FuzzFactory: Domain-Specific Fuzzing with Waypoints

ROHAN PADHYE, University of California, Berkeley, USA
CAROLINE LEMIEUX, University of California, Berkeley, USA
KOUSHIK SEN, University of California, Berkeley, USA
LAURENT SIMON, Samsung Research America, USA
HAYAWARDH VIJAYAKUMAR, Samsung Research America, USA

Coverage-guided fuzz testing has gained prominence as a highly effective method of finding security vulnerabilities such as buffer overflows in programs that parse binary data. Recently, researchers have introduced various specializations to the coverage-guided fuzzing algorithm for different domain-specific testing goals, such as finding performance bottlenecks, generating valid inputs, handling magic-byte comparisons, etc. Each such solution can require non-trivial implementation effort and produces a distinct variant of a fuzzing tool. We observe that many of these domain-specific solutions follow a common solution pattern. In this paper, we present FuzzFactory, a framework for developing domain-specific fuzzing applications without requiring changes to mutation and search heuristics. FuzzFactory allows users to specify the collection of dynamic domain-specific feedback during test execution, as well as how such feedback should be aggregated. FuzzFactory uses this information to selectively save intermediate inputs, called waypoints, to augment coverage-guided fuzzing. Such waypoints always make progress towards domain-specific multi-dimensional objectives. We instantiate six domain-specific fuzzing applications using FuzzFactory: three re-implementations of prior work and three novel solutions, and evaluate their effectiveness on benchmarks from Google's fuzzer test suite. We also show how multiple domains can be composed to perform better than the sum of their parts. For example, we combine domain-specific feedback about strict equality comparisons and dynamic memory allocations, to enable the automatic generation of LZ4 bombs and PNG bombs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: fuzz testing, domain-specific fuzzing, frameworks, waypoints

ACM Reference Format:
Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (October 2019), 29 pages. https://doi.org/10.1145/3360600

1 INTRODUCTION

Fuzz testing is a popular technique for discovering security vulnerabilities, such as buffer overflows, in programs that parse binary data. *Fuzz testing* in general refers to the random generation of test inputs. However, the *coverage-guided* fuzz testing (CGF) algorithm has gained particular prominence recently. CGF maintains a continuously evolving set of saved inputs, starting with a set of known seed inputs. In each fuzzing round, CGF selects a saved input and randomly mutates it to generate

Authors' addresses: Rohan Padhye, EECS Department, University of California, Berkeley, USA, rohanpadhye@cs.berkeley.edu; Caroline Lemieux, EECS Department, University of California, Berkeley, USA, clemieux@cs.berkeley.edu; Koushik Sen, EECS Department, University of California, Berkeley, USA, ksen@cs.berkeley.edu; Laurent Simon, Samsung Research America, USA, l.simon@samsung.com; Hayawardh Vijayakumar, Samsung Research America, USA, h.vijayakuma@samsung.com.

2475-1421/2019/10-ART174
https://doi.org/10.1145/3360600

a new input. It then executes the program under test with this mutated input. CGF uses lightweight program instrumentation to gather feedback about test execution, such as the path taken through the program's control-flow graph. Like purely random fuzzing, if a mutated input causes a crash, it is saved for bug triaging. However, the core innovation of the coverage-guided testing algorithm is that if the mutated input leads to new code coverage, it is saved for use in subsequent rounds of fuzzing as the base for mutation. CGF has been popularized by tools such as AFL [Zalewski 2014] and libFuzzer [LLVM Developer Group 2016], which have found hundreds of security vulnerabilities in applications such as media players, web browsers, servers, compilers, and widely used libraries.

Recent work has shown that fuzz testing has applications beyond finding program crashes. For example, fuzz testing can be used for directed testing [Böhme et al. 2017], property-based testing [Padhye et al. 2019a], differential testing [Petsios et al. 2017a], side-channel analysis [Nilizadeh et al. 2019], discovering algorithmic complexity vulnerabilities [Petsios et al. 2017b], discovering performance hot spots [Lemieux et al. 2018], etc. In each case, researchers modified the original fuzzing algorithm to produce a specialized solution. Similarly, researchers have tweaked the original CGF algorithm to leverage domain-specific information from programs in order to improve code coverage, such as the use of magic bytes in file formats [LafIntel 2016; Li et al. 2017; Rawat et al. 2017] or measures of input validity [Laeufer et al. 2018; Padhye et al. 2019c; Pham et al. 2018].

Currently, the practice of developing domain-specific fuzzing applications is quite ad-hoc. For every new domain, researchers must find a way to tweak the fuzzing algorithm and produce a new variant of AFL or some other fuzzing tool. Each such solution can require non-trivial implementation. Further, these variants are independent and cannot be easily composed.

In this paper, we present FuzzFactory, a framework for implementing domain-specific fuzzing applications. Our framework is based on the following observation: many domain-specific fuzzing problems can be solved by augmenting the coverage-guided fuzzing algorithm to selectively save newly generated inputs for subsequent mutation, beyond those that only improve code coverage. We call these intermediate inputs *waypoints*, inspired by the corresponding term in the field of navigation. These waypoints give the fuzzing algorithm steps towards a domain-specific goal. A *domain-specific fuzzing application* for domain d is specified via a predicate: $is\_waypoint(i, \mathcal{S}, d)$. This predicate answers the following question: given a newly generated input i and a set of previously saved inputs $\mathcal{S}$, should we save input i to $\mathcal{S}$? FuzzFactory provides a simple mechanism for defining *is_waypoint*, based on *domain-specific feedback* that can be dynamically collected during test execution. A domain-specific fuzzing application can instrument programs under test to collect such custom feedback via a small set of APIs provided by FuzzFactory.

FuzzFactory enables development of domain-specific fuzzing applications without requiring changes to the underlying search algorithm. We were able to easily re-implement three algorithms from prior work and evaluate their strengths and weaknesses: SlowFuzz [Petsios et al. 2017b], PerfFuzz [Lemieux et al. 2018], and validity fuzzing [Padhye et al. 2019c]. We also used FuzzFactory to prototype three novel applications: for smoothing hard comparisons, for generating inputs that allocate excessive amounts of memory, and to perform incremental fuzzing following code changes. We describe these six domain-specific fuzzing applications as well as our experimental results on six real-world benchmark programs from a test suite released by Google [2019b].

A key advantage of FuzzFactory is that domain-specific feedback is naturally composable. We combine our domain-specific fuzzing applications for exacerbating memory allocations and for smoothing hard comparisons to produce a composite application that performs better than each of its constituents. The composite application automatically generates LZ4 bombs and PNG bombs: tiny inputs that lead to dynamic allocations of 4GB in `libarchive` and 2GB in `libpng` respectively.

To summarize, we make the following contributions in this paper:

Algorithm 1 The coverage-guided fuzzing algorithm

Input: an instrumented test program p, a set of initial seed inputs S_0
Output: a corpus of automatically generated inputs $\mathcal{S}$

```
1:  S ← S_0
2:  totalCoverage ← ∅
3:  repeat                                                    ▷ Main fuzzing loop
4:    for i in S do
5:      if sample fuzzProb(i) then
6:        i' ← mutate(i)                                      ▷ Generate new test input i'
7:        coverage ← execute(p, i')                           ▷ Run test with new input i'
8:        if coverage ∩ totalCoverage ≠ ∅ then
9:          S ← S ∪ {i'}                                      ▷ Save i' if new code coverage achieved
10:         totalCoverage ← totalCoverage ∪ coverage
11: until given time budget expires
12: return S
```

(1) We present FuzzFactory, a framework for specifying domain-specific fuzzing applications using custom feedback collected dynamically during test executions.
(2) We describe a domain-specific fuzzing algorithm that incorporates custom feedback as well as user-provided reducer functions to selectively save intermediate inputs, called waypoints.
(3) We identify key properties that reducer functions must satisfy in order to guarantee that every saved waypoint contributes towards domain-specific progress.
(4) We describe the implementation of six domain-specific fuzzing applications implemented using our framework, along with results of our experimental evaluation of these applications on six real-world test programs.
(5) We describe how to combine multiple domain-specific fuzzing applications and empirically show how such combinations can perform better than their constituents.
(6) We describe the API provided by our domain-specific fuzzing framework, FuzzFactory, and make the the tool publicly available at https://github.com/rohanpadhye/fuzzfactory.

2 BACKGROUND AND MOTIVATION

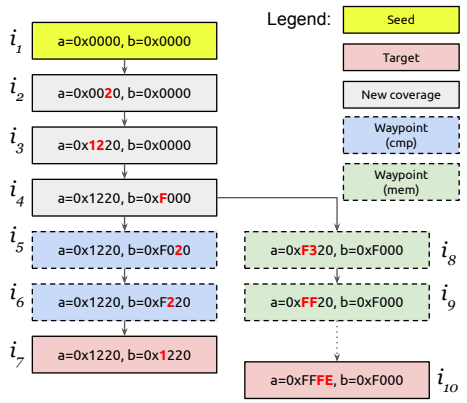
2.1 Coverage-Guided Fuzzing

In recent years, coverage-guided fuzzing (CGF) has emerged as one of the most effective techniques for fuzzing real-world software. CGF has been implemented in several popular tools including AFL [Zalewski 2014] and libFuzzer [LLVM Developer Group 2016]. CGF works by executing a test program with a large number of randomly generated inputs. Instead of generating totally random inputs from scratch, CGF selects a set of previously generated inputs and mutates them to derive new inputs. The high-level pseudo-code of CGF is shown in Algorithm 1.

The CGF algorithm takes an instrumented program and a set of user-provided *seed inputs*. CGF maintains two global states: (1) $\mathcal{S}$ which maintains a set of saved inputs to be mutated by the algorithm, and (2) *totalCoverage* which tracks the cumulative coverage of the program on the inputs in $\mathcal{S}$. CGF could track any kind of coverage; in practice, branch coverage or basic block transition coverage are most commonly used. $\mathcal{S}$ is initialized to the set of user-provided seed inputs and *totalCoverage* is initialized to the empty set. The main fuzzing loop of CGF goes over the set of inputs, selecting an input i from the set $\mathcal{S}$. With some probability determined by an implementation-specific heuristic function fuzzProb(i), CGF decides whether to mutate the input i

```
1  void* Test(int16_t a, int16_t b) {
2    if (a % 3 == 2) {
3      if (a > 0x1000) {
4        if (b >= 0x0123) {
5          if (a == b) {
6            abort();
7          } else {
8            return malloc(a);
9          }
10       }
11     }
12   }
13 }
```

(a) Sample function in the test program. Parameters a and b are the test inputs.



(b) Sample fuzzed inputs starting with initial seed $a = 0$, $b = 0$. Arrows indicate mutations.

Fig. 1. A motivating example

or not. If i is selected for mutation, CGF randomly mutates i to generate i'. The random mutation can be selected from a set of predefined mutations such as bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with "interesting" integer values (0, MAX_INT), etc. CGF then executes the program with the newly generated input and collects the coverage of the input in the temporary variable *coverage*. If the observed coverage *coverage* contains some new coverage point that is not present in the global cumulative coverage *totalCoverage*, the new input i' is added to the set of saved inputs $\mathcal{S}$. The input i' will then get mutated during a future iteration of the fuzzing loop. The fuzzing loop continues until a time budget has expired.

