Chapter 1

Introduction

Software bugs remain pervasive in modern software systems. As software becomes increasingly intertwined in all aspects of our lives, the consequences of these bugs become increasingly severe. For instance, in the last few years, several important security vulnerabilities have emerged from basic correctness bugs in software [101, 155, 89].

Though not all bugs cause important security vulnerabilities, their overall cost remains high: one estimate puts the cost of dealing with software failures and defects *in the United States alone* in 2018 at nearly USD$1,540,000,000 [108].

This dissertation explores methods to help developers improve the quality of software before bugs ship into productions and incur these costs. The particular focus is on *fuzz testing* or *fuzzing* tools. These tools automatically find bug-inducing inputs in programs; inputs x that, when run on a program p, cause the program to crash.

Access to such tools greatly helps reduce the time to find and fix bugs, and thus, reduces the impact of these bugs. As a concrete example, consider a case study of the OpenSSL library [73]. In 2012, a bug was introduced into the library which allowed an attacker to read memory from a server that did not belong to them [144]. This bug, later named Heartbleed, was only discovered in 2014. By that time, it had spread widely, and was exploited to great cost. For instance, it enabled the leak of millions of patient records from the U.S.'s second largest hospital chain [76]. It also forced the Canada Revenue Agency website to shut down for several days [116], but not before over 900 Social Insurance Numbers were leaked from the agency's website [143].

By contrast, nearly 4 years later, another bug was introduced into OpenSSL. This bug could have allowed an attacker to execute arbitrary code, and was judged as much more severe than Heartbleed [145]. However, there are no widely publicized costs associated with this bug, CVE-2016-6309. This may be in large part because it was discovered one day after its introduction by a modern fuzz testing tool, honggfuzz [174].

The term *fuzz testing* dates back to Miller et al.'s seminal work from 1990 [135], but has re-emerged as an area of interest following the appearance of the pioneering AFL [185] around 2014. Fuzz testing tools assume the following program setting. There is a program p which runs on some input x. For the purposes of this dissertation, differences in the behavior of p

on different executions should be entirely explained by differences in the input x. That is, the program should be deterministic, single-threaded, and relatively fast-running. In the classical fuzzing domain, the input x is typically a file—modelled as a sequence of bytes—processed by the program p, but it can also be a more general data structure.

Fuzzing tools then repeatedly run the program p on inputs x that are generated via a variety of random search procedures, until an input x^* is found which induces a bug in p. Typically, these tools identify the presence of a bug via universal correctness oracles — if the program crashes or exits earlier due to an assertion error. The pure random fuzzing introduced by Miller et al. [135] simply created the inputs x by sampling random sequences of bytes, but modern fuzzing algorithms are much more sophisticated in their input creation methodology. Many modern fuzzing algorithms rely on detailed program feedback to guide the generation of inputs (Chapters 3, 4, 5, 6) or rely on user-provided information about input structure to make the search more targeted (Chapters 6, 7, 8).

Overall, this dissertation presents several methods to make fuzz testing tools more widely applicable and easy-to-use. This dissertation is divided into three main parts, each of which identifies a different key component of modern fuzzing algorithms, and how to generalize this component for different fuzzing goals: new bug domains, deeper program exploration, and applications beyond testing. Chapter 2 provides a discussion of related work, including detailed background of a particular coverage-guided fuzzing algorithm, AFL. The rest of this introduction describes the two modern fuzzing algorithms studied in this dissertation, and the key drawbacks addressed in each subsequent part of the dissertation.

1.1 Coverage-Guided Fuzzing

In the mid-2010s, coverage-guided fuzzing (CGF) emerged as one of the most effective techniques for fuzzing real-world software. Pioneered by AFL [185], CGF has been implemented in several other popular tools including libFuzzer [167] and honggfuzz [174].

Like random fuzzing [135, 133]—which sends many random inputs to the program under test—CGF works by executing a test program with a large number of inputs generated via random search. However, instead of generating totally random inputs from scratch, CGF mutates inputs from a set of saved inputs to derive new inputs. Algorithm 1 walks through the high-level algorithm shared by most implementations of CGF.

The CGF algorithm takes an input an instrumented program (p) and a set of user-provided *seed inputs* (S_0). Again, typically this program accepts as input a byte-sequence (either via a file or standard input), and thus, each input in S_0 is modelled as a sequence of bytes.

CGF maintains two global states: (1) $\mathcal{S}$, 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 $\mathcal{S}$. The instrumented test program p returns the coverage achieved by the input it is executed on. The forms of coverage most commonly used by CGF are branch coverage, basic block coverage, or basic block transition coverage. $\mathcal{S}$ is initialized to the set of

