe our experimental methodology, report and discuss the results of our experiments, and compare our benchmark with the LAVA test cases [23].

A. Research questions

The goal of our evaluation is to answer the following research questions about BUG-INJECTOR and its generated benchmarks.

RQ1: *Do the benchmarks contain bugs which are seemingly in scope for the tool but which the tool fails to detect?* Such bugs could provide useful concrete feedback to the tool's developers.

RQ2: *Can the benchmarks discriminate between different static analysis tools?*

RQ3: *Can the benchmarks discriminate between different parameter settings for a given static analysis tool?* Such an ability suggests the use of BUG-INJECTOR for automated tuning of a tool's parameters specific to a given codebase.

RQ4: *Can BUG-INJECTOR create benchmarks that include bugs from multiple CWEs?* Such an ability shows whether the technique is applicable to multiple bug types.

In addition to answering the above research questions, we also compare BUG-INJECTOR with the LAVA test suite with respect to the same research themes.

B. Experimental setup and methodology

Static analysis tools and configurations. We perform our experiments using two open-source state-of-the-art static analysis tools for C/C++ programs: Clang Static Analyzer (CSA) [21] and Infer [22]. We use CSA version 3.8[6], and run the tool on Ubuntu 16.04. We use CSA with the analyzer configuration mode set to "shallow" (CSA-S), as well as the default mode "deep" (CSA-D). CSA-S mode changes certain default analysis parameters, such as the style of the inter-procedural analysis and maximum inlinable size. We use the

---

[6]This is the default version available on Ubuntu 16.04.

term CSA to refer to both modes. CSA is run with all the default checkers enabled, along with the optional `alpha`, `security`, `osx`, `llvm`, `nullability`, and `optin` checkers.

We use Infer version 0.14, and run it from the tool's official Docker image. Infer is run with default options and `compute-analytics`, `biabduction`, `quandary`, and `bufferoverrun` enabled. For both CSA and Infer, our intention is to enable as many checkers as possible to maximize the tool's chance of finding the injected bugs.

Projected recall. This metric computes the percentage of the BUG-INJECTOR injected bugs found by a tool. We report this value by rounding to a whole number percentage. To determine whether a tool found an injected bug successfully, as discussed in §IV, we consider the locations of the bug injection as the bug locations. We credit a tool with finding an injected bug if it reports a bug of an appropriate type on at least one of the injected code lines. The table below summarizes which tool-specific bug types (cell contents) reported by the tools (row headers) are considered to correspond to the injected bug types (column headers). We interpret the bug types reported by the tools quite generously, to maximize their chances of being credited with finding the injected bugs.

	Buffer overrun (BO)	Null pointer dereference (NPD)
CSA	Out of bound array access, Result of operation is garbage or undefined, `malloc()` size overflow	Dereference of null pointer, Uninitialized argument value, Argument with 'nonull' attribute passed null
Infer	Array out of bounds, Buffer overrun, Memory leak, Stack variable address escape	Array out of bounds, Buffer overrun, Dangling pointer dereference, Null dereference, Memory leak

C. Experiments and results

To help answer research questions **RQ1**, **RQ2**, and **RQ3**, we run CSA-S, CSA-D, and Infer, on benchmark **B1** (described in §V). Tables IV and V provide the projected recall of the tools on various partitionings of **B1**.

The two tables provide different views of the same experimental results. Table IV partitions the results by bug template source (Table III) and host program (Table II). The last row provides results on the entire **B1** benchmark. Table V partitions the results by injected bug type (NPD or BO) and host program. The "No. of Bugs" column in both these tables describes the number of benchmark programs–each containing one known bug—in the specified partition. The rightmost six rows in both the tables provide the projected recall of the tools on the given benchmark partition. The highest projected recall in each row is highlighted.

A tool can obtain higher projected recall by simply reporting more warnings overall, which will increase its chances of also reporting an injected bug. A tool could also take a lot longer than is acceptable to a user to report bugs. Therefore, it is instructive to look at two additional metrics: "Warnings per KLOC" and "Time taken". Table VI reports these metrics.

The columns under "Warnings per KLOC" in Table VI provide the average number of total warnings reported by the

TABLE IV: Projected recall of different tools for various pairs of bug template sources and host programs. Last row summarizes the results over the entire benchmark suite **B1**.