2.2 A Motivating Example

Consider the sample test program in Figure 1a. The function `Test` takes as input two 16-bit integers, `a` and `b`. A common test objective is to generate inputs that maximize code coverage in this program. We apply Algorithm 1 to perform CGF on this test program. Let us assume that we start with the *seed input*: a=0x0000, b=0x0000. The seed input does not satisfy the condition at Line 2. The CGF algorithm randomly mutates this seed input and executes the test program on the mutated inputs while looking for new code coverage. Figure 1b depicts in grey boxes a series of sample inputs which may be saved by CGF, starting with the initial seed input i_1 in an yellow box. A solid arrow between two inputs, say i and i', indicates that the input i is mutated to generate i'. After some attempts, CGF may mutate the value of `a` in i_1 to a value such as 0x0020, which satisfies the condition at Line 2. Since such an input leads to new code being executed, it gets saved to $\mathcal{S}$. In Fig. 1b, this is input i_2. Small, byte-level mutations enable CGF to subsequently generate inputs that satisfy the branch condition at Line 3 and Line 4 of Fig. 1a. This is because there are many possible solutions that satisfy the comparisons `a > 0x1000` and `b >= 0x0123` respectively; we call these *soft* comparisons. Fig. 1b shows the corresponding inputs in our example: i_3 and i_4. However, it is much more difficult for CGF to generate inputs to satisfy comparisons such as `a == b` at Line 5; we call these *hard* comparisons. Random byte-level mutations on inputs i_1–i_4 are unlikely to produce an input where `a == b`. Therefore, the code at Line 6 may not be exercised in a reasonable amount of time using conventional CGF.

Now, consider another test objective, where we would like to generate inputs that maximize the amount of memory that is dynamically allocated via `malloc`. This objective is useful for generating stress tests or to discover potential out-of-memory related bugs. The CGF algorithm enables us to generate inputs that invoke `malloc` statement at Line 8, such as i_4. However, this input only allocates `0x1220` bytes (i.e., just over 4KB) of memory. Although random mutations on this input are likely to generate inputs that allocate larger amount of memory, CGF will never save these because they have the same coverage as i_4. Thus, it is unlikely that CGF will discover the *maximum* memory-allocating input in a reasonable amount of time.

2.3 Waypoints

Both of the challenges listed above can be addressed if we save some useful intermediate inputs to $\mathcal{S}$ regardless of whether they increase code coverage. Then, random mutations on these intermediate inputs may produce inputs achieving our test objectives. We call these intermediate inputs *waypoints*. For example, to overcome hard comparisons such as `a == b`, we want to save intermediate inputs if they maximize the number of common bits between `a` and `b`. Let us call this strategy `cmp`. The blue boxes in Fig. 1b show inputs that may be saved to $\mathcal{S}$ when using the `cmp` strategy for waypoints. In such a strategy, the inputs i_5 and i_6 are saved to $\mathcal{S}$ even though they do not achieve new code coverage. Now, input i_6 can easily be mutated to input i_7, which satisfies the condition `a == b`. Thus, we easily discover an input that triggers `abort` at Line 6 of Fig. 1a. Similarly, to achieve the objective of maximizing memory allocation, we save waypoints that allocate more memory at a given call to `malloc` than any other input in $\mathcal{S}$. Fig. 1b shows sample waypoints i_8 and i_9 that may be saved with this strategy, called `mem`. The dotted arrow from i_9 to i_{10} indicates that, after several such waypoints, random mutations will eventually lead us to generating input i_{10}. This input causes the test program to allocate the maximum possible memory at Line 8, which is almost 64KB.

Now, consider a change to the condition at Line 4 of Figure 1a. Instead of an inequality, suppose the condition is `b == 0x0123`. To generate inputs that invoke `malloc` at Line 8, we first need to overcome a hard comparison of `b` with `0x0123`. We can combine the two strategies for saving waypoints as follows: save a new input i if *either* it increases the number of common bits between operands of hard comparisons *or* if it increases the amount of memory allocated at some call to `malloc`. In Section 4.7, we demonstrate how a combination of these strategies allows us to automatically generate PNG bombs and LZ4 bombs, i.e. tiny inputs that allocate 2–4 GB of memory, when fuzzing `libpng` and `libarchive` respectively.

We propose a framework, called FuzzFactory, which enables users to implement strategies for choosing waypoints. To do so, the user specifies what custom feedback they need to collect from the execution of a program under test in addition to coverage information. The user also specifies a function for aggregating such feedback across a collection of inputs; the aggregated feedback is used to decide whether an input should be considered a waypoint.

We next describe the framework and its underlying algorithm. The framework has enabled us to rapidly implement three existing strategies in the literature and four new strategies, including a composite strategy.

3 FUZZFACTORY: A DOMAIN-SPECIFIC FUZZING FRAMEWORK

Our goal is to construct a framework which allows users to build a domain-specific fuzzing application d by simply defining a custom predicate: $is\_waypoint(i, \mathcal{S}, d)$. The predicate tells the fuzzer whether a new input i is a *waypoint*; that is, whether i should to be saved to the set of saved inputs $\mathcal{S}$ so that later on it can be mutated to generate new inputs.

In the conventional CGF algorithm, the decision of whether to save an input is defined in terms of the dynamic behavior of the program on the input i. Specifically, if the coverage of the program

on the input i includes a coverage point that is not present in the coverage cumulatively attained by the program on the inputs in $\mathcal{S}$, then CGF deems i as interesting and saves it to $\mathcal{S}$. The decision is based on a specific kind of feedback (i.e. coverage) from the execution of the program on i. The feedback is directly related to the goal of CGF, which is to increase the coverage of the program.

Although improving code coverage is important for discovering new program behavior, we believe that a fuzzer could be made more effective and diverse if it was guided by other testing goals, such as: discovering performance bottlenecks or memory usage problems, covering recently modified code, exercising valid input behavior, etc.

FuzzFactory enables users to prototype fuzzers that target user-defined custom goals. To support custom or domain-specific goals, the user needs to specify: (1) the specific kind of feedback to collect from the execution of the program on any input, and (2) how this feedback should be used to determine if the input should be considered interesting and saved.

We next describe the mechanism with which the FuzzFactory user specifies the kind of domain-specific feedback they want from an execution. We then explain how the *is_waypoint* predicate uses such custom feedback to determine if an input needs to be saved. We also describe how to compose such domain-specific feedback. Finally, we show how to extend the CGF algorithm in Algorithm 1 to take domain-specific feedback into account.

3.1 Domain-Specific Feedback

In FuzzFactory, we provide a mechanism for users to specify a *domain* and to collect custom *domain-specific feedback* (DSF) from the execution of the program under test. A domain-specific feedback (DSF) is a map of the form $dsf_i : K \rightarrow V$, where i is a program input, K is a set of keys (e.g. program locations) and V is a set of values (usually a measurement of something we want to optimize). The map is populated by executing the program under test on input i. As an example, if we are interested in generating inputs on which the program execution increases memory allocation, then dsf_i is a map from $\mathbb{L}$ to $\mathbb{N}$, where $\mathbb{L}$ is the set of program locations where a memory allocation function (e.g. `malloc`) is called and $\mathbb{N}$ is the set of natural numbers. $dsf_i(k)$ represents the total amount of memory in bytes that is allocated at program location k during the execution of the program on the test input i.

In general, the user specifies a domain as a tuple of the form $d = (K, V, A, a_0, \triangleright)$ where K is a set of keys, V is a set of values, A is a set of aggregation values, a_0 is an initial aggregation value, and $\triangleright : A \times V \rightarrow A$ is a reducer function. The user specifies how to update the map dsf_i during an execution of the test program on input i, by inserting appropriate instrumentation in the test program. We explain the meaning of A, a_0, and $\triangleright$ in a user-defined domain in the next subsection.

3.2 Waypoints

We use the dsf_i map from the execution of the test program on input i in order to determine if i needs to be saved. To do so, FuzzFactory aggregates the domain-specific feedback collected from the executions of multiple test inputs into a value that belongs to the user-defined set A. To compute this aggregate value, the user provides an initial aggregate value $a_0 \in A$ and a *reducer* function $\triangleright : A \times V \rightarrow A$ as part of the domain. A reducer function must satisfy the following properties for any $a \in A$ and any $v, v' \in V$:

$$a \triangleright v \triangleright v = a \triangleright v \quad (1)$$

$$a \triangleright v \triangleright v' = a \triangleright v' \triangleright v \quad (2)$$

These rules imply idempotence and application-order insensitivity, respectively, in the second operand. For the memory-allocation domain (say d^{mem}): both V and A are the set of natural numbers $\mathbb{N}$. The initial aggregate value $a_0 = 0$, and $\triangleright$ is the *max* operation on natural numbers. We can

therefore define $d^{mem} = (\mathbb{L}, \mathbb{N}, \mathbb{N}, 0, max)$. Property 1 is satisfied because $\max(\max(a, v), v) = \max(a, v)$ for any $a, v \in \mathbb{N}$. Property 2 is satisfied because $\max(\max(a, v), v') = \max(\max(a, v'), v)$ for any $a, v, v' \in \mathbb{N}$. The properties help ensure that the every saved waypoint contributes towards domain-specific progress; this point will be visited when encountering Theorem 1 below. Note that these properties are not statically verified by FuzzFactory; it is the responsibility of the user to ensure that their chosen reducer function satisfies Properties 1 and 2.

In general, let dsf_i be the DSF map populated during the execution of program p with i. For a given set of inputs $\mathcal{S} = \{i_1, i_2, \ldots, i_n\}$, we define the aggregated domain-specific feedback value $\mathcal{A}(\mathcal{S}, k, d)$ for the domain d and for key $k \in K$ as follows:

$$\mathcal{A}(\mathcal{S}, k, d) \stackrel{\text{def}}{=} a_0 \rhd dsf_{i_1}(k) \rhd dsf_{i_2}(k) \rhd \ldots \rhd dsf_{i_n}(k), \text{ where } d = (K, V, A, a_0, \rhd) \tag{3}$$

Due to the Properties 1 and 2, the value of $\mathcal{A}(\mathcal{S}, k, d)$ is uniquely defined; the choice of ordering $i_1, \ldots, i_n$ does not matter.

For the memory-allocation domain, the aggregated feedback value $\mathcal{A}(\mathcal{S}, k, d^{mem})$ represents the *maximum* amount of memory allocated at program location $k \in \mathbb{L}$ across all inputs in $\mathcal{S}$. For this domain, we would like to save an input i to set $\mathcal{S}$ if the execution on i causes more memory allocation at some program location k than that of any of the allocations observed at k during the execution of the inputs in $\mathcal{S}$.

In FuzzFactory, we define the predicate $is\_waypoint(i, \mathcal{S}, d)$ as follows:

$$is\_waypoint(i, \mathcal{S}, d) \stackrel{\text{def}}{=} \exists k \in K : \mathcal{A}(\mathcal{S}, k, d) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d), \text{ where } d = (K, V, A, a_0, \rhd) \tag{4}$$

The definition implies that we will save input i if the execution on the input results in a change in the aggregated domain-specific feedback value for some key.

Note that, in order to decide if an input i should be considered a waypoint, we only check if the total aggregation *changes*; i.e., whether $\mathcal{A}(\mathcal{S}, k, d) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d)$. However an important consequence of Properties 1 and 2 is that this change is always in a direction that implies some sort of *domain-specific progress*, denoted by a partial order $\preceq$ on A. In other words, the function $\mathcal{A}$ is monotonic in its first argument with respect to partial order $\preceq$. For example, in the memory allocation domain d^{mem}: if $\mathcal{A}(\mathcal{S}, k, d^{mem}) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d^{mem})$ for some program location $k \in \mathbb{L}$, this means that the memory allocated at k during the execution of i is *more* than the memory allocated at k by any other input in $\mathcal{S}$. The partial order in this example is simply the total ordering on natural numbers: $\leq$. More generally, we can state the following theorem:

Theorem 1 (Monotonicity of Aggregation). *A domain $d = (K, V, A, a_0, \rhd)$ whose reducer function $\rhd$ satisfies properties 1 and 2 imposes a partial order $\preceq$ on A such that the function $\mathcal{A}$ is monotonic in its first argument with respect to $\preceq$. That is, the following always holds for any such domain d, any key $k \in K$, and for some binary relation $\preceq$ on A:*

$$\mathcal{S}_1 \subseteq \mathcal{S}_2 \Rightarrow \mathcal{A}(\mathcal{S}_1, k, d) \preceq \mathcal{A}(\mathcal{S}_2, k, d)$$

We prove this theorem in Appendix A.