Algorithm 1 The generic 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 ← initCoverage(S_0)
3:  repeat                                                    ▷ Main fuzzing loop
4:    for input in S do
5:      with probability fuzzProb(input) do
6:        for 1 ≤ i ≤ numChildren(input) do
7:          input' ← mutate(input)                  ▷ Generate new test input input'
8:          coverage ← execute(p, input')                    ▷ Run test with input'
9:          if coverage ⊈ totalCoverage then
10:           S ← S ∪ {input'}                    ▷ Save input' if it covers new code
11:           totalCoverage ← totalCoverage ∪ coverage
12: until given time budget expires
13: return S
```

user-provided seed inputs (Line 1) and *totalCoverage* is initialized to the coverage achieved by those user-provided seed inputs (Line 2).

The main coverage-guided fuzzing loop goes over each input *input* in the set of inputs $\mathcal{S}$. With some probability determined by an implementation-specific heuristic function fuzzProb, CGF decides whether to mutate the input *input* or not (Line 5). If *input* is selected for mutation, CGF decides on how many mutant inputs to generate from *input* via another heuristic function numChildren (Line 6).

Then, numChildren(*input*) times, CGF randomly mutates *input* to generate a mutant *input*′. A random mutation typically involve choosing a random set of bytes in the inputs and performing a mutation operation there, like: bit flipping, byte flipping, incrementing and decrementing integer values, or replacing bytes with "interesting" integer values (0, `MAX_INT`). CGF executes the program with the newly generated input and collects the coverage of the input in the temporary variable *coverage* (Line 8). If the observed coverage contains some previously-unseen coverage points (Line 9), the new input *input*′ is saved to the set of inputs $\mathcal{S}$ (Line 10). This *input*′ can now be mutated during a future iteration of the fuzzing loop. The process repeats until the time budget expires.

1.1.1 Drawback: Fixed Testing Goal

Coverage-guided fuzzing tools were conceived as catch-all security auditing tools, to automatically find input-dependent crashes in the program under test. For this broad testing goal, where the location of a bug is unknown, increased coverage is a reasonable testing signal under which to save inputs. The logic is that this encourages broad exploration of paths

through the program, and that exercising all paths in a program should reveal any bugs in the program.

However, not all bugs are uniformly distributed throughout the program under test, and furthermore, certain bugs are not revealed by branch coverage alone. For instance, a security auditor may know that bugs in a particular program component are likely to be more severe, and want to direct input generation to exercise that component. Or, if an input causes a negative data value to flow into the argument of a memory allocation, the program under test may use an unreasonable amount of memory, even if the coverage of the program under test is not abnormal for that input.

Unfortunately, the notion of interesting inputs being those that increase branch coverage is generally quite tightly integrated into coverage-guided fuzzing tools, and so, these testing goals cannot be achieved. Chapters 3 and 4 address this drawback directly. Chapter 3 introduces a multi-objective feedback-directed fuzzing algorithm and shows that this algorithm can be used to effectively find inputs with pathological performance behavior. Chapter 4 further generalizes this feedback-directed fuzzing algorithm, and introduces a framework that allows for easier customization of the testing goal.

1.1.2 Drawback: Malformed Inputs

Part of the success of random testing methods, including random fuzzing, comes from the fact that these methods produce inputs outside of the input space the software developer has reasoned about. As such, random testing quickly reveals the program's behavior in exceptional—and often buggy—circumstances.

Coverage-guided fuzzing builds on this intuition as well, with a twist. By generating inputs via small mutations to existing inputs, it can better ensure that the inputs still get to some interesting program state (by relying on reasonable inputs to start with), while still exploring corner cases (revealed via the small mutations of the input). This means that the inputs generated by CGF find deeper corner cases in input validation than an approach that relies on purely random sequences of characters as input.

However, the byte-level mutations applied by CGF are still likely to generate inputs that are, overall, malformed. This is great to stress test parsers, but means that it is very difficult for coverage-guided fuzzing to consistently generate inputs that pass the validation checks of a program and explore its core logic. Chapters 5 and 6 address this drawback. In particular, Chapter 5 describes a mutation masking approach that enables CGF to explore the program under test more deeply without any additional input from the user. Chapter 6 presents an approach that can explore the core logic of programs even better, but relies on a user-written generator to perform high-level mutations during coverage-guided fuzzing.

```
# Generate a (possibly negated) integer as a string
def generate_unexpr():
    val = str(random.integer())
    if random.boolean():
        val += "-"  + val
    return val

# Generate a binary expression
def generate_binexpr():
    lhs = generate_expr()
    op = random.choice(["-", "+", "/", "*"])
    rhs = generate_expr()
    return lhs + op + rhs