Bug Template Source	Host Program	No. of Bugs	CSA-S	CSA-D	Infer
CSA	grep	251	88%	69%	45%
	nginx	122	92%	92%	52%
Infer	grep	179	36%	37%	50%
	nginx	39	69%	69%	18%
Both	Both	591	72%	64%	46%

TABLE V: Projected recall of different tools for various pairs of bug types and host programs.

Bug Type	Host Program	No. of Bugs	CSA-S	CSA-D	Infer
NPD	grep	98	82%	67%	15%
	nginx	60	90%	90%	20%
BO	grep	332	61%	53%	57%
	nginx	101	84%	84%	58%

tools for every thousand lines of code, on the benchmark suite **B1**. The number provided after the ± symbol is the standard deviation (rounded to one decimal place) over all variants of that host program. This metric can be helpful to check that a tool is not reporting so many warnings on real-world programs that it is effectively unusable. Note that comparing this metric directly between two analysis tools which do not have comparable warning classes (such as Infer vs. CSA) is not particularly meaningful. On "grep", CSA-D reports more total warnings than CSA-S, whereas, on "nginx", CSA-S reports more total warnings than CSA-D.

The columns under "Time taken" in Table VI provide the average time taken by the tool to run on a given host program. We run each tool five times on a four-core Intel(R) Xeon(R) 2.10Ghz machine with 16GB RAM and report the average. CSA-S runs much faster than CSA-D.

Addressing RQ1. The benchmark suite **B1** consists of the bugs that CSA and Infer care about injected into popular open-source programs (representing the expected targets of the chosen static analysis tools). Tables IV and V show that both the tools detect some but not all of the injected bugs.

If a tool reports a bug on a small example with a simple context, we might expect that the tool also reports a similar

TABLE VI: Total warnings reported per KLOC and average time taken by the tools.

Host	Warnings per KLOC			Time taken (seconds)		
	CSA-S	CSA-D	Infer	CSA-S	CSA-D	Infer
grep	7.9±.2	9.7±.2	1.9±.1	14.7	41.1	20.5
nginx	6.8±.0	2.8±.0	0.8±.1	229.6	366.1	338.7

bug in a more complex setting. However, in the case of both CSA and Infer—the leading open-source static analysis tools for C/C++—we find that they "lose" bugs (projected recall is not 100%) across all rows in Tables IV and V. That is, CSA and Infer find bugs in their respective documentation examples and regression tests, but in many cases they lose the ability to find the "same" bug when it is injected and integrated into a larger program. These "lost" bugs can represent concrete feedback for the analysis tool developers.

Addressing RQ2. Tables IV and V show that our generated benchmarks can be used for contrasting the projected recall of the evaluated tools. Depending on the specific subset of the benchmark suite (i.e., table row) that is of interest to the evaluator, different tools have higher projected recall. Thus, BUG-INJECTOR can be used to perform tool evaluations to suit specific customer needs by providing control over the distribution of bug templates and host programs. CSA-S has the highest recall on benchmark suite **B1** as a whole.

Addressing RQ3. Static analysis tools are typically configurable, with the chosen configuration affecting tool recall, precision, and scalability. There is generally no single best configuration: it depends on several factors including the codebase being analyzed, the warning classes that are of interest, etc. To evaluate how our generated benchmarks discriminate different configurations of the same tool, we examine two configurations of Clang Static Analyzer: CSA-D and CSA-S. In a majority of cases in **B1**, CSA-S has equal or higher projected recall compared to CSA-D, while also being significantly faster. This is a surprising result that may be of interest to CSA users and developers.

In this paper, we only compare two configuration points of CSA. However, CSA (and many other analysis tools) have several configuration parameters. Projected recall from BUG-INJECTOR generated tests can be used (in conjuction with other metrics of interest) to tune the settings of these parameters for a given codebase.

Addressing RQ4. BUG-INJECTOR is able to generate the benchmark **B2**, which contains injected bugs corresponding to 55 different CWEs, based on bug templates sourced from Juliet test suite version 1.3. This artifact shows that BUG-INJECTOR can be used to inject a wide-variety of bug types.