Corollary 2. *An input i is considered a waypoint iff the aggregated domain-specific feedback strictly makes progress for some key k, without sacrificing progress for any other key. In particular:*

$$\begin{aligned} is\_waypoint(i, \mathcal{S}, d) \Leftrightarrow & (\forall k \in K : \mathcal{A}(\mathcal{S}, k, d) \preceq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d)) \\ & \wedge (\exists k \in K : \mathcal{A}(\mathcal{S}, k, d) \prec \mathcal{A}(\mathcal{S} \cup \{i\}, k, d)) \end{aligned}$$

where $a \prec b \Leftrightarrow a \preceq b \wedge a \neq b$

Proof. Follows from the definition of *is_waypoint* in Eq. 4 and Theorem 1. □

Algorithm 2 The domain-specific fuzzing algorithm. The grey boxes indicate additions to the standard coverage-guided fuzzing algorithm in Algorithm 1.

Input: an instrumented test program p, a set of initial seed inputs S_0, a set of domain-specific feedback D
Output: a corpus of automatically generated inputs $\mathcal{S}$

1: $\mathcal{S} \leftarrow S_0$
2: $totalCoverage \leftarrow \emptyset$
3: **repeat** ▷ Main fuzzing loop
4: **for** i in $\mathcal{S}$ **do**
5: **if** sample FUZZPROB(i) **then**
6: $i' \leftarrow$ MUTATE(i) ▷ Generate new test input i'
7: $coverage$, $dsf_{i'}^1, \ldots, dsf_{i'}^{|D|} \leftarrow$ EXECUTE(p, i') ▷ Run test with new input i'
8: **if** $coverage \cap totalCoverage \neq \emptyset$ **then**
9: $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ ▷ Save i' if new code coverage achieved
10: $totalCoverage \leftarrow totalCoverage \cup coverage$
11: **if** $is\_waypoint(i', \mathcal{S}, D)$ **then**
12: $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ ▷ Save i' to fuzzing corpus
13: **until** given time budget expires
14: **return** $\mathcal{S}$

3.3 Composing Domains

FUZZFACTORY allows the user to naturally compose multiple domains for a program under test. This enables fuzzing to target multiple goals simultaneously.

Assume that the user has specified a set of domains D, where $d = (K, V, A, a_0, \triangleright)$ for each $d \in D$. Then we extend the definition of the predicate *is_waypoint* to D as follows:

$$is\_waypoint(i, \mathcal{S}, D) \stackrel{\text{def}}{=} \bigvee_{d \in D} is\_waypoint(i, \mathcal{S}, d) \tag{5}$$

which says that $is\_waypoint(i, \mathcal{S}, D)$ is true for a set of domains D if and only if $is\_waypoint(i, \mathcal{S}, d)$ is true for some domain $d \in D$. We save the input i in $\mathcal{S}$ if $is\_waypoint(i, \mathcal{S}, D)$ is true. Note that Corollary 2 naturally extends to a composition of multiple domains: $is\_waypoint(i, \mathcal{S}, D)$ implies strict progress in at least one key in at least one domain $d \in D$.

3.4 Algorithm for Domain-Specific Fuzzing

Algorithm 2 describes the domain-specific fuzzing algorithm implemented in FUZZFACTORY. The algorithm extends the conventional coverage-guided fuzzing algorithm described in Algorithm 1. The extensions are marked with grey background. The extension is quite straightforward: during the execution of the program p on an input i', the algorithm not only collects *coverage*, but also collects domain-specific feedback maps $dsf_{i'}^1, \ldots, dsf_{i'}^{|D|}$ for each domain in D. It then uses those maps in the call to $is\_waypoint(i', \mathcal{S}, D)$ to determine if the new input i' should be added to the set of saved inputs $\mathcal{S}$.

Table 1. Lines of code (LoC) required to implement each domain-specific fuzzing application. For implementing in FuzzFactory, the table counts C++ code that implements compile-time instrumentation (LoC_{inst}), run-time support code (LoC_{rt}), and the reducer function ($\text{LoC}_{\triangleright}$), as well as the sum of these three numbers ($\text{LoC}_{\text{total}}$). For domains that are re-implementations or prior work, we list the lines of code added or modified (LOC_{ext}) by the corresponding standalone implementations by comparing with the underlying coverage-guided fuzzer that was extended. All measurements performed using `cloc` 1.74; blank and comment-only lines ignored.

	FuzzFactory				Standalone Tool		
Domain	LoC_{inst}	LoC_{rt}	$\text{LoC}_{\triangleright}$	$\text{LoC}_{\text{total}}$	**Prior Work**	**Baseline**	LOC_{ext}
`slow`	11	2	5	**18**	[Petsios et al. 2017b]	LibFuzzer 4.0	386
`perf`	12	2	5	**19**	[Lemieux et al. 2018]	AFL 2.52b	312
`mem`	22	2	5	**29**	-	-	-
`valid`	11	8	5	**24**	[Padhye et al. 2019c]	N/A	†621
`cmp`	97	245	13	**355**	-	-	-
`diff`	121	12	13	**146**	-	-	-

† The original validity fuzzing algorithm was implemented from scratch in Java instead of extending an underlying coverage-guided fuzzer. The LoC_{ext} listed here corresponds to the entire Java class that implements this algorithm (`ZestGuidance.java`), and is therefore an over-approximation.

4 DOMAIN-SPECIFIC FUZZING APPLICATIONS

We demonstrate the applicability of FuzzFactory by instantiating six independent domain-specific fuzzing applications. Some of these fuzzing algorithms were already proposed and implemented in prior work. Our motivation behind implementing these algorithms was to evaluate whether we could prototype these algorithms in our framework, without changing the underlying fuzzing algorithm or search heuristics. Sections 4.1 through 4.6 describe six domains, in increasing order of complexity:

(1) `slow`: An application for maximizing execution path lengths, based on SlowFuzz [Petsios et al. 2017b]. This is the most trivial domain to implement in FuzzFactory.
(2) `perf`: An application for discovering hot spots by maximizing basic block execution counts, based on PerfFuzz [Lemieux et al. 2018]. In FuzzFactory, this naturally generalizes `slow`.
(3) `mem`: A novel application for generating inputs that maximize dynamic memory allocations.
(4) `valid`: An application of the validity fuzzing algorithm [Padhye et al. 2019b,c], which attempts to bias input generation towards inputs that satisfy program-specific validity checks.
(5) `cmp`: A domain for smoothing hard comparisons. Although a lot of prior work address this application, our particular solution strategy is novel.
(6) `diff`: A novel application for incremental fuzzing after code changes in a test program.

For each application, (1) we define the domain d in terms of the tuple $(K, V, A, a_0, \triangleright)$ (2) we describe, with the help of some utilities defined Table 2, how we instrument test programs to populate the map dsf_i during test execution on input i[1], and (3) we report the results of applying the domain-specific fuzzing implementation to a set of real-world programs.

Composition. A key advantage of FuzzFactory is that it enables us to naturally compose multiple domain-specific fuzzing applications with no extra effort. In Section 4.7, we describe a composition of `cmp` and `mem` that smooths hard comparisons in order to exacerbate memory allocations. Remarkably, we find that such a composition can perform better than just the sum of its parts.

[1]We will drop the subscript i from dsf_i when it is clear from context.

Table 2. Definition of instrumentation functions used for injecting code which updates domain-specific feedback maps. They are used in Table 3 through 8. Hooks are activated when corresponding syntactic objects are encountered during a compile-time pass over the program under test. The handler logic for these hooks can inject code in the program under test. Actions are the functions that are used to actually inject code during instrumentation. Utility functions are available to the handler logic at compile-time.

Instrumentation Hooks	**Description**
new_basic_block()	Activated at the beginning of a basic block in the control-flow graph of the program under test.
entry_point()	Activated at the entry point for test execution (e.g. start of the `main` function).
func_call(*name*, *args*)	Activated at an expression that invokes function named *name* with arguments *args*.
bin_expr(*type*, *left*, *op*, *right*)	Activated at an expression with binary operator of the form '*left op right*' (e.g. `x == 42`), where the operands have type *type* (e.g. `long`).
switch(*type*, *val*, *cases*)	Activated when encountering a `switch` statement on value *val* of type *type*, where *cases* is a list of the `case` clauses.
Instrumentation Actions	**Description**
insert_after(*inst*)	Inserts an instruction *inst* immediately after the instruction whose instrumentation hook is currently activated.
insert_before(*inst*)	Inserts an instruction *inst* immediately before the instruction whose instrumentation hook is currently activated.
Utility functions	**Description**
current_program_loc()	Returns the program location (i.e., a value in set $\mathbb{L}$) corresponding to the current instrumentation location.
target_program_loc(*case*)	Returns the program location (i.e., a value in set $\mathbb{L}$) that is the target of a `case` within a `switch` statement.
comm_bits(*a*, *b*, *n*)	Counts the number of common bits between two *n*-byte operands *a* and *b*. For example, *comm_bits*(1025, 1026, 4) = 30, since only 2 bits in these 32-bit operands differ.

Implementation. Traditionally, implementing each such domain would require non-trivial effort in modifying a fuzzing tool such as AFL to achieve a domain-specific objective. With FuzzFactory, four of the above six domains can be implemented in less than 30 lines of C++ code each. Table 1 lists the lines of code required to implement each of the six domains that we present in this paper using FuzzFactory. Section 6 provides some more details about our implementation. For domains that are re-implementations of prior work, the table also lists the lines of code that were required to implement the corresponding specialized standalone fuzzing tools.

Program Instrumentation. Sections 4.1 through 4.6 describe how test programs are instrumented to implement each of the six domains that we present in this paper. The instrumentation enables the collection of domain-specific feedback in the map dsf_i when executing the test program on an input i. Such instrumentation is performed at compile-time. Although our implementations performs instrumentation at the LLVM IR level, for ease of presentation we describe the instrumentation logic for each of the six domains at a higher level of abstraction. Table 2 lists some hooks, actions, and utility functions that we use in our abstract descriptions of domain-specific instrumentation. We next describe how to interpret the information in Table 2.

A *hook* is activated at compile-time by an instrumentation framework (e.g. LLVM) whenever a corresponding element in a program is encountered while making a pass over the test program. For example, the *func_call*(*name*, *args*) hook is invoked at compile-time for every function call expression in the program. Here, *name* is a string and *args* is a list of references to the syntactic expressions that form the arguments to the function call. An instrumentation pass, such as the one we write for each fuzzing domain, specifies some logic to handle such hooks. The handler logic can optionally insert new code before or after the program element whose hook is currently activated. For example, a handler for *func_call* can statically look at *name* (say f) to decide whether to insert code around a call to f. Code is inserted by invoking *actions* such as *insert_after* and *insert_before*. The inserted code can use compile-time constants or refer to static program elements such as: one or more arguments to f, global variables, or user-defined functions. For ease of presentation, we will show the inserted code as source-level pseudocode instead of an instruction in some IR. Commonly, we will insert code that updates the dsf_i map—in practice, we insert an instruction that invokes one of the APIs listed in Section 6.1. The handler logic is unrestricted; in our implementation, it is arbitrary C++ code that uses the LLVM API. The handler logic can make use of *utility functions* provided by FuzzFactory at compile-time. Table 2 only lists the hooks and utility functions required to describe the six domains presented in the paper (Tables 3–8). To implement new domains, other language constructs such as branches, loads, stores, etc. can also be instrumented.