# Generate a (possibly parenthesized) expression
def generate_expr():
    if random.boolean():
        expr = generate_binexpr()
    else:
        expr = generate_unexpr()
    if random.boolean():
        expr = "(" + expr + ")"
    return expr
```

Figure 1.1: Generator of expressions in a simple calculator language.

1.2 Generator-Based Fuzzing

Coverage-guided fuzzing relies heavily on feedback from the programs's execution in its input generation strategy because it has so little information about the structure of inputs to the program under test. When information about input structure is available, the input generation strategy can be made much more effective.

This is the key of the effectiveness of generator-based fuzzing (also called *property-based testing* [58]). The core idea is to create new inputs by repeatedly calling a (typically, user-written) generator. This generator is a non-deterministic function, which, each time it is called, returns a random input in a given search space. An example generator is given in Figure 1.1: each time `generate_expr` is called, it returns an input in a simple calculator language, e.g. "`14`", "`(3) + 119`", or "`(87+1230-(467/234*2))`".

Given a generator like that in Figure 1.1, the fuzzing process is quite straightforward. Given a program p, repeatedly call the generator to generate an input x. As in coverage-guided fuzzing, run p on x to observe whether x induces a bug (i.e., a crash or assertion failure).

While the generator in Figure 1.1 still generates byte-sequence inputs, one can easily write similar generators for data structures. The term *property-based testing*, introduced by QuickCheck for Haskell [58], is used to refer to generator-based fuzzing in a more unit testing context. There, the program $p(x)$ takes in inputs of a type $\mathcal{X}$ and should encode a property $P(x) \Rightarrow Q(x)$ to be tested. For some types $\mathcal{X}$, a generator can be derived automatically from the type definition. By generating many inputs $x \in \mathcal{X}$ and running them through p, the developer can approximately check that the property $\forall x, P(x) \Rightarrow Q(x)$ holds.

1.2.1 Drawback: Coupling of Distribution and Search Space

The key drawback of generator-based fuzzing emerges when the domain of the generator does not closely match the space of "interesting" inputs for the program under test. In particular, if the program encodes a property $P(x) \Rightarrow Q(x)$, and very few inputs x generated by the generator satisfy $P(x)$, then most of the runs of the program under test are simply validating the case in which the property is vacuously true.

For example, for the generator of expression in Figure 1.1, suppose the developer wants to validate that in the presence of a division by zero, an exception is thrown *before* the evaluation of a division by zero. So $P(x)$ is true for inputs x which contain a division by zero. But many of the inputs sampled from the generator in Figure 1.1 will not include division by zero. The developer could spend some time tuning the generator in Figure 1.1 to increase the probability of generating inputs with a division by zero, for example, by forcing more generation of division expressions with the denominator as the integer literal `0`.

Requiring the developer to conduct this tuning comes with its own set of challenges. First, of course, tuning a generator so that most of the inputs x it generates satisfies $P(x)$ is a significantly more complex task than writing a simple generator of inputs. Second, as a human tries to tune the generator to a smaller input space, they may prune the space of inputs generated unnecessarily. There are many ways to write an expression that has a division by zero, e.g. "`4/(6-3*2)`". If the developer did the simple "add more divisions by `0`" tuning described above, they may prevent the generation of such expressions. The problem gets worse as the input space becomes more complex, e.g. inputs in a programming language.

The core problem here is that a generator is a specification of a search space (the domain of inputs x which can be generated by some path through the generator) paired with a probability distribution over that search space (the probability of any particular path being taken through the generator). If we could simply adjust the probability distribution from which inputs are drawn, we could automatically adapt a generic generator to a particular $P(x)$ without requiring the user to think about distributions. Chapter 6 discusses a method to increase the number of valid inputs generated by indirectly adjusting these probability distributions via coverage-guided fuzzing. Chapter 7 presents a reinforcement learning approach to tune these distributions more directly. Finally, Chapter 8 shows how this same abstraction—splitting the generator of elements from the distribution from which those elements are drawn—can be used for program synthesis.

Chapter 2

Background and Related Work

In this section, we first provide additional background on AFL [185], and then discuss the numerous fuzz testing and input-generation tools related to the work in this dissertation.

The discussion of related work focuses on those which are the most pertinent to this work in this dissertation. Some survey papers provide additional context. Valentin et al. present a unified model of fuzzing—blackbox, greybox and whitebox—, as well as a genealogy of significant fuzzers up to 2019 [127]. The model of coverage-guided fuzzing in this dissertation is less overarching, but instead focuses on the core components that must be altered for different testing goals. Godefroid's 2020 review overview the field at a higher-level, highlighting the *differences* between different branches of fuzzing, rather than unifying them in a single model [79]. Klees et al. discuss some of the common issues in evaluating different fuzzers, and how to build more consistent baselines [106].

2.1 American Fuzzy Lop (AFL)

American Fuzzy Lop, or AFL [185], was the first tool to introduce the coverage-guided fuzzing algorithm, which we discussed previously.

A number of the algorithms presented in this dissertation (Chapters 3, 4, 5) are implemented directly on top of AFL. For context of the implementation details of these algorithms, we detail the way in which AFL implements the abstract coverage-guided fuzzing algorithm introduced in Algorithm 1. The main implementation-specific points are: (1) how coverage is gathered, (2) how inputs are selected for mutation (FUZZPROB in Line 5), (3) how many mutants are produced (Line 6), and (4) how mutations are performed (Line 7).

AFL Coverage Calculation The notion of coverage collected by AFL (*coverage* in Line 8 of Algorithm 1) while the program under test executes, and its use in the fuzzing loop, is one of AFL's key innovations.

In order to collect coverage information efficiently, AFL inserts instrumentation into the program under test. To track coverage, it first associates each basic block with a random

number via compile-time instrumentation. The random number is treated as the *unique ID* of the basic block. The basic block IDs are then used to generate unique IDs for the transitions between pairs of basic blocks. In particular, for a transition from basic block A to B, AFL uses the IDs of each basic block—$ID(A)$ and $ID(B)$, respectively—to define the ID of the transition, $ID(A \to B)$, as follows:

$$ID(A \to B) \stackrel{def}{=} (ID(A) \gg 1) \oplus ID(B).$$

Where $\oplus$ designates bitwise exclusive or (xor). Right-shifting ($\gg$) the basic block ID of the transition start block (A) ensures that the transition from A to B has a different ID from the transition from B to A. In this dissertation, in particular in Chapter 5, we associate the notion of basic block transition with that of a *branch* in the program, unless stated otherwise.

The coverage of the program under test on a given input is collected as a set of pairs of the form (*branch ID*, *branch hits*). If a (*branch ID*, *branch hits*) pair is present in the coverage set, it denotes that during the execution of the program on the input, the branch with ID *branch ID* was exercised *branch hits* number of times. The branch hits are simplified into one of 8 buckets: hit 1 time, 2 times, 3 times, 4–7 times, 8–15 times, 16–31 times, 32–127 times, or 128–255 times. AFL calls this set of pairs the "path" of an input. AFL says that an input achieves *new coverage* if it discovers a new (*branch ID*, *branch hits*) pair.

Choosing which input to mutate. In Line 5 of Algorithm 1, an input is selected from the set of saved inputs for mutation with a probability FUZZPROB. To compute this, AFL first determines a set of *favored* inputs. AFL assigns a FUZZPROB of 100% to favored inputs that have not yet been mutated, and 1% to non-favored or already-mutated inputs. If the set of favored inputs is empty, it relaxes this a little, assigning 25% chance of mutating not already-mutated inputs, and a 5% chance of mutating already-mutated inputs.

The notion of favored input is crucial in this computation. AFL creates this set in a greedy manner. For each *branch ID* that has been covered, it finds the input with the smallest product of execution time and input length. This input is the winner for that *branch ID*. AFL assigns a winner as favored if it also covers a *branch ID* that had not been seen in the previous fuzzing iteration.

Choosing the number of mutants and mutating inputs. In the abstract CGF algorithm we separated the process of choosing the number of mutants to produce (Line 6) from the mutation process (Line 7). In AFL, these components are tightly intertwined. AFL's mutation strategies assume the input to the program under test is a sequence of bytes, and can be treated as such during mutation. AFL mutates inputs in two main stages: the *deterministic* stages and the *havoc* stage.

All the deterministic mutation stages operate by traversing the input under mutation and applying a mutation at each position in this input. These mutations include bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with "interesting" integer values (0, `MAX_INT`), etc. The number of mutated inputs produced in each

Algorithm 2 "Havoc" mutations in AFL.