D. Causes of lost bugs

A large number of injected bugs are "lost" by the evaluated tools (ranging between 28% to 54% lost bugs per tool). An extensive study of all lost bug cases by each tool is out of scope for this work. Instead, we sampled a small number of randomly-selected lost bugs to manually check whether there were particular patterns or language constructs that were causing the tools to lose track of the bugs. However, we found no single dominant pattern for the lost bugs: there seems to be a long tail of several issues that cause tools to lose bugs. In our limited study, we see that each lost bug belongs to one of three categories:

needs-fix: the tool needs to be fixed for the bug to be found

param: adjusting the tool's parameters can find the bug

expected: the bug is lost by design

Below, we discuss some simplified examples of lost bugs in **B1**. We mark each discussed bug with our diagnosis with respect to the above categories. We have reported [44], [45], [46] some of these lost bugs to the analysis developers.

Infer fails to report [45] any warnings in a function that has enum declarations of the following form (**needs-fix**):

```
1    enum { L, R } dirs[12];
```

Infer fails to report [44] the null pointer dereference in this simple case (**needs-fix** or **param**):

```
1    int *nullable; int *firstpos; int *lastpos;
2    int * buf = 0;
3    nullable = malloc(2*sizeof(int));
4    firstpos = malloc(2*sizeof(int));
5    lastpos = malloc(2*sizeof(int));
6    for (int i = 0; i < 3; i++) { /* no-op */ }
7    *buf = 1; // null pointer dereference
8    nullable++; firstpos++; lastpos++;
```

Infer fails to report any bugs present in the source code of those functions for which library models exist (**expected**). The source code of such functions are ignored. This behavior may result in supply-chain attacks going unnoticed by the tool.

CSA fails to report [46] a buffer overrun in the presence of an intervening function call, presumably due to unsound early termination in the tool's path exploration (**needs-fix**).

Many of the bugs lost by CSA can be found by tuning the analysis parameters (**param**). E.g., high values of the parameter -maxloop, which controls the number of times a block can be visited before giving up, finds many lost bugs.

Thus, BUG-INJECTOR generated benchmarks can expose the various real-world scenarios in which an analysis tool can fail to report a bug, which is of interest to both analysis users and analysis developers.

E. Comparison with LAVA benchmarks

We run CSA and Infer tools on the LAVA-1 benchmarks. The LAVA-1 benchmarks consist of 69 variations of the file program, with each variant having one injected buffer overflow bug. As discussed in §IV, we stipulate for the sake of this evaluation that the bug location is the line consisting of a lava_get() call, and give a tool credit for identifying the bug if it specifies a location within 5 lines of this. In all LAVA-1 test cases, we found that the lava_get() call location matched the first location provided in the corresponding backtrace included with the LAVA corpus.

CSA reports between 41 and 51 warnings on each of the LAVA-1 benchmarks. In 58 of the 69 programs, CSA does not report on any LAVA-injected bugs. In the remaining 11 programs, CSA issues warnings at the injected bug locations. Upon manual inspection of each of these examples, we determined these warnings to be unrelated to buffer overflows.[7]

---

[7]The reported warnings were one of: "pointer of type void* used in arithmetic", "nested extern declaration of vasprintf", "implicit declaration of function vasprintf", "pointer arithmetic on non-array variables relies on memory layout: which is dangerous".

```
1  /* inside a for loop */
2  if (ml->map)
3    apprentice_unmap((ml->map))+(lava_get())*((0x12345678
4      <= (lava_get()) && 0x123456f8 >= (lava_get())) ||
5      (0x12345678 <= __bswap_32((lava_get())) && 0x123456f8
6      >= __bswap_32((lava_get())))));
7  free(ml);
```

Fig. 4: Example LAVA-injected bug.

Infer reports between 16 and 18 warnings on each of the LAVA-1 benchmarks. However, none of the Infer warnings are at the LAVA-injected bug locations.

To summarize, both CSA and Infer report warnings on the LAVA-1 benchmarks, but none of these are related to the LAVA-injected bugs. Thus, the projected recall of both of these tools is 0% on the LAVA-1 benchmarks. This result is not particularly surprising, as LAVA is biased towards testing the limits of fuzzing tools, and injects code that looks like the snippet in Figure 4. Such bugs would typically be out of scope for accurate reasoning by most static analysis tools, as the tools have to make static approximations and/or heuristic choices that balance precision, recall, and scalability. These results—that leading open-source static analysis tools have **zero** projected recall—indicate that LAVA benchmarks are not well-suited for discriminating between different static analysis tools (refer **RQ2**), or that they include bugs that are in scope for the evaluated static analysis tools (refer **RQ1**). Also, LAVA can only inject a very small number of bug kinds (refer **RQ4**).