Experimental Evaluation. For our experiments, we use six benchmark programs from the Google fuzzing test suite [Google 2019b]. This suite contains specific historical versions of programs that have been thoroughly fuzzed using the OSS-fuzz infrastructure [Google 2019a]. The six benchmarks we use include: (1) libpng-1.2.56, (2) libarchive-2017-01-04, (3) libjpeg-turbo-07-2017, (4) libxml2-v2.9.2, (5) vorbis-2017-12-11, and (6) boringssl-2016-02-12.[2] The benchmarks are written in C or C++. Benchmarks (1)–(4) were chosen because they are commonly used in the fuzzing literature [Chen and Chen 2018; Chen et al. 2019; Lemieux et al. 2018; Lemieux and Sen 2018; Peng et al. 2018; Pham et al. 2018]. Benchmarks `vorbis` and `boringssl` were chosen because they expect markedly different input formats. We only used six benchmarks from Google's test suite because of resource constraints: for our evaluation, we spent two CPU-years fuzzing these six benchmarks alone.

All experiments were run on Amazon AWS 'c5.18xlarge' instances. Each experiment was repeated 12 times to account for variability in the randomized algorithms. Unless otherwise stated, our fuzzing experiments used the initial seed inputs provided in the benchmark suite, limited input sizes to at most 10KB during fuzzing, and were run for 24 hours at a time.

For each application, we evaluate the following research question: "*Does FuzzFactory help achieve domain-specific fuzzing goals, without modifying the underlying search algorithm?*". FuzzFactory is implemented as an extension to AFL, and inherits its mutation and search heuristics. For each application domain, we thus compare the results of domain-specific fuzzing with the baseline: conventional coverage-guided fuzzing using AFL. Naturally, the metrics on which we perform this comparison vary depending on the domain. We note that it is not meaningful to compare the results of FuzzFactory with the results of specialized domain-specific fuzzing tools implemented in prior work if such specialized tools also use different mutations and search heuristics. As such, we only perform a direct comparison with prior work if it extends AFL, similarly to FuzzFactory.

4.1 `slow`: Maximizing Execution Path Length

Fuzz testing can be used to generate inputs that exacerbate the algorithmic complexity of a program under test. SlowFuzz [Petsios et al. 2017b] introduced this idea using a resource-guided evolutionary

[2]For `boringssl`, we use the target `fuzz/server.cc`, which fuzzes the server side of the TLS handshake protocol, instead of the default `fuzz/privkey.cc`, which fuzzes the parsing of private key files.

Table 3. `slow`: Application for maximizing execution path length

Domain d: $K = \{0\}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \rhd v = \max(a, v)$	
Hook	**Instrumentation**
entry_point()	**insert_after**(‘$dsf(0) \leftarrow 0'$)
new_basic_block()	**insert_after**(‘$dsf(0) \leftarrow dsf(0) + 1'$)

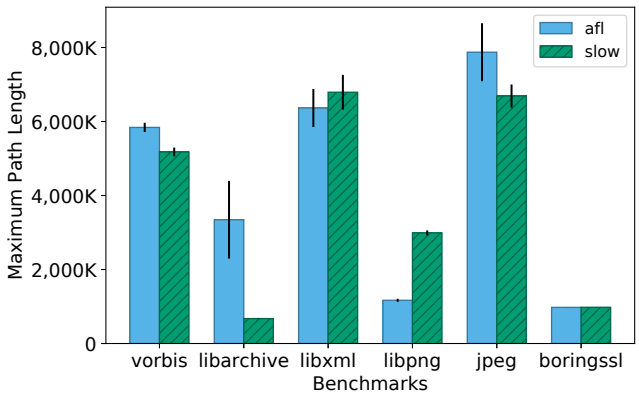


Fig. 2. Maximum execution path lengths achieved by baseline (afl) and domain-specific fuzzing application (slow). Higher is better.

search. The search uses a fitness function that counts the number of basic blocks executed during the execution of a single test input. We call this metric the *execution path length*.

Our first domain-specific fuzzing application is a port of SlowFuzz to our framework. The goal of this application is to generate inputs that maximize the execution path length in the program under test. We want to define the $is\_waypoint(i, \mathcal{S}, d)$ predicate as follows: an input i should be saved if its execution leads to a higher path length than any other input in $\mathcal{S}$.

The first row of Table 3 defines this domain (say d) as follows. The domain-specific feedback map dsf maps the single key 0 ($K = \{0\}$) to a natural number ($V = \mathbb{N}$). In the map, $dsf(0)$ represents the execution path length for a test input i. These values are aggregated into a number ($A = \mathbb{N}$) which represents the maximum execution path length observed across a set of inputs ($a_0 = 0,\ a \rhd v = \max(a, v)$).

Table 3 also describes how we instrument test programs to correctly update entries in the map dsf at run-time. We make use of the instrumentation hooks *entry_point* and *new_basic_block*, and the action *insert_after*, all defined in Table 2. Using these functions, we can interpret the description in Table 3 as follows: At the entry point of the program under test, insert a statement that sets $dsf(0)$ to 0. Then, at each basic block in the program, insert a statement that increments the value stored at $dsf(0)$. Thus, during a test execution, the value of $dsf(0)$ is incremented by one each time a basic block is visited. At the end of the test input execution, the value of $dsf(0)$ will contain the execution path length. Since the reducer function for this domain is defined to be *max* with an initial value of 0 (see first row of Table 3), the aggregated value of the domain-specific feedback $\mathcal{A}(\mathcal{S}, 0, d)$ will be the maximum execution path length observed across all the inputs in $\mathcal{S}$.

Experimental evaluation. Figure 2 shows the results of our experiments with this application on our benchmark programs. We evaluate the maximum execution path lengths (across the generated test corpus) for the baseline (`afl`) and our domain-specific fuzzing application (`slow`), after 24 hours of fuzzing. The figure plots the mean value and standard error of this metric across 12 repetitions. For `libpng`, the domain-specific feedback enables the generation of inputs whose path lengths are more than 2.5× that of the baseline. For `boringssl` and `libxml`, the increase is not as significant. Interestingly, the maximum execution path length for `slow` is actually *lower* than that found by `afl` on the remaining three benchmarks. One possible explanation for this result is that `slow` attempts to aggressively maximize execution path lengths starting from the very first input. On the other hand, `afl` spends its time maximizing code coverage and discovers longer execution paths in components of the test program that are not exercised by the seed inputs. The difference is most noticible in `libarchive`. Among all of the benchmarks we considered, `libarchive` is the only benchmark for which the initial seed input provided in Google's test suite is invalid. That is, the initial seed input for `libarchive` leads the test program to exit early in an error state. Since AFL spends its 24 hours increasing only code coverage, it is able to eventually generate inputs that are valid archives (e.g. ZIP files), whose processing leads to longer execution paths. On benchmarks such as `libpng`, the provided seed input is valid and already covers interesting code paths within the test programs; therefore, `slow` is able to maximize path lengths effectively. This SlowFuzz-inspired approach appears to work best when initial seed inputs already provide good code coverage.

Note that we did not directly compare our implementation with the SlowFuzz tool implemented by Petsios et al. [2017b]. SlowFuzz is an extension of libFuzzer, wheras FuzzFactory is built on top of AFL. The mutations and search heuristics used by libFuzzer differ from AFL; therefore, a comparison between the SlowFuzz tool and our implementation of `slow` would not help us determine the value of domain-specific feedback independent of the search heuristics.

4.2 `perf`: Discovering Hot Spots

PerfFuzz [Lemieux et al. 2018] is another tool that uses fuzz testing for generating inputs with pathological performance. Unlike SlowFuzz, which maximizes a single criteria–execution path length—PerfFuzz independently maximizes execution counts for each basic block in the program under test. To do this, PerfFuzz extends the coverage-guided fuzzing algorithm to save newly generated inputs if they increase the maximum observed execution count for *any* basic block. In this domain, the goal is to find inputs that execute the same basic block many times.

Table 4 describes how we implement PerfFuzz in our framework. The first line defines the domain. The keys in the DSF map (i.e. K) range over the set of program locations $\mathbb{L}$. The values of the DSF map as well as the aggregated values represent execution counts (i.e. $V = \mathbb{N}$ and $A = \mathbb{N}$). The reducer function (i.e. $\triangleright$) is *max* with initial value $a_0 = 0$, just as in SlowFuzz.

Table 4 also describes how we instrument the program under test. At the start of every test execution (*entry_point*), we initialize the entire DSF map with values 0. Each time a new basic block k is visited, we increment the value stored at $dsf(k)$. This is done in the instrumentation hook function *new_basic_block*, using the *current_program_loc*() function to statically get the program location of the basic block being instrumented (ref. Table 2). At the end of test execution, $dsf(k)$ will contain the number of times that basic block k was executed. Since the reducer function is *max*, a newly generated input will be considered a waypoint if it increases the execution count for any basic block k in the test program.

Experimental evaluation. Figure 3 contains the results of our experiments with this application on our benchmark programs. Since the PerfFuzz tool implemented by Lemieux et al. [2018] is also an extension of AFL, it uses the same mutation and search heuristics as FuzzFactory; therefore, for

Table 4. `perf`: Application for discovering hot spots

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \rhd v = \max(a, v)$	
Hook	**Instrumentation**
entry_point()	**insert_after**(‘$\forall k \in K : dsf(k) \leftarrow 0$’)
new_basic_block()	$k \leftarrow$ *current_program_loc*()
	insert_after(‘$dsf(k) \leftarrow dsf(k) + 1$’)

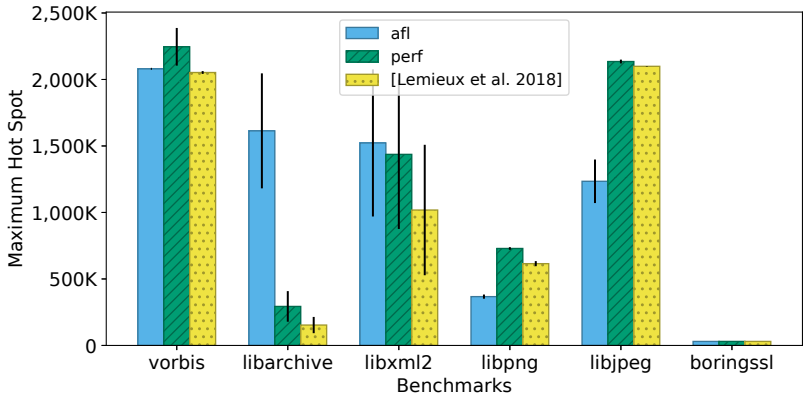


Fig. 3. Maximum basic block execution counts achieved by baseline (afl), domain-specific fuzzing application (perf), and PerfFuzz [Lemieux et al. 2018]. Higher is better.

this application, we can perform a direct comparison with PerfFuzz. We evaluate the FuzzFactory domain-specific fuzzing application (`perf`), the baseline (`afl`), and the PerfFuzz tool, on the metric *max hot spot*. The PerfFuzz paper defines *max hot spot* to be the maximum execution count for any basic block across all inputs in the generated test corpus. The figure plots the mean value and standard error of this metric across 12 repetitions.