```
1: procedure MUTATEHAVOC(Prog, input)
2:     numMutations ← RANDOMBETWEEN(1,256)
3:     newinput← input
4:     for 0 ≤ i <numMutations do
5:         mutation ← RANDOMMUTATIONTYPE
6:         position ← RANDOMBETWEEN(0, |newinput|)
7:         newinput ← MUTATE(newinput, mutation, position)
```

of these stages is governed by the length of the input being mutated. So, if the deterministic mutation stages are not skipped, part of NUMCHILDREN is simply a linear function of the input length.

On the other hand, the havoc stage works by applying a sequence of random mutations to the input being mutated to produce a new input. Algorithm 2 shows this process. Several mutations are repeatedly applied to the original input (Line 7) before running it through the program. The type of mutation that can be applied includes all the mutations that can be applied in the deterministic stages.

The number of total havoc-mutated inputs to be produced is determined by a performance score, which is the non-input-length dependent part of NUMCHILDREN. This performance score is higher for inputs with (a) faster execution times, (b) higher average coverage, (c) which have been more recently added to the saved input set, and (d) which have been discovered "deeper" in the fuzzing process (a mutant of a seed input has depth 1, a mutant of that mutant has depth 2, etc.).

Finally, AFL also includes a crossover-like mutation phase (called *splicing*) which combines sequences of the parent input with sequences of other saved inputs. None of algorithms discussed in this dissertation alter this stage of mutation.

Trimming Inputs Not illustrated in Algorithm 1 is an additional trimming stage before inputs are mutated. For additional efficiency, inputs are trimmed with an approximate delta-debugging method [191], which tries to reduce the size of inputs as much as possible while ensuring they cover the same "path" (the set of *(branch ID, branch hits)* pairs).

2.2 Random and Mutational Fuzzers

The term *fuzz* was introduced by Miller et al.'s seminal work on randomized testing of UNIX utilities [135]. After observing that random characters caused by rain on dial-up phone lines could cause remote command-line utilities to crash, Miller et al. developed the `fuzz` tool. This tool stress-tests command line utilities by repeatedly sending random sequences of characters as input to these utilities. This original work also included the `ptyjig` tool, essentially a generator-based fuzzer for interactive utilities.

This study found that numerous popular utilities—including `bc`, `emacs`, `ftp` `telnet`, and `vi`—could be crashed with inputs generated by this pure random fuzzing. The study was revisited five years later [136], and though fewer UNIX utilities could be caused to crash with this random fuzzing, many of the bugs were still present, and more than half of X-Window utilities could be crashed with the process. Subsequent studies on Windows NT GUI applications [72] and Mac OS utilities and GUI applications [134] found similar results. Finally, a 2020 study found that this method was still effective at finding crashes, showing the persistent prevalence of pointer and array-related bugs in C/C++ code [133].

A step more sophisticated than these pure random fuzzers are *mutational* fuzzers, which generate inputs by mutating some seed inputs, but without the genetic-algorithm-like loop introduced by AFL. Traditional blackbox mutational fuzzers such as zzuf [95] mutate user-provided seed inputs according to a mutation ratio, which may need to be adjusted to the program under test. BFF [100] and SymFuzz [53] adapt this parameter automatically. BFF measures the crash density—the proportion of generated inputs that cause crashes—for different mutation ratios, and adjusts the mutation ratio accordingly. SymFuzz conducts the heavier duty strategy of input-bit dependence inference to adapt the mutation ratio to the program under test.

Radamsa is a modern mutational fuzzer which has found numerous CVEs in sophisticated software components such as Chrome, Mozilla Firefox, Adobe Reader and CISCO WebVPN [29]. One of its appeals is its very simple command line interface, which is in part due to its black-box nature. Unlike traditional mutational fuzzers, it has some sophistication in its mutation techniques, i.e. it can recognize parts of inputs that look like numbers and mutate them as numbers rather than treating them as random bytes.

2.3 Coverage-Guided Fuzzers

Coverage-guided fuzzing piqued the interest of the practitioner and academic community after the publication of some impressive results on the part of AFL, which we have already discussed extensively. Particularly impressive results include the automatic discovery of JPEG structure [187] and the shellshock bug [186].

Several other industrial coverage-guided fuzzers were built after this. First is libFuzzer [167], integrated into the LLVM-based `clang` compiler [113] toolchain. The underlying input-generation method is fairly similar to AFL; a notable difference is that only one mutant is generated per parent (i.e. numChildren in Line 6 of Algorithm 1 always returns 1) and that the search terminates after the discovery of any crash. Unlike AFL, which operates by repeatedly calling a compiled external program under test, libFuzzer is *in-process*, meaning it repeatedly runs a single test driver function. This test driver takes in a sequence of bytes and the length of that sequence as input. In order to run libFuzzer properly, the test driver should be side-effect free, not leak memory, etc., so that it can be invoked hundreds of thousands of time. Unlike AFL whose feedback instrumentation is fixed, libFuzzer provides hooks that allow users to inject extra instrumentation at basic blocks and comparison operations.

Another notable coverage-guided fuzzer is honggfuzz [174]. It supports the same test driver abstraction as libFuzzer. The distinguishing feature of honggfuzz is its high-performance aspect: it supports multi-process and multi-threaded fuzzing out-of-the-box, as opposed to requiring the launch of multiple fuzzing runs. It is the fuzzer that found the notable critical security vulnerability in OpenSSL mentioned in the introduction [145].

AFLFast [46] is the seminal work on improving coverage-guided fuzzing in academia. It presents a Markov Chain model of the fuzzing process, and the hypothesis that inputs which exercise low-frequency paths —coverage sets exercised by very few fuzzer-generated inputs—are more likely to produce inputs finding new coverage in the program under test. It built from this model several improvements to the numChildren, that caused relatively larger emphasis on the havoc mutation stages of inputs exercising rare paths. This led to large increases in coverage and decreases in time to reveal bugs compared to AFL, and inspired improvements of some of AFL's default heuristics [188].

VUzzer [161] adds several smarter data-flow and control-flow analyses to the coverage-guided fuzzing process. In a pre-fuzzing static analysis stage, VUzzer identifies basic blocks containing error handling code, in order to de-prioritize them during fuzzing, as well as deeply nested basic blocks, in order to prioritize them during fuzzing. VUzzer also uses taint analysis to improve its mutation strategies, for example to determine which input bytes are used in direct comparisons, as well as the values they are directly compared against. Thus it can more easily get through "hard" comparisons, as discussed in Chapter 4. VUzzer re-implements the whole coverage-guided fuzzing loop rather than building on top of e.g. AFL or libFuzzer.