Therefore, while LAVA has been successful in advancing fuzzing techniques [47] and helping create capture-the-flag-style competitions [28], it is less relevant in evaluating static analysis tools.

VII. LIMITATIONS AND FUTURE WORK

BUG-INJECTOR currently chooses an injection point in the host program uniformly at random from all the dynamic trace points that match the bug template's preconditions. Thus, host program points that are exercised more frequently by the accompanying tests are more likely to be used for injection, as they appear more frequently in the dynamic traces. BUG-INJECTOR can be combined with coverage-increasing input-generation techniques like concolic testing [48] to obtain an improved program-wide distribution of injected bugs.

BUG-INJECTOR does not currently support the injection of concurrency-related bugs. We plan to add such support. Our first step will be to improve instrumentation so that concurrency-related information such as the current thread and process is available in the trace.

BUG-INJECTOR cannot always inject a bug template into a host program, because there is not always a dynamic trace point that matches all the preconditions and free-variable requirements for the template. To increase the chances of finding injection points in a host program, we plan to enhance BUG-INJECTOR to allow for variable rebinding to aggregate structs and fields.

We envision running BUG-INJECTOR's pipeline multiple times in an evolutionarily-guided heuristic search. This process would allow injection of multiple bugs into a single host program, maximizing an objective function that balances factors such as number of injected bugs, naturalness of code [49], realistic distribution of bugs [50], [51], retention of the original program behavior, and syntactic/stylistic similarity [52] between the buggy program and the original program. BUG-INJECTOR is built using SEL, which supports evolutionary search with multi-objective fitness functions. Leveraging this support, we have early prototypes that fulfill this vision.

Regarding our experimental methodology, the main threat to validity relates to how we measure whether a tool finds a specific injected bug, both for BUG-INJECTOR and LAVA test cases. As we explain in §IV and §VI, we use simple heuristics to match the location contained in the tool's warning with the location of the known bug and determine that the correct bug has been identified if the bug types match and the locations are within a certain maximum distance. We could refine this heuristic by using more sophisticated matching techniques from related work on the issue of deduplicating and/or clustering tool warning reports [53], [54].

VIII. CONCLUSION

In this paper, we introduce BUG-INJECTOR, a system that automatically generates bug-containing benchmarks suitable for evaluating and testing software analysis tools. BUG-INJECTOR operates by injecting bug templates into real-world programs, and is able to create custom benchmarks that are real-world-like, can draw from a wide variety of bug types, and come with bug-triggering inputs. Our experimental evaluation shows that BUG-INJECTOR benchmarks are useful for several purposes: (a) showcasing bugs that are seemingly in scope for a tool to find but that the tool misses, (b) discriminating between and guiding the improvement of static analysis tools, and (c) tuning tool parameters for a specific codebase. We also show that BUG-INJECTOR can create bugs from multiple CWEs.

REFERENCES

[1] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, 2015.

[2] P. E. Black, "Juliet 1.3 Test Suite: Changes From 1.2," in *National Institute of Standards and Technology (NIST) Technical Note (TN) 1995*, June 2018.

[3] W. Vanderlinde, "Securely taking on new executable software of uncertain provenance (STONESOUP)," http://www.iarpa.gov/index.php/research-programs/stonesoup.

[4] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 12–15.

[5] "OWASP WebGoat Project," https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, accessed 2018-05-01.

[6] J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *Nordic Workshop on Secure IT Systems (NordSec)*, Karlstad, Sweden, 2002/11/07/November 7 2002, pp. 68–84, karlstad, Sweden.

[7] Wilander, John and Kamkar, Mariam, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Symposium on Network and Distributed System Security (NDSS)*. The Internet Society, February 2003, pp. 149–162.

[8] T. Newsham and B. Chess, "Abm: A prototype for benchmarking source code analyzers," in *Workshop on Software Security Assurance Tools, Techniques, and Metrics. US National Institute of Standards and Technology (NIST) Special Publication (SP)*, 2006, pp. 500–265.

[9] C. Timperley, S. Stepney, and C. Le Goues, "Poster: BugZoo – A Platform for Studying Software Bugs," in *International Conference on Software Engineering*, ser. ICSE, 2018.

[10] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.

[11] G. Nilson, K. Wills, J. Stuckman, and J. Purtilo, "BugBox: A vulnerability corpus for PHP web applications," in *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/cset13/workshop-program/presentation/Nilson

[12] "Stanford SecuriBench," https://suif.stanford.edu/~livshits/securibench/, accessed 2018-05-01.