Figure 3 shows that `perf` is able to generate inputs that significantly maximize hot spots for three of the six benchmarks: `vorbis`, `libpng`, and `libpjeg`. For `libpng` and `libjpeg-turbo`, the hot spots discovered by `perf` execute 2× and 1.7× more than those discovered by the baseline `afl`. For `libarchive`, the `perf` application performs much worse. Similar to the experiments reported in the previous section, the main problem here is that the initial seed inputs provided with `libarchive` lead to an early exit. Since baseline AFL spends more time increasing code coverage rather than basic block execution counts, it eventually generates valid archive files (e.g. ZIP). Given that `libarchive` is a program that performs decompression, the generation of a valid archive is sufficient to discover a huge hot spot in the code component that performs decompression. On the other hand, `perf` only discovers hot spots in `libarchive`'s parsing of file meta-data. Our evaluation indicates that the PerfFuzz algorithm also depends on initial seed inputs that cover interesting code paths. On all benchmarks, `perf`'s results are similar to or slightly better than the specialized PerfFuzz tool.

4.3 `mem`: Exacerbating Memory Allocations

We now describe a novel application of FuzzFactory: generating inputs that exacerbate memory allocation. There are several use cases for such a domain such as discovering the maximum amount

Table 5. `mem`: Application for exacerbating memory allocation

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \triangleright v = \max(a, v)$	
Hook	**Instrumentation**
entry_point()	**insert_after**(‘$\forall k \in K : dsf(k) \leftarrow 0$’)
func_call(*name*, *args*)	if *name* ∈ {‘`malloc`’, ‘`calloc`’} :
	$k \leftarrow$ *current_program_loc*()
	bytes ← *args*[0]
	insert_after(‘$dsf(k) \leftarrow dsf(k) + bytes$’)

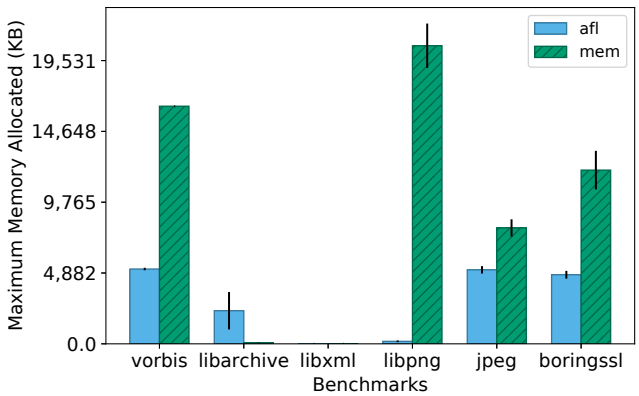


Fig. 4. Maximum amount of dynamic memory allocated (in KB) due to inputs generated by baseline (afl) and domain-specific fuzzing application (mem). Higher is better.

of memory the program under test may dynamically allocate for a given size input, discovering inputs that could lead to bugs related to out-of-memory conditions, or generating a corpus of memory-stress tests for benchmarking purposes.

Table 5 describes our instrumentation for the memory-allocation domain. The definition of the domain on the first line of this table, as well as the initialization of *dsf* at the entry point, is exactly the same as that of the PerfFuzz domain (Table 4). However, instead of incrementing the values in the DSF map at every basic block, we instrument expressions in the test program that invoke the function `malloc` or `calloc`. Whenever the test program allocates new memory using `malloc` or `calloc` at program location k, we increment the value of $dsf(k)$ by the number of bytes allocated. At the end of test execution, the value of $dsf(k)$ contains the total number of bytes allocated at program location k for all such locations k.

Experimental evaluation. Figure 4 shows the results of our experiments with this application on our benchmark programs. We evaluate the domain-specific fuzzing application (`mem`) as well as the baseline (`afl`) on the maximum amount of dynamic memory allocated by generated inputs after the 24-hour fuzzing runs. The plots show means and standard errors of this metric across 12 repetitions.

The benchmark `libxml` did not seem to perform any input-dependent dynamic memory allocations. On the benchmarks `vorbis`, `libpng`, `libjpeg-turbo` and `boringssl`, our domain-specific

```
1 void Test(uint8_t* data, int size) {
2   /* set up png_ptr  */
3   if (png_get_IHDR(png_ptr, ...) != 0)
4     return; // invalid header
5   /* process PNG data */
6 }
```

(a) Original test driver

```
1 void Test(uint8_t* data, int size) {
2   /* set up png_ptr  */
3   assume(png_get_IHDR(png_ptr, ...)
4     == 0); // valid header
5   /* process PNG data */
6 }
```

(b) Modified test driver

Fig. 5. Sample change to `libpng` test driver to enable validity fuzzing.

Table 6. `valid`: Application for validity fuzzing

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = 2^{\mathbb{N}}, a_0 = \emptyset,\ a \triangleright v = a \cup \log_2(v)$	
Hook	**Instrumentation**
entry_point()	**insert_after**(‘$\forall k \in K : dsf(k) \leftarrow 0$’)
new_basic_block()	$k \leftarrow$ *current_program_loc*() **insert_after**(‘$dsf(k) \leftarrow dsf(k) + 1$’)

func_call(*name*, *args*)	if *name* = ‘`assume`’ :
	  cond $\leftarrow$ *args*[0]
	  insert_before(‘if *cond* = false then $\forall k \in K : dsf(k) \leftarrow 0$’)

fuzzing application generated inputs that allocate 1.5×–120× more memory. For `libpng` our application generated input PNG images whose metadata specified the maximum allowable image dimensions–as per the validation rules hard-coded in the test driver—of 2 million pixels. Even though such PNG files themselves were only about 1KB in size, their processing required over 24MB of dynamically allocated memory. In Section 4.7, we discuss a composite domain-specific fuzzing application that generates PNG images of dimensions smaller than one thousand pixels, but whose processing required over 2GB of dynamic memory allocation from `libpng`.

Just like with `slow` and `perf` (ref. Sections 4.1 and 4.2 respectively), the `mem` application was not effective on `libarchive`. Recall that this is the only benchmark in our suite where the initial seed input leads to an early exit due to a validation error.

4.4 `valid`: Validity Fuzzing

A major problem associated with CGF is that most randomly generated inputs are invalid; that is, they cause the test program to exit early with an error state. For example, traditional CGF on `libpng` is unlikely to generate many valid PNG images, even if fuzzing is seeded with valid inputs to begin with. Most of the code coverage achieved by the newly generated inputs lies in code paths that deal with input validation and error reporting. Therefore, CGF algorithms struggle to effectively test and find bugs in the main functionalities of such programs.

In many cases, it is desirable to generate *valid* inputs that maximize code coverage. For example, one may want to test programs such as image viewers and media players that download and process files that were uploaded on a social media website. Most likely, such websites do not allow users to upload invalid files. Bugs in the image viewers or media players would then manifest only during the processing of valid files.

Validity fuzzing [Padhye et al. 2019c] has been recently proposed to address the problem of generating valid inputs. In validity fuzzing, test programs are augmented to return feedback about

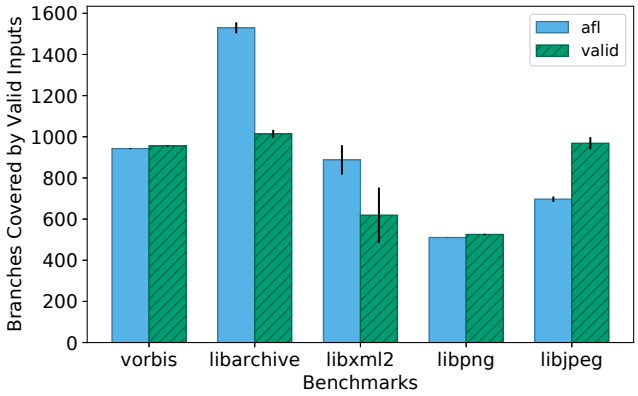


Fig. 6. Branch coverage among valid inputs, as achieved by inputs generated by baseline (afl) and domain-specific fuzzing application (valid). Higher is better.

whether or not an input is *valid*, according to some program-specific notion of validity, e.g. whether an input to `libpng` is a valid PNG file. During the fuzzing loop, newly generated inputs are saved either (1) if they increase overall code coverage, or (2) if the newly generated input is valid *and* it covers code that has not been covered by any previously generated *valid* input. The first criterion allows saving intermediate inputs regardless of validity as long as they produce new cumulative code coverage. The hope is that mutating these inputs will lead to more interesting valid inputs being generated later on. The second criterion attempts to maximize code coverage among the valid inputs. Other researchers have also used notions of program-specific validity to guide the fuzzing search towards generating more valid inputs [Laeufer et al. 2018; Pham et al. 2018].

We now demonstrate how we implemented the validity fuzzing algorithm in our framework. First, we modified the test drivers that ship with the benchmark suite to add program-specific `assume(expr)` statements. The semantics of `assume` is similar to that of the more familiar `assert`: if the argument `expr` evaluates to `true` at run-time, then the statement is a no-op; otherwise, the test execution is stopped. Figure 5 demonstrates one of the three single-line changes we made to the `libpng` test driver. Instead of exiting early due to an invalid PNG header, we simply wrap the validity check with an `assume` statement. We were able to make such small changes in the test drivers of all benchmarks except `boringssl`. Across the five benchmarks whose drivers we modified, we added 1–3 `assume` statements that wrapped existing validity checks in the test drivers, changing 1–11 lines of code. Second, we instrumented the test program to populate the DSF map with information about code coverage during test execution, similar to traditional coverage-guided fuzzing. At runtime, if any of the arguments to `assume` evaluates to `false`, the entire DSF map is reset to the initial state before exiting. Therefore, the DSF map mirrors the traditional code coverage information if and only if the test input is valid. Invalid inputs produce no domain-specific feedback. This scheme leads to the following behavior for Algorithm 2: a newly generated input is saved if either it leads to new cumulative code coverage, or if the input is valid and achieves more code coverage (i.e., changes the aggregate domain-specific feedback) than any other valid input seen so far (i.e., among inputs that produce domain-specific feedback).

Table 6 describes the validity fuzzing application more formally. The first line of this table defines the domain. The DSF map for this domain maps program locations (i.e. $K = \mathbb{L}$) to execution counts (i.e. $V = \mathbb{N}$), similar to the `perf` application (ref. Section 4.2). However, when aggregating

domain-specific feedback, the validity fuzzing application collects a *set of orders of magnitude* of the execution counts for each basic block (i.e. $A = 2^{\mathbb{N}}$). This mirrors the heuristics used by AFL in collecting code coverage [Zalewski 2017]. The aggregation is defined by the reduce operator: $a \triangleright v = a \cup \log_2(v)$, where $\log_2(v)$ extracts the position of the highest set bit in the value v extracted from the DSF map. The initial value is the empty set: $a_0 = \emptyset$. Such information allows for differentiation between inputs that execute the same code fragment, say, 2 times versus 4 times (since these counts have different orders of magnitude), but not, say, 10 times versus 11 times (since these counts have the same order of magnitude). The actions described for hooks *entry_point* and *new_basic_block* in Table 6 are exactly the same as those for the `perf` application (Table 4). The hook for *func_call* handles calls to `assume()`. The instrumentation inserts code that performs the required logic: if the argument to `assume` evaluates to `false`, then clear all entries in the DSF map before calling `assume`, which stops the test.

Experimental evaluation. Figure 6 contains the results of our experiments with this application on our benchmark programs. We evaluate the domain-specific fuzzing application (`valid`) as well as the baseline (`afl`) on the branch coverage achieved by valid inputs after the 24 hour fuzzing runs. Branch coverage is computed using `gcov` [Stallman et al. 2009]. The plots show means and standard errors of branch coverage across 12 repetitions.