Steelix [117] is another coverage-guided fuzzer built on top of AFL, whose goal is to better get through hard comparisons. It relies on extra instrumentation to detect whether progress is made in a multi-byte equality comparison, i.e. one byte matches in a multi-byte comparison. When it notices this, it runs an adaptive mutation phase, exhaustively checking all byte values for the adjacent bytes to the matched byte, and continuing this process until no more adjacent bytes can be made to match. This is similar to the `cmp` fuzzer discussed in Section 4.4.2, but with more targeted mutations. pFuzzer [129] leverages unsatisfied byte-level comparisons in order to extract the byte that must be present at the last position of an input to get through the comparison, especially well-suited to fuzzing recursive descent parsers.

Angora [54] also improves on AFL using taint-tracking and a gradient descent methodology. It uses the LLVM instrumentation framework to track which input bytes flow into different path constraints, and in order to try and get through those constraints, restricts its mutations accordingly. Angora treats each predicate in a path constraint as a blackbox function $f(\mathbf{x})$ over the part of the input $\mathbf{x}$ that flows into the predicate. It slightly modifies the $\mathbf{x}$ and, with the results of this mutant, uses the finite difference method to approximate the gradient of $f(\mathbf{x})$ to guide its search. In addition to this, Angora also tries to infer the type of different byte sequences (i.e. are these four bytes used as an integer). And, instead of simply looking at increased branch coverage as AFL does, it adds a notion of *context*: the coverage point includes not only the branch, but also a hash of the call stack when that branch was taken.

Matryoshka [55] adopts some of Angora's taint-tracking and gradient-descent strategies, but targets the generation of inputs that satisfy highly-nested conditional statements. Given

a target condition, it first identifies the set of conditions that dominate it, as well as the data flow of input bytes to these conditions. Then it uses a variety of mutation masking procedures, and combining inputs that satisfy individual conditions, to create inputs that satisfy the highly-nested condition.

MOPT [126] alters the order in which different AFL mutation stages are applied, and whether they are applied at all. In particular, it allows for the scheduling of each different deterministic mutation stage, as well as the havoc and splicing (crossover) stage. It models each of these mutation stages as a particle moving in a probability space. Then it uses Particle Swarm Optimization to try and maximize the efficiency of mutation, i.e. keep the mutation stages which resulted in most inputs with new coverage.

RedQueen [32] uses a more lightweight approach to get through hard branches. It identifies correspondences between the input and the program state by simple equality; are there data strings in the program that are exact subsequences of the input. It instruments comparison operators to get this data for direct equality conditions. It can then patch the corresponding input data to get through the equality conditions. It uses a similar approach to get through checksums: first by identifying computations that look like checksums, and patching inputs with the checksum-computed values.

ProFuzzer [183] aims to recover some structural information from seed inputs in order to improve the performance of fuzzing. For each byte in the seed inputs, it generate mutants with each different possible value of the byte. Based on the changes in coverage caused by the mutation, it determines a category for the byte, e.g.: is it a constant, is it used in a loop count, is it an offset, a size. Then it groups together sequential bytes of the same category, and performs more relevant mutations based on the category.

Entropic [44] leverages the model of software testing as species discovery [43] to improve the heuristics, in particular fuzzProb in Line 5 of Algorithm 1, of libFuzzer. It introduces the notion of local Shannon's Entropy for a seed t — essentially a score of how likely mutants of t are to discover new branches. Then, fuzzProb is proportional to a Bayesian estimator of this entropy metric, over low-frequency branches. Due to consistent improvements in coverage achieved and reductions in time to expose bugs, this implementation of fuzzProb was incorporated into mainstream libFuzzer.

AFL++ [71] has been developed to integrate many of the innovations discussed above into one single fuzzer. It includes AFLFast's energy schedules, RedQueen's input-to-state replacement, MOPT's mutation scheduling, amongst others. It also provides an API to more easily customize the underlying fuzzer's input generation strategies. As of September 2020, it was the fuzzer with the highest overall score as per Google's FuzzBench project [175].

An alternative to a single mega-fuzzer is to combine multiple fuzzers in parallel. EnFuzz [55] found that an ensemble fuzzer that shared saved inputs between multiple different fuzzers generally achieved higher coverage and bug-finding ability compared to the sum of its parts.

2.4 Specialized Feedback-Directed Fuzzers

As will be discussed in Chapter 4, there has been interest in using the high-level coverage-guided fuzzing algorithm for different specialized use cases beyond general increased coverage of command-line utilities.

AFLGo [45] is a directed fuzzing tool, which aims to generate inputs hitting a set of targets, each of which is a line in a file. It uses call graphs obtained from whole-program static analysis to compute the distances of between basic blocks, and gives positive feedback to inputs that reduce this distance. Wüstholz and Christakis propose a directed fuzzing tool for smart contracts, Bran [180]. Instead of conducting a whole-program static analysis, it conducts online static analysis to determine the smallest no-target-ahead prefix of an input's path—assuming that input does not exercise the target. It uses information about the rarity of these prefixes to influence numChildren, and emphasize the mutation of inputs which have a rare no-target-ahead path.

Ijon [31] allows security analysts to insert feedback statements into the program under test, much like the approach described in Chapter 4. The feedback statements are at a different level, however. They allow the developer to increment/decrement/maximize feedback keys, as in Chapter 4, but also to enable and disable coverage feedback in different parts of the program, and also provide an explicit notion of state feedback.

SlowFuzz [157], built on top of libFuzzer, aims to find inputs showing algorithmic complexity vulnerabilities. The main idea is to create inputs with longer path lengths through the program by saving inputs with longer path lengths, and adapting the mutation strategy to locations that result in longer paths. Chapter 3 contrasts this approach with PerfFuzz.

Nezha is a differential fuzzing tool [156] which fuzzes multiple programs at the same time on one input and tries to maximize the difference in their behavior on that input. In particular, Nezha introduces the concept of δ-diversity and uses this to force the generation of inputs showing different behaviors in the program under test. It prioritizes the inputs which have radically different paths through the programs compared to their parents, as well as the outputs. This allows for faster discovery of inputs which illustrate semantic differences between the programs—most likely bugs. DifFuzz [140] also looks at running multiple versions of a program on a same input, but in order to find side-channel vulnerabilities. Essentially, the input being mutated consists of a public input as well as two distinct secrets. Then DifFuzz prioritizes inputs which increase the difference in execution cost between the runs of the program under the two secrets.