[13] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 97–106.

[14] NIST, "SARD test cases," https://samate.nist.gov/SRD/testsuite.php, accessed 2018-05-27.

[15] J. Pewny and T. Holz, "EvilCoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 214–225.

[16] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 110–121.

[17] 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: ACM, 2018, pp. 224–234.

[18] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders," in *First International Workshop on Complex faUlts and Failures in LargE Software Systems (COUFLESS)*, 2015.

[19] J. Herter, D. Kastner, C. Mallon, and R. Wilhelm, "Benchmarking static code analyzers," in *Computer Safety, Reliability, and Security*. Cham: Springer International Publishing, 2017, pp. 197–212.

[20] "Common Weakness Enumeration - a community-developed list of software weakness types." https://cwe.mitre.org/, accessed 2018-04-24.

[21] "Clang static analyzer," https://clang-analyzer.llvm.org/, accessed 2018-05-21.

[22] "Infer," http://fbinfer.com/, accessed 2018-05-21.

[23] "LAVA synthetic bug corpora," http://moyix.blogspot.com/2016/10/the-lava-synthetic-bug-corpora.html, accessed 2018-05-31.

[24] 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 *International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 339–349. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00048

[25] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015, DOI: 10.1109/TSE.2015.2454513.

[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis*. ACM, 2014.

[27] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE. ACM, 2018.

[28] P. Hulin, A. Davis, R. Sridhar, A. Fasano, C. Gallagher, A. Sedlacek, T. Leek, and B. Dolan-Gavitt, "Autoctf: creating diverse pwnables via automated bug injection," in *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[29] "The Heartbleed Bug," http://heartbleed.com/, accessed 2018-07-11.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, April 1978.

[31] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279–290, July 1977.

[32] M. Christakis, P. Emmisberger, P. Godefroid, and P. Müller, "A general framework for dynamic stub injection," in *"International Conference on Software Engineering (ICSE)"*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 586–596. [Online]. Available: https://doi.org/10.1109/ICSE.2017.60

[33] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 379–388.

[34] E. Martins, C. M. F. Rubira, and N. G. M. Leme, "Jaca: a reflective fault injection tool based on patterns," in *International Conference on Dependable Systems and Networks*, June 2002, pp. 483–487.

[35] E. Schulte and Contributors, *Software Evolution Library*, GrammaTech, eschulte@grammatech.com, 1 2018, https://grammatech.github.io/sel/.

[36] "Static Analysis for C++ with Phasar," http://phasar.org/wp-content/uploads/2018/06/phasar_block_2-3.pdf, slide 37, Accessed 2018-06-30.

[37] NIST, "SATE: Static Analysis Tool Exposition," https://samate.nist.gov/SATE.html.

[38] V. Kashyap, J. Ruchti, L. Kot, E. Turetsky, R. Swords, S. A. Pan, J. Henry, D. Melski, and E. Schulte, "Bug Injector Benchmarks for IEEE SCAM 2019," Jul. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3341585

[39] "Gnu grep," http://www.gnu.org/software/savannah-checkouts/gnu/grep/, accessed 2018-05-28.

[40] "nginx," http://www.nginx.com/, accessed 2018-05-28.

[41] "Clang static analyzer: Available checkers," http://clang-analyzer.llvm.org/available_checks.html/, accessed 2018-05-28.

[42] "Inferbo: Infer-based buffer overrun analyzer," http://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/, accessed 2018-05-28.

[43] "Infer: Regression tests," http://github.com/facebook/infer/tree/master/infer/tests, accessed 2018-05-28.

[44] "Infer bug report," https://github.com/facebook/infer/issues/1093, 2019.

[45] "Infer bug report," https://github.com/facebook/infer/issues/1094, 2019.

[46] "Clang static analyzer bug report," https://bugs.llvm.org/show_bug.cgi?id=41833, 2019.

[47] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[48] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*, 2007.

[49] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[50] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *International Symposium on Software Testing and Analysis*, 2002, pp. 55–64.

[51] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, 2000.

[52] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 255–270.

[53] Z. P. Fry and W. Weimer, "Clustering static analysis defect reports to reduce maintenance costs," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct 2013, pp. 282–291.

[54] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct 2016, pp. 157–166.