The experiments show that validity fuzzing enables improvement in branch coverage among valid inputs for `libpng` (3%) and `libjpeg-turbo` (39%). For `vorbis`, validity feedback did not appear to have any impact. For `libxml`, the validity fuzzing algorithm produced 30% less branch coverage among valid inputs. Unlike the other benchmarks, which process binary input data, `libxml` expects valid inputs to conform to a context-free grammar. For such a domain, validity fuzzing by itself does not appear to be sufficient. Intuitively, mutating valid XML files using byte-level mutations does not necessarily help produce more valid XML files with diverse code coverage. On `libarchive`, as usual, the domain-specific fuzzing application is not very effective. Since `libarchive` is seeded with an invalid input, most of the inputs generated during the first few hours of fuzzing lead to assumption failures. Naturally, the validity fuzzing algorithm relies on having some valid inputs to begin with in order for its domain-specific feedback to be useful.

With FuzzFactory, we were able to rapidly prototype the validity fuzzing algorithm and evaluate the scenarios in which it does or does not perform well. Note that we did not perform a direct comparison with Zest [Padhye et al. 2019b], which combines validity fuzzing with parametric generators. Such a comparison would not be meaningful, both because Zest it is written in Java, and because it uses mutation and search heuristics that differ from AFL's.

4.5 `cmp`: Smoothing Hard Comparisons

We next describe a novel solution to a well-known problem, that of hard comparisons. Recall the motivating example in Figure 1, which required generating inputs `a` and `b` that were equal to each other. For CGF, similar obstacles arise when encountering operations such as `strncmp`, `memcmp`, and `switch-case` statements. The problem of hard comparisons has been addressed by several researchers in the past [LafIntel 2016; Li et al. 2017; Peng et al. 2018; Rawat et al. 2017; Stephens et al. 2016; Yun et al. 2018]. Common solutions to this problem include, but are not limited to: (1) starting with seed inputs that already satisfy most of the complex invariants, (2) mining magic constants—such as `0x0123`—from the test program and then randomly inserting these values as part of the mutation process, (3) transforming the test program to "unroll" an n-byte comparison into a sequence of branches performing 1-byte comparisons, and (4) performing sophisticated static analysis, dynamic taint analysis, or symbolic execution to identify and overcome hard comparisons. Some solutions, such as statically mining magic constants or unrolling multi-byte comparisons, do

Table 7. `cmp`: Application for smoothing hard comparisons

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \triangleright v = \max(a, v)$	
Hook	**Instrumentation**
entry_point()	**insert_after**(‘$\forall k \in K : dsf(k) \leftarrow 0$’)

bin_expr(*type*, *left*, *op*, *right*)	if $op \in \{$‘==’, ‘! =’$\}$:
	  $k \leftarrow$ *current_program_loc*(), $n \leftarrow$ `sizeof`(*type*)
	  insert_after(‘$dsf(k) \leftarrow \max(dsf(k), comm\_bits(left, right, n)$’)

func_call(*name*, *args*)	if *name* $\in \{$‘`memcmp`’, ‘`strncmp`’, ‘`strncasecmp`’$\}$:
	  $k \leftarrow$ *current_program_loc*()
	  $left \leftarrow args[0], right \leftarrow args[1], n \leftarrow args[2]$
	  insert_after(‘$dsf(k) \leftarrow \max(dsf(k), comm\_bits(left, right, n)$’)

switch(*type*, *val*, *cases*)	for *case* $\in$ *cases* :
	  $k \leftarrow$ *target_program_loc*(*case*), $n \leftarrow$ `sizeof`(*type*)
	  insert_after(‘$dsf(k) \leftarrow \max(dsf(k), comm\_bits(val, case, n)$’)

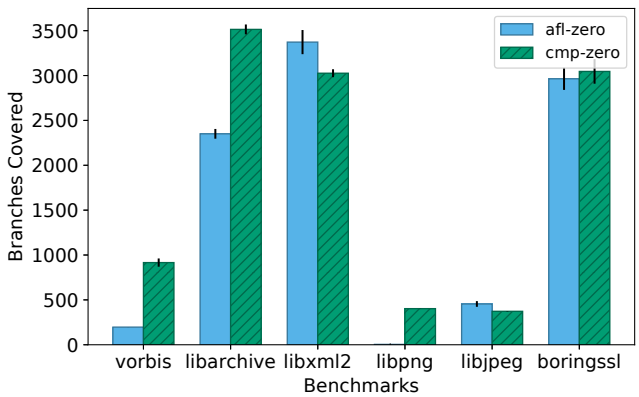


Fig. 7. Branch coverage, as achieved by inputs generated by baseline (afl-zero) and domain-specific fuzzing application (cmp-zero). The suffix `zero` indicates that seed inputs were simply strings of zeros. Higher is better.

not work with hard comparisons of variable-length arguments, e.g. `memcmp(a, b, n)`, where all operands are derived from the program input.

We show how we can prototype a solution for overcoming hard comparisons using FuzzFactory. We do not rely on the domain knowledge in seed inputs or on expensive symbolic analysis. Table 7 describes our domain-specific fuzzing application. The core idea is to provide domain-specific feedback for each comparison operation in the test program ($K = \mathbb{L}$), where the feedback represents the number of bits $V = \mathbb{N}$ that are common between the two operands being compared. The feedback is aggregated using the *max* reduce operator; therefore, a newly generated input will be saved as a waypoint if it maximizes the number of bits that match at any hard-comparison operation in the program under test. Table 7 goes on to describe the program instrumentation strategy. Refer to Table 2 for definitions of *bin_expr*, *switch*, *target_program_loc*, and *comm_bits*. The instrumentation strategy is as follows: First, the DSF map is initialized to 0 at the entry point.

Table 8. `diff`: Application for incremental fuzzing

Domain d: $K = \mathbb{L} \times \mathbb{L}, V = \mathbb{N}, A = 2^{\mathbb{N}}, a_0 = \emptyset,\ a \rhd v = a \cup \log_2(v)$	
Hook	**Instrumentation**
entry_point()	$c \leftarrow$ *current_program_loc*() **insert_after**(‘$\forall k \in K : dsf(k) \leftarrow 0$′) **insert_after**(‘*hits_diff* $\leftarrow$ false′) **insert_after**(‘$p \leftarrow c$′)
new_basic_block()	$c \leftarrow$ *current_program_loc*() if *within_diff*(c): **insert_after**(‘*hits_diff* $\leftarrow$ true′) **insert_after**(‘if *hits_diff* then $dsf(\langle p, c\rangle) \leftarrow dsf(\langle p, c\rangle) + 1$′) **insert_after**(‘$p \leftarrow c$′)

Then, operations such as integer equality, string comparisons, and `switch-case` statements are instrumented. The inserted code populates the DSF map entries corresponding to their program location with the maximum observed count of common bits between their operands.

Experimental evaluation. Figure 7 contains the results of our experiments with this application on the benchmark programs. For this experiment alone, we do not use the initial seed inputs provided in the benchmark suite, but instead seed all fuzzers with an input containing a string of zeros. We do this so that we can study how hard comparisons can be overcome without relying on program-specific knowledge embedded in the seeds. This experiment also simulates a scenario where one wishes to fuzz a program that has an unknown input format, and therefore has no seed inputs available. We evaluate the domain-specific fuzzing application (`cmp-zero`) as well as the baseline (`afl-zero`) on the branch coverage (as computed by `gcov`) achieved by inputs after the 24 hour fuzzing runs. The suffixes `zero` indicate that these experiments did not use meaningful seed inputs. The plots show means and standard errors of branch coverage across 12 repetitions.

From the figure, we see that `cmp-zero` achieves higher code coverage than the baseline in four benchmarks: `vorbis`, `libarchive`, `libpng`, and `boringssl`. Manual investigation revealed that these programs expected their inputs to either contain magic values or to satisfy strict invariants that required hard comparisons. On `vorbis`, the `cmp` front-end achieved 5× more code coverage. On `libpng`, the baseline (`afl-zero`) performed particularly poorly, since the PNG image format requires an 8-byte magic value at the beginning of every input file; the test program exits early if this magic value is not found. The `cmp` front-end effortlessly surpassed this hard comparison and was able to cover over 100× more branches. On `libxml` and `libjpeg-turbo`, the `cmp` front-end does not appear to be useful. In these benchmarks, we did not find any input-dependent hard comparisons between operands larger than two bytes in size. Thus, the baseline approach was sufficient.

4.6 `diff`: Incremental Fuzzing

We now describe another novel application of FuzzFactory: incremental fuzzing after code changes.

It is common practice to let fuzzing tools run for many hours or days in order to find bugs in stable versions of complex software. However, if a developer makes a change to such software, there is currently no straightforward way for them to *quickly* fuzz test their changes. They could use the test corpus generated by the long-running fuzzing session on the previous version of the software as a regression test suite, but those inputs may not exercise code paths affected by the

```
1  int foo(int a, int b) {
2    int d = a;
3    if ((a + b) % 2) {
4  -   d = 2 * a;
4  +   d = 2 - a;
5    }
6    if (a % 3 && a > 0) {
7      return b/d;
8    } else {
9      return 0;
10   }
11 }
```

(a) Program with a diff: the * in Line 4 is modified to a −.

Input	Execution Path
i_1 : a=3, b=4	$\langle 2,4\rangle$, ϟ, $\langle 4,6\rangle$, $\langle 6,9\rangle$
i_2 : a=4, b=4	$\langle 2,6\rangle$, $\langle 6,7\rangle$
i_3 : a=4, b=3	$\langle 2,4\rangle$ ϟ, $\langle 4,6\rangle$, $\langle 6,7\rangle$

(b) Inputs and their execution paths through the program in Figure 8. $\langle x,y\rangle$ designates an executed basic block transition between x and y, and ϟ the hitting of a diff. $\langle x,y\rangle$ highlights the first time an input exercises $\langle x,y\rangle$ *after* hitting the diff during execution.

Fig. 8. Example motivating new post-diff basic block transitions as DSF for incremental (diff) fuzzing.

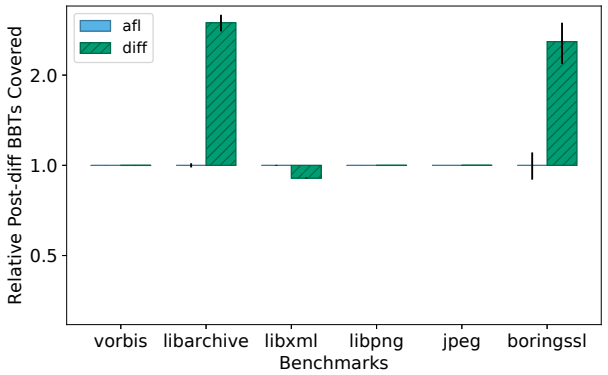


Fig. 9. Relative coverage of basic block transitions after five minutes of incremental fuzzing with the domain-specific `diff` front-end. The baseline is the average coverage achieved by `afl`.

changes to the software. They could also start a new fuzzing session with the previously generated corpus of inputs as the initial seeds. However, they have no way to communicate to the fuzzing engine that it should focus on the code paths affected the changes to the software. Directed fuzzing tools such as AFLGo [Böhme et al. 2017] address this application, but can require several hours of static analysis to pre-compute distances to target program locations[3]. Such approaches may not be practical for use in continuous integration environments where a developer wishes to perform quick regression tests after every code change.

To this end, we propose and implement a domain-specific fuzzing application for incremental fuzzing. The goal of this application is to guide fuzzing towards *quickly* discovering interesting code paths that visit the lines of code that have just been modified. We refer to the set of modified lines of code as the *diff*. To measure the variety of paths executed by the inputs, we will focus on basic block transitions (BBTs) rather than basic blocks alone.