Memlock [178] aims to find memory consumption crashes, vulnerabilities, and memory leaks. It uses an approach similar to that described in Chapter 3 to accomplish this, but it instruments memory allocation and deallocation statements in order to build a *perfmap* mapping allocations locations to the amount of memory allocated there. MemFuzz [61] saves inputs that read/write new values to input-dependent memory addresses; this allows it to find new bugs compared to AFL.

Lauefer et al. [109] use validity feedback in order to fuzz circuits that have constrained interfaces. This validity feedback is similar to that described in Chapter 6. Harvey [179]

is a greybox fuzzer optimized for smart contracts. It adds feedback that guides it through conditional statements, by translating conditional statements to linear expressions and trying to minimize those expressions. It also detects branches that require more than one transaction, and uses this bound the transaction sequences it explores. When exploring sequences of length more than 1, it only prioritizes increased coverage of the last transaction in the sequence.

Bugariu et al. build a fuzzer to test implementations of abstract domains [49] by building specialized oracles for static analysis bugs. Each of these oracles is encoded as a test-driver, which also encodes the generation of inputs in a given abstract interpretation domain. A coverage-guided backend controls the generation of these inputs. In some ways, this can be seen as a specialized guided generator-based fuzzer, although the inputs are captured as a sequence of operations in the abstract interpretation domain.

2.5 Structure-Aware and Generator-Based Fuzzers

There is a rich history of randomized testing methods that leverage specifications of input structure in order to generate inputs. At some level, most of these fall under the abstraction of generator-based fuzzing, but there are many different branches of generator-based fuzzing, and ways to utilize structural information in the production of inputs.

Generator-based fuzzing, also called property-based testing, which was discussed in the introduction, was first popularized by QuickCheck [58]. It allows users to quickly check a property of the form $P(x) \Rightarrow Q(x)$ over a domain of inputs $\mathcal{X}$. In particular, given a user-defined test driver implementing $P(x) \Rightarrow Q(x)$ with a generator of inputs $x \in \mathcal{X}$, QuickCheck will execute $P(x) \Rightarrow Q(x)$ on many inputs x. This gives an approximate check of $\forall x \in \mathcal{X}, P(x) \Rightarrow Q(x)$. The method has grown in popularity thanks to implementations in many different languages [14, 13, 18, 19, 151], including prominent languages such as Python [15], JavaScript [16], and Java [98].

UDITA [78] allows developers to write random input generators in a QuickCheck-like language. UDITA then performs *bounded-exhaustive* enumeration of the paths through the generators, along with several optimizations. Targeted property-testing [123, 124] guides input generators used in property testing towards a user-specified fitness value using techniques such as hill climbing and simulated annealing. GödelTest [67] attempts to satisfy user-specified properties on inputs. It performs a meta-heuristic search for stochastic models that are used to sample random inputs from a generator, similar to the guides discussed in Part III.

One problem with traditional property-based testing is when few of the generator-generated inputs satisfy $P(x)$. One way to solve this problem is to do whitebox analysis [81, 47, 78] of the generator and/or the implementations of $P(x)$ and $Q(x)$. A constraint solver can be used to generate inputs $x \in \mathcal{X}$ that are guaranteed to satisfy $P(x)$, which also exercise different code paths within the implementation of $Q(x)$ [164]. Another approach is to collect code coverage during test execution [112]. This information can be used in an evolutionary algorithm to generate inputs that are likely satisfy $P(x)$, while optimizing to increase code coverage through

$Q(x)$. Chapter 6 presents a similar code-coverage-based approach; a blackbox approach is discussed in Chapter 7.

Grammar-based fuzzing [130, 170, 62, 40] techniques rely on context-free grammar specifications to generate structured inputs. Godefroid et al. [80] use grammars to build symbolic constraints at the level of grammar elements, resulting in much higher coverage of the programs under test than regular *symbolic execution* (discussed in the next subsection). LangFuzz [97] generates random programs using a grammar and by recombining code fragments from a codebase, a mixture of generational and mutational fuzzing. The PEACH fuzzer [17] allows for more effective fuzzing of programs expecting structured inputs by randomly sampling inputs according to complex models of common file formats.

CSmith [181] generates random C programs for differential testing of C compilers. The generator of these programs is highly optimized to prevent the generation of C programs with undefined behavior, though it requires heavy duty dynamic checks to prevent all unsafe programs from being generated. YARPGen [122], yet another random generator of C and C++ programs, does not rely on these dynamic checks. It carefully separates variables into different categories (read-only, write-only, read-write). Then it interleaves a type-checking static analysis with the generation process. This analysis is bottom-up, identifying sub-expressions with undefined behavior and mutating them to remove the undefined behavior.

Sulley [28] and BooFuzz [154] allow users to effectively fuzz protocols, based on specifications of those protocols. These specifications must be user-provided. LZFuzz [48] tries to automatically learn some of the structure of inputs for unknown protocols. It uses an adapted compression algorithm to identify blocks in the inputs, and leverages this learned structure for fuzzing.

Several approaches have also looked at leveraging input structures to improve the performance of coverage-guided fuzzing. For instance, libprotobuf-mutator [168] extracts high-level mutation operations from protocol buffer specifications of inputs. AFLSmart [158] uses PEACH specifications of inputs to get higher-level mutation operators, as well as validity feedback to guide CGF to generating inputs that fully parse.

Superion [176] uses grammars to enhance the mutation strategies of AFL—including a LangFuzz-like strategy—and add a structured trimming (i.e. minimization) phase that works on the AST level. Nautilus [30] also enhances coverage-guided fuzzing with grammars. It leverages the grammar to (1) generate inputs from scratch which cover diverse aspects of the grammar; (2) minimize inputs; (3) mutate inputs at the AST level. It introduces 4 different AST-based mutation strategies as well as retaining some AFL mutations on the leaves.

Another direction is to learn some version of the input structure, rather than assume the user provides a specification of input structure. LZFuzz [48], which we already discussed, does this for network protocols. GRIMOIRE [41] learns something close to a context-free grammar while fuzzing inputs. Essentially it learns a single-level hierarchy grammar, by figuring out which parts of the input can be modified while still maintaining the newly-discovered branch which made the input interesting to save. DIFUZE [63] infers device driver interfaces from a running kernel to bootstrap subsequent structured fuzzing. Learn&Fuzz [83] uses a sequence-to-sequence model to learn PDF objects, then leverages this model to generate new

PDF documents with different objects. Interestingly, this model-based approach resulted in lower overall coverage, likely due to AFL's coverage of error states.

There are also some recent works that focus more on the grammar learning rather than fuzzing. GLADE [37] uses an iterative approach and repeated calls to an oracle to learn a context-free grammar for a set of inputs. The first phase of this learning generalizes each input as a regular expression; a second phase merges learned substructures of the regular expression. The learned grammars are not meant to be human-readable, but can be used to sample inputs for fuzzing.

Others approaches use whitebox or greybox information about the program under test to learn grammars. Lin et al.'s work examines execution traces in order to reconstruct program input grammars [120, 119]. AUTOGRAM [99] tracks input flows into variables, and uses this dataflow information to learn a well-labeled grammar. Mimid [87] goes a step further, tracking the control-flow nodes in which input characters are accessed. It directly maps this control-flow structure to the grammar structure, and takes advantage of function names in order to sensibly label nonterminals.

2.6 Other Approaches to Automated Testing

The core idea behind symbolic execution [59, 104] is to generate inputs by reasoning about the *path constraints* of the program under test. A *path constraint* is the conjunction of logical constraints that must be satisfied for an input to follow a certain path through a program. Advances in SMT solvers have made these methods viable in practice [65, 36, 66].

Traditionally the term symbolic execution is used to refer to methods that collect the whole set of path constraints at the same time. KLEE is the classic example of such a symbolic execution engine [51]. KLEE works by analyzing the execution of inputs at the LLVM bytecode level. Each time it hits a branch statement, it adds the two alternative paths to the state space of the program. To explore the state space efficiently, it uses either random sampling of paths, or tries to explore paths that are more likely to result in increased coverage. One of the big problems with this full exploration is that the space of path constraints grows exponentially in the number of branches; this is called the path explosion problem.

Concolic execution or dynamic symbolic execution [165, 81] works slightly differently. It starts with a concrete input and runs it through the program, collecting the constraint of the path it follows. Then it flips the last branch in that path which has not been fully explored, and generates an input that solves that path constraint. If all branches are flipped eventually, this will ensure full coverage of the program under test. Nonetheless, it still suffers from the aforementioned path explosion problem. MultiSE [166] uses an alternative representation of the search space—value summaries—that enables it to incrementally merge the state space, alleviating some of these problems.

The term whitebox fuzzing [82] generally refers to input-generation techniques that utilize constraint solving to generate inputs, but explore the path constraint space in a more random manner. SAGE [82] is the most well-known whitebox fuzzer. As in concolic execution, it

starts with a concrete path through the program. However, instead of just flipping the last constraint in the path, it tries to flip each constraint individually, generating a new child input for each constraint. Ideally, this process would be repeated for the child inputs, but this too explodes the search space. So SAGE uses the heuristic of prioritizing the expansion of child inputs which result in the largest increase in basic block coverage.

Mayhem's [52] goal is to find exploitable bugs in program binaries. It switches between a concrete executor client and a symbolic execution server to generate inputs exposing potentially exploitable bugs. The concrete execution side performs dynamic taint tracking, and when it finds a branch condition or jump target which is tainted, it switches control to the symbolic execution side. The idea is that if a jump target is tainted by the input, it could potentially be used maliciously by an attacker to cause a jump to unverified code. The symbolic execution engine then tries to determine whether the branches sent to it from the concrete executor are feasible. In addition to checking path constraints, Mayhem also keeps track of an exploitability formula when hitting tainted jump conditions. If this formula is satisfiable, then the input satisfying it will be an exploit, a distinctive feature of Mayhem.

There has been a recent resurgence of interest in building more scalable symbolic execution engines from the security community. SymCC [159] is a drop-in replacement to the `clang` or `clang++` compiler, which builds concolic execution into the binary. The concept is similar to that in CUTE [165]; during compilation, instructions are added to the binary which, when executed, build up the symbolic constraints. The important point is that this reasoning about building up symbolic path expressions is only conducted at compile time, as opposed to interpreter-based symbolic executors [56, 169]. SymQEMU [160] achieves similar benefits, but without requiring access to the source code of the program under test. They use the Tiny Code Generator (TCG) component of QEMU [39] to do this. In regular QEMU, this component translates the binary to TCG operations before compiling them to the target machine code. During this process, SymQEMU also adds TCG operations that will, when executed, build up the symbolic constraints.

The key advantage of the constraint solver based techniques is that they can easily get through "hard" constraints, a key problem for coverage-guided mutational fuzzers. Several works have looked at integrating symbolic execution and coverage-guided mutational fuzzing to get the best of both worlds. Driller [172] runs coverage-guided fuzzing until it gets "stuck" in certain component, i.e. has not discovered new branches in a certain function for a while. At this point, it invokes concolic execution on the saved inputs for that component. In only tries to flip constraints that have not been previously exercised by a saved input. Munch [142] also orchestrates symbolic execution (KLEE) and AFL. It can run in two hybrid modes. If seed inputs are available, it runs the FS hybrid mode: run AFL for some time, then use KLEE to generate inputs that reach function not reached by AFL. It saves some of the symbolic execution effort by excluding paths to functions explored by AFL already. Otherwise, it runs the SF hybrid mode: start by generating seed inputs via symbolic execution, then run fuzzing.

Before describing QSym [184], Yun et al. evaluate the actual performance costs of hybrid (symbolic + coverage-guided) fuzzing. They find that the path explosion problem is not the