[3] https://github.com/aflgo/aflgo/issues/21

Consider the example program given in Figure 8a. This program performs a division at Line 7. In the original program, the divisor `d` was always a multiple of the input `a`, so the division at Line 7 was always safe. Unfortunately, the new change to the program, which switches `2 * a` to `2 - a` in Line 4, makes a division by zero possible. Figure 8b shows some inputs and the execution paths they take through this program. The execution path is represented as the sequence of BBTs executed by the input. We use $\langle x, y\rangle$ to represent the transition from the basic block starting at line x to the basic block starting at line y. We represent the execution of a diff-affected basic block with the symbol ϟ.

Consider the three inputs in Figure 8b. Input i_1 (`a=3,b=4`) exercises the diff, but not the division at Line 7. Input i_2 (`a=4,b=4`) exercises the division at Line 7, but not the diff at Line 4. Notice that input i_3 (`a=4,b=3`) does not exercise new BBTs compared to inputs i_1 and i_2, so regular coverage-guided fuzzing would not save it. However, input i_3 is the first to exercise the true branch leading to Line 7 *after having hit the diff*. We call the BBTs executed after hitting the diff as *post-diff BBTs*; the newly exercised post-diff BBTs are highlighted in blue in Figure 8b. Since input i_3 covers a new post-diff BBT, it is interesting in an incremental fuzzing setting because it exercises a new code path affected by the change in the diff. In fact, it is only one mutation away from `a=2, b=3`, which would trigger a division by zero.

Our FuzzFactory application, `diff`, ensures that input such as i_3 are saved as waypoints. It does so by populating the DSF map with the number of times each BBT is executed *after* the diff code has been executed (i.e., it must keep track of the BBTs after the ϟ). For example, for input i_1, the DSF map is $\{\langle 4, 6\rangle \mapsto 1, \langle 6, 9\rangle \mapsto 1\}$. For input i_2, the DSF map is $\{\}$ because input i_2 does not hit the diff. Finally, for input i_3, the DSF map is $\{\langle 4, 6\rangle \mapsto 1, \langle 6, 7\rangle \mapsto 1\}$.

Table 8 formally defines the incremental fuzzing domain and describes the instrumentation. Since we keep track of basic block transitions rather than simply basic blocks, $K = \mathbb{L} \times \mathbb{L}$. To better approximate paths, the DSF map collects order-of-magnitude aggregation of BBT execution counts, similar to that used for domain `valid` (ref. Section 4.4). Thus, $A = 2^{\mathbb{N}}$, $a_0 = \emptyset$, and the reducer function is $a \triangleright v = a \cup \log_2(v)$. To keep track of BBTs, the instrumentation adds a global variable p to track the location of the *previously visited* basic block. p is combined with the current block c to create the BBT tuple $\langle p, c\rangle$. This is inspired by AFL's BBT tracking logic [Zalewski 2017].

To make sure that we only track *post-diff* BBTs, the instrumentation also defines a new global variable `hits_diff` in the test program. This variable is set to `false` at the test entry point. At each basic block, the instrumentation adds a check to see whether the basic block is *within_diff*—that is, the basic block was added or modified in the code change of interest—and sets `hits_diff` to `true` if that is the case. Then, the DSF for the BBT $\langle p, c\rangle$ is only incremented if `hits_diff` is `true`, effectively counting only post-diff BBTs.

Experimental evaluation. To simulate the incremental fuzzing environment on our benchmarks without cherry-picking diffs, we perform the following procedure. For each benchmark, we randomly choose one of the saved input directories from our 24-hour runs of AFL on the benchmark. This is our new starting set of test inputs, S_0. To find a relevant code change, we then advance the code repository by one git commit until we find a diff that (1) affects code in the main test driver, and (2) is exercised by at least one input in S_0. We keep advancing through the commit history, and accumulate the diffs, until such a diff is found, or until the most recent commit.

To evaluate utility in a continuous integration environment, we run the tools for *five minutes* each. Since we are interested in evaluating the power of the tools to generate inputs with high code coverage downstream from the diff, we logged any input AFL generated that hit the diff in the five minute run. In our coverage evaluation, we augment AFL's regular saved inputs with these.

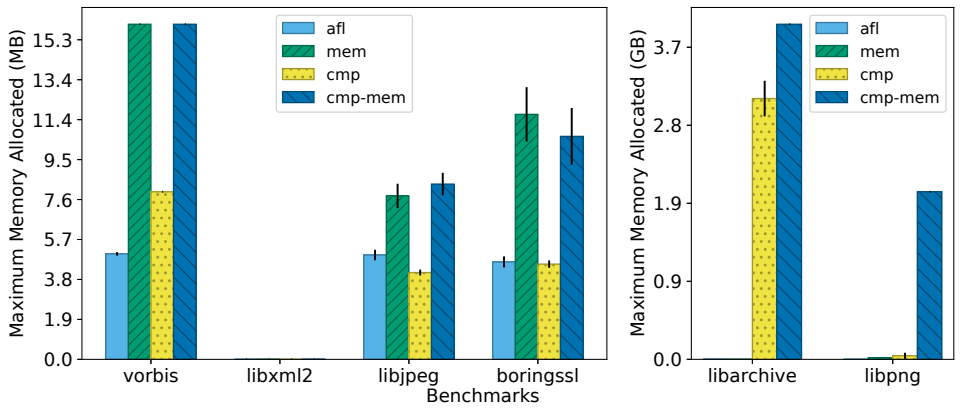


Fig. 10. Evaluation of composing `cmp` and `mem` into the `cmp-mem` domain. Bars show the maximum dynamic memory allocated—in MB on the left and in GB on the right—at a single program location. Higher is better.

Figure 9 contains the results of our 5-minute incremental fuzzing evaluation. The figure plots means and standard errors of the number of post-diff BBTs hit by all generated inputs, relative to the baseline `afl`. We plot the coverage achieved by our domain-specific fuzzing application, called `diff`, relative to `afl`. For `libpng` and `libjpeg-turbo`, the diffs yielded by our procedure were hit by all inputs in the starting corpus, and for `vorbis`, no inputs in the seed corpus initially hit the diff. This resulted in very large diffs. As expected for such large diffs, `diff` and `afl` were equally successful at finding a variety of post-diff behaviors on these benchmarks. For `libarchive` and `boringssl`, only a few inputs hit the initial diff, and the diff was not very large. These more closely mirrored the incremental changes motivated by our techniques. For these benchmarks, the FuzzFactory domain-specific fuzzing application `diff` achieves 2.5-3× more coverage downstream from the diff than `afl`.

4.7 Composing Multiple Domains

Due to the clean separation between domain-specific feedback maps and the underlying fuzzing algorithm, we can easily compose multiple domain-specific fuzzing applications in the same test program binary. Composing two domain-specific fuzzing applications requires no more than incorporating the instrumentation associated with each domain. In our implementation, this is as simple as setting compile-time flags for each domain. Each domain's associated instrumentation only updates its own DSF map. Similarly, our domain-specific fuzzing algorithm aggregates feedback from each registered domain independently (ref. Algorithm 2).

Figure 10 shows the results of our experiments with a composition of `cmp` (ref. Section 4.5) and `mem` (ref. Section 4.3). The goal of this experiment is to maximize memory allocation in the test programs, while also smoothing hard comparisons which may be required to exercise hard-to-reach program branches. This experiment used the initial seed inputs that ship with the benchmark suite. We compare the composite domain (`cmp-mem`) with the baseline (`afl`) as well each independent application (`cmp` and `mem`). For most benchmarks, the composite application `cmp-mem` generates inputs that allocate more (or equal amounts of) memory than those generated by `cmp` or `mem`. In particular, the combined `cmp-mem` application was able to generate inputs that allocate the maximum memory possible with `libarchive` and `libpng`—4GB and 2GB respectively. For `libarchive`, this result is remarkable because the `mem` domain itself performed much worse than the `afl` baseline,

due to the fact that the initial seed inputs were invalid (ref. Section 4.3). However, when combined with the application that smooths hard comparisons, it was able to quickly generate valid archive files and eventually generated a LZ4 bomb: a small input that when decoded leads to excessive memory allocation. Similarly, in `libpng`, the `cmp-mem` application was able to generate a PNG bomb. Unlike the most memory-allocating input discovered by `mem` alone, which was an image that declared very large geometric dimensions in its metadata (ref. Section 4.3), the PNG bomb generated by `cmp-mem` exploits the decoding of `pCAL`/`sCAL` chunks. Such an input demonstrates a known bug: simply capping an image's geometric dimensions does not limit memory usage when decoding PNG files. We can conclude that a composition of the `cmp` and `mem` domains can perform better than the sum of its parts.

New bugs discovered. Since the benchmark suite used in our experiments contains old, historical versions of heavily fuzzed software, we expected to only find previously known bugs, if any, while fuzzing. To our surprise, we found that the inputs saved by `cmp-mem` when fuzzing the January 2017 snapshot of `libarchive` revealed two *previously unknown* bugs in the latest (March 2019) version: a memory leak[4] and an inadvertent integer sign cast that leads to huge memory allocation[5].

5 DISCUSSION

Our framework allows developers and researchers to control the process of fuzz testing by defining a strategy to selectively save intermediate inputs. Our framework does not currently provide any explicit hooks into various other search heuristics used in the CGF algorithm, such as the mutation operators or seed selection strategies. In principle, it should be possible to port general-purpose heuristics such as those used in AFLFast [Böhme et al. 2016] or FairFuzz [Lemieux and Sen 2018] to work with any of the various domain-specific fuzzing applications described in this paper. The work on improving general-purpose fuzzing heuristics is orthogonal to this paper's contributions. Our main contribution is the proposed separation of concerns between the fuzzing algorithm and the choice of feedback from the instrumented program under test.

In theory, a basic increase in code coverage can itself be considered a domain-specific feedback. That is, we could define a domain d where $is\_waypoint(i, \mathcal{S}, d)$ is satisfied when input i leads to the execution of code that is not covered by any input in $\mathcal{S}$. However, in Algorithm 2, we always save an input if it increases code coverage, instead of modeling this criteria through yet another domain. In practice, we found that an increase in code coverage is useful for all domains, since it leads to discovering new program behavior. To put it another way, we always compose every custom domain with a default domain that tries to maximize code coverage. Our implementation allows disabling the default domain via an environment variable if desired.

Since the completion of our experiments for this paper, even more specialized fuzzers that fit our abstraction of *waypoints* have appeared: e.g. (1) Coppik et al. [2019] save inputs that read/write new values to input-dependent memory addresses, and (2) Nilizadeh et al. [2019] discover side-channel vulnerabilities by saving inputs whose execution paths maximally differ from a reference path. We are encouraged by such work, as it strengthens the case for FuzzFactory.

6 IMPLEMENTATION

We have implemented FuzzFactory as an extension to AFL. In FuzzFactory, domain-specific fuzzing applications are implemented by instrumenting test programs. Table 1 described the lines of code required to implement each of the six domains described in this paper. In our applications, we performed instrumentation using LLVM. However, test programs can also be instrumented using

[4] https://github.com/libarchive/libarchive/issues/1165 and CVE-2019-11463
[5] https://github.com/libarchive/libarchive/issues/1237

```
type dsf_t; /* Domain-specific feedback map */

/* Register a new domain. To be invoked once during initialization. */
dsf_t new_domain(int key_size, function reduce, int a_0);

/* Updates to the DSF map. To be invoked during test execution. */
int  dsf_get(dsf_t dsf, int k);              // return dsf[k]
void dsf_set(dsf_t dsf, int k, int v);       // dsf[k] = v
void dsf_increment(dsf_t dsf, int k, int v); // dsf[k] = dsf[k] + v
void dsf_union(dsf_t dsf, int k, int v);     // dsf[k] = dsf[k] | v
void dsf_maximize(dsf_t dsf, int k, int v);  // dsf[k] = max(dsf[k], v)
```

Fig. 11. API for domain-specific fuzzing in pseudocode.

any other tool, such as Intel's Pin [Luk et al. 2005]. In fact, domain-specific fuzzing applications can also be implemented by manually editing test programs to add code that calls the FuzzFactory API. We next describe this API.