only bottleneck to performance. In particular, (1) they help reduce the cost of emulation by writing a concolic executor at the x86 instruction level rather than the LLVM IR level; (2) they get rid of some snapshotting since the goal is to use concolic execution to get through a given hard branch; and (3) they reduce the cost of constraint solving by only trying to flip the target branch, rather than trying to keep an exact path prefix. This resulted in the discovery of new bugs in software that had been heavily fuzzed by coverage-guided fuzzing.

Another approach to smarter fuzzing is to find locations in seed inputs related to likely crash locations in the program and focus mutation there. BuzzFuzz [77] starts from a set of seed inputs and runs them through the program. It conducts dynamic taint tracking to figure out which bytes of the input flow into dangerous locations—e.g. the input of a `malloc` call. Then it creates mutant inputs by setting those bytes to extremal values. Similarly, TaintScope [177] tracks which bytes flow to security-sensitive operations and focuses mutations on these bytes. In addition, it automatically identifies checksums in the program under test, and bypasses these during fuzzing. If an interesting input is found which bypasses these checksums, it uses dynamic symbolic execution to repair the checksum.

Dowser [92] focuses on finding inputs which cause buffer overflows. It identifies locations in the program where buffer overflows may occur, in particular, loops containing pointer dereferences. It then conducts taint-tracking to determine which bytes flow to these target locations. Then it conducts partial dynamic symbolic execution, treating only those bytes that flow to the target locations as symbolic.

While coverage-guided fuzz testing emerged from the security community, the idea of using genetic algorithms for testing has been long explored in the software engineering community [137, 107]. The field of search-based software testing [131, 91, 90, 182] uses optimization techniques such as hill climbing and genetic algorithms to generate inputs that optimize some observable fitness function. These techniques work well when the fitness curve is smooth with respect to changes in the input.

For instance, Sapienz [128] automatically creates test suites for Android applications. It uses proper multi-objective search (i.e. Pareto optimal [93]) to maximize code coverage and number of crashes found while reducing the length of interaction sequences in the test suite. Since the whole test suite is being evolved at a time, the maximization of code coverage is a reasonable approach; coverage-guided fuzzing tools like AFL operate on the single test-case level, and thus must rely on the novelty search approach of saving inputs that find new coverage. Sapienz uses several mutation operators specialized to the Android domain in order to optimize the search.

Finally, another direction in automated testing is to automatically create test cases rather than inputs to a test driver. Randoop [147] and EvoSuite [74] automatically produce JUnit tests for a particular class by incrementally trying and combining sequences of method invocations on the component classes. During the generation of sequence of calls, both Randoop and EvoSuite take some form of feedback into account. For instance, Randoop avoids extending call sequences that led to exceptions. EvoSuite uses a genetic algorithm to evolve a test suite using code coverage as a fitness function.

Bibliography

[1] CVE-2011-3414. Available from MITRE, 2011.

[2] CVE-2011-4858. Available from MITRE, 2011.

[3] CWE-400: Uncontrolled Resource Consumption. Available from MITRE, 2011. Accessed Jan 2018.

[4] CVE-2014-5265. Available from MITRE, 2014.

[5] CVE-2017-9804. Available from MITRE, 2017.

[6] `wf` - Simple word frequency counter, 2017. Accessed Jan 2018.

[7] Apache Ant. `https://ant.apache.org`, 2018. Accessed August 24, 2018.

[8] Apache Byte Code Engineering Library. `https://commons.apache.org/proper/commons-bcel`, 2018. Accessed August 24, 2018.

[9] Apache Maven. `https://maven.apache.org`, 2018. Accessed August 24, 2018.

[10] Google Closure. `https://developers.google.com/closure/compiler`, 2018. Accessed August 24, 2018.

[11] Mozilla Rhino. `https://github.com/mozilla/rhino`, 2018. Accessed August 24, 2018.

[12] React.JS. `https://reactjs.org`, 2018. Accessed August 24, 2018.

[13] Eris: Porting of QuickCheck to PHP. `https://github.com/giorgiosironi/eris`, 2019. Accessed January 28, 2019.

[14] FsCheck: Random testing for .NET. `https://hypothesis.works/`, 2019. Accessed January 28, 2019.

[15] Hypothesis for Python. `https://hypothesis.works/`, 2019. Accessed January 28, 2019.

[16] JSVerify: Property-based testing for JavaScript. `https://github.com/jsverify/jsverify`, 2019. Accessed January 28, 2019.

[17] PeachFuzzer. https://www.peach.tech/, 2019. Accessed January 28, 2019.

[18] ScalaCheck: Property-based testing for Scala. https://www.scalacheck.org/, 2019. Accessed January 28, 2019.

[19] test.check: QuickCheck for Clojure. https://github.com/clojure/test.check, 2019. Accessed January 28, 2019.

[20] CVE-2020-7212. Available from MITRE, 2020.

[21] Address Sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, 2021. Accessed Apr 30, 2021.

[22] Leak Sanitizer. https://clang.llvm.org/docs/LeakSanitizer.html, 2021. Accessed Apr 30, 2021.

[23] Memory Sanitizer. https://clang.llvm.org/docs/MemorySanitizer.html, 2021. Accessed Apr 30, 2021.

[24] Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2021. Accessed Apr 30, 2021.

[25] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[26] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016.

[27] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*, 2018.

[28] P. Amini and A. Portnoy. Sulley. https://github.com/OpenRCE/sulley, 2012. Accessed August 22nd, 2017.

[29] B. Archer and Drakkey. Radamsa: a general-purpose fuzzer. https://gitlab.com/akihe/radamsa, 2019. Accessed August 21, 2019.

[30] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[31] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[32] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security*, NDSS '19, 2019.

[33] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[34] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[35] T. Ball, P. Mataga, and M. Sagiv. Edge Profiling Versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 134–148, New York, NY, USA, 1998. ACM.

[36] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 2017.

[38] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[39] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[40] M. Beyene and J. H. Andrews. Generating String Test Data for Code Coverage. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 270–279, 2012.

[41] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, Aug. 2019. USENIX Association.

[42] M. Böhme. AFLFast.new. `https://groups.google.com/d/msg/afl-users/1PmKJC-EKZ0/lbzRb8AuAAAJ`, 2016. Accessed August 23rd, 2017.

[43] M. Böhme. STADS: Software Testing as Species Discovery. *ACM Trans. Softw. Eng. Methodol.*, 27(2), June 2018.

[44] M. Böhme, V. J. M. Manès, and S. K. Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 678–689, New York, NY, USA, 2020. Association for Computing Machinery.

[45] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[46] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[47] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[48] S. Bratus, A. Hansen, and A. Shubina. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. Technical report, Department of Computer Science, Darmouth College, 2008.

[49] A. Bugariu, V. Wüstholz, M. Christakis, and P. Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 768–778, New York, NY, USA, 2