6.1 API for Domain-Specific Fuzzing

Figure 11 outlines the API provided by FuzzFactory. The type `dsf_t` defines the type of a domain-specific map. In our implementation, the keys and values are always 32-bit unsigned integers. However, users can specify the size of the DSF map; that is, the number of keys that it will contain.

The API function `new_domain` registers a new domain whose key set K contains `key_size` keys. The arguments `reduce` and `a_0` provide the reducer functions (of type `int x int -> int`) and the initial aggregate value respectively. For the `slow` domain, `key_size` is 1. For applications where K is a set of program locations $\mathbb{L}$, we use `key_size` of 2^{16} and assign 16-bit pseudorandom numbers to basic block locations, similar to AFL. For the incremental fuzzing applications, where $K = \mathbb{L} \times \mathbb{L}$, we use a hash function to combine two basic block locations into a single integer-valued key. The sets V and A are defined implicitly by the usage of DSF maps and the implementation of the `reduce` function. For applications such as validity fuzzing, where A is a set of orders of magnitude, we use bit-vectors to represent sets.

The function `new_domain` returns a handle to the DSF map, which is then used in subsequent APIs listed in Fig. 11, such as `dsf_increment`. Calls to the `new_domain` are inserted at test program startup, before any tests are executed. It is up to the user to ensure that the provided reducer function satisfies properties 1 and 2, which in turn guarantee monotonic aggregation (Theorem 1). API functions that start with '`dsf_`' manipulate the DSF map. The argument `key` must be in the range `[0, key_size)`.

7 RELATED WORK

To the best of our knowledge, FuzzFactory is the first framework for implementing domain-specific fuzzing applications. JQF [Padhye et al. 2019a] allows users to implement custom fuzzing algorithms for Java; unlike FuzzFactory however, the instrumentation is fixed, while the search algorithm can be customized. The LLVM-based Clang compiler [Lattner and Adve 2004] provides a customizable tracing framework for C/C++ programs. With the use of command-line flags such as `-fsanitize-coverage`, one can ask Clang to instrument basic blocks and comparison operations to call specially named functions; users can link-in custom implementations of these functions to trace program execution. LibFuzzer [LLVM Developer Group 2016] uses these hooks to provide feedback from a program under test in order to perform coverage-guided fuzzing. However, `libFuzzer` does

not provide a mechanism to provide arbitrary domain-specific feedback with custom aggregation functions. That is, while LLVM provides hooks into a program's execution, there is currently no way to communicate information to the fuzzing algorithm. However, it is relatively easy to use LLVM's tracing hooks to call into FuzzFactory's API for domain-specific fuzzing.

A lot of research in the field of fuzz testing targets a general-purpose improvement in the search process, as surveyed by Manès et al. [2018]. These techniques usually adapt the various heuristics used in the fuzzing algorithm [Böhme et al. 2016; Chen and Chen 2018; Lemieux and Sen 2018], or seek to combine fuzz testing with heavyweight approaches such as concolic execution [Ognawala et al. 2018; Stephens et al. 2016; Yun et al. 2018]. Our proposed design does not conflict with any of these techniques. General-purpose tweaks to the fuzzing process can be applied to Algorithm 2, without affecting the mechanism for collecting domain-specific feedback.

Structured fuzzing tools such as protobuf-mutators [Serebryany et al. 2017], AFLSmart [Pham et al. 2018], Nautilus [Aschermann et al. 2019], and Superion [Junjie Wang and Liu 2019] leverage domain-specific information about the input format expected by the program under test. Such approaches can be combined with the validity fuzzing domain presented in Section 4.4, to overcome the limitations that we observed with formats such as XML [Padhye et al. 2019b].

8 CONCLUSION

We presented FuzzFactory, a framework for implementing domain-specific fuzzing applications. Our framework provides a mechanism for communicating to a fuzzing engine, arbitrary domain-specific feedback during the execution of a program under test. Our experiments with six front-ends demonstrates that FuzzFactory can be used to prototype domain-specific applications without changing the underlying search algorithm. The effectiveness of domain-specific feedback varies based on the nature of test programs, the objective, and the initial seed inputs. Our hope is that our proposed framework will enable researchers to quickly develop highly specialized domain-specific solutions and advance the state-of-the-art.

A MONOTONICITY OF AGGREGATION

Lemma 1 (No ping-pong). *Given a reducer function* $\triangleright : A \times V \longrightarrow A$ *satisfying Properties 1 and 2, then* $\forall a \in A$ *and any* $n \geq 0$ *terms* $v_1, \ldots, v_n \in V$*, if* $a \triangleright v_1 \triangleright \ldots \triangleright v_n = a$*, then:*

$$\forall 0 \leq k \leq n : a \triangleright v_1 \triangleright \ldots \triangleright v_k = a$$

In other words, if we start with aggregate value a and then apply n reductions, and if the final result is also the value a, then the result of all the intermediate reductions must also be a. This lemma states that aggregate values cannot *ping-pong*; that is, they cannot oscillate between distinct values.

Proof. For $n = 0$, the lemma is trivially true. For $n > 0$, we prove the lemma by contradiction: given that $a \triangleright v_1 \triangleright \ldots \triangleright v_n = a$, assume that there exists some k, where $1 \leq k \leq n$, such that $a \neq a \triangleright v_1 \triangleright \ldots \triangleright v_k$. In this inequality, we can substitute the value of a on both sides with the equivalent $a \triangleright v_1 \triangleright \ldots \triangleright v_n$, to get:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright \ldots \triangleright v_n \triangleright v_1 \triangleright \ldots \triangleright v_k$$

Then, we can repeatedly apply Property 2 on the right-hand side to rearrange terms:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright v_1 \triangleright v_2 \triangleright v_2 \triangleright \ldots \triangleright v_k \triangleright v_k \triangleright v_{k+1} \triangleright v_{k+2} \triangleright \ldots \triangleright v_n$$

Then, we can repeatedly apply Property 1 on the right-hand side to remove redundant terms:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright \ldots \triangleright v_n$$

This is a contradiction; therefore, no such k can exist. □

Definition 3 (Progress). If $\triangleright : A \times V \rightarrow A$ is a reducer function, then we can define a binary relation $\preceq$ on A called *progress* as follows:

$$a \preceq b \Leftrightarrow \exists\, v_1, \ldots, v_n \in V, \text{ where } n \geq 0, \text{ such that } a \triangleright v_1 \triangleright \ldots \triangleright v_n = b$$

Lemma 2 (Reflexivity of progress). *If $\triangleright : A \times V \rightarrow A$ is a reducer function and $\preceq$ is its progress relation, then $\forall a \in A : a \preceq a$.*

Proof. Straightforward from Definition 3 with $n = 0$. □

Lemma 3 (Transitivity of progress). *If $\triangleright : A \times V \rightarrow A$ is a reducer function and $\preceq$ is its progress relation, then $\forall a, b, c \in A : a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$.*

Proof. If $a \preceq b$ and if $b \preceq c$, then by Definition 3 there exist some terms $u_1, \ldots, u_m \in V$ and $v_1, \ldots, v_n \in V$ for $m, n \geq 0$ such that:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m = b \tag{6}$$

$$b \triangleright v_1 \triangleright \ldots \triangleright v_n = c \tag{7}$$

Substituting the b on the LHS of Equation 7 with the LHS of Equation 6, we can write:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m \triangleright v_1 \triangleright \ldots \triangleright v_n = c \tag{8}$$

Which, by Definition 3, means $a \preceq c$. □

Lemma 4 (Anti-symmetry of progress). *If $\triangleright : A \times V \rightarrow A$ is a reducer function and $\preceq$ is its progress relation, then $a \preceq b \wedge b \preceq a \Rightarrow a = b$.*

Proof. If $a \preceq b$ and if $b \preceq a$ then by Definition 3 there exist some terms $u_1, \ldots, u_m \in V$ and $v_1, \ldots, v_n \in V$ for $m, n \geq 0$ such that:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m = b \tag{9}$$

$$b \triangleright v_1 \triangleright \ldots \triangleright v_n = a. \tag{10}$$

Substituting the b on the LHS of Equation 10 with the LHS of Equation 9, we can write:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m \triangleright v_1 \triangleright \ldots \triangleright v_n = a.$$

By Lemma 1, all intermediate aggregates must be equal to a, in particular:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m = a$$

Plugging this result into the LHS of Equation 9, we get $a = b$. □

Proof of Theorem 1. Let $\preceq$ be the progress relation for the reducer $\triangleright$. From Lemmas 2, 3, and 4, it follows that this relation is a *partial order*. Now, let $S_1 \subseteq S_2$. From the definition of $\mathcal{A}$ in Equation 3, we can write:

$$\mathcal{A}(S_2, k, d) = \mathcal{A}(S_1, k, d) \triangleright v_1 \triangleright \ldots v_n$$

where $\{v_1, \ldots, v_n\} = S_2 \setminus S_1$. From Definition 3, this implies that $\mathcal{A}(S_1, k, d) \preceq \mathcal{A}(S_2, k, d)$; that is, $\mathcal{A}$ is *monotonic* in its first argument with respect to $\preceq$. □

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback, which helped us improve the paper. This research is supported in part by gifts from Samsung, Facebook, and Futurewei, by NSF grants CCF-1409872 and CNS-1817122, and by the AWS Cloud Credits for Research program.

REFERENCES

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. Nautilus: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium (NDSS '19)*.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.

Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*.

Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 48–58.

Google. 2019a. Continuous fuzzing of open source software. https://opensource.google.com/projects/oss-fuzz. Accessed March 26, 2019.

Google. 2019b. Set of tests for fuzzing engines. https://github.com/google/fuzzer-test-suite. Accessed March 20, 2019.

Lei Wei Junjie Wang, Bihuan Chen and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *41st International Conference on Software Engineering (ICSE '19)*.

Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '18)*. ACM, New York, NY, USA, Article 28, 8 pages. https://doi.org/10.1145/3240765.3240842

LafIntel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. Accessed March 20, 2019.

Chris Lattner and Vikram Adve. 2004. 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 (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*.

Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

LLVM Developer Group. 2016. libFuzzer. http://llvm.org/docs/LibFuzzer.html. Accessed March 20, 2019.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140

Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DifFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 176–187. https://doi.org/10.1109/ICSE.2019.00034

Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1475–1482. https://doi.org/10.1145/3167132.3167289

Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019a. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019b. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New

York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019c. Validity Fuzzing and Parametric Generators for Effective Random Testing. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 266–267. https://dl.acm.org/citation.cfm?id=3339777

Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017a. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 615–632.

Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017b. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2018. Smart Greybox Fuzzing. *CoRR* abs/1811.09447 (2018). arXiv:1811.09447 http://arxiv.org/abs/1811.09447

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.

Kostya Serebryany, Vitaly Buka, and Matt Morehouse. 2017. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator.

Richard M. Stallman et al. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 745–761. http://dl.acm.org/citation.cfm?id=3277203.3277260

Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed March 20, 2019.

Michał Zalewski. 2017. American Fuzzy Lop Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed March 20, 2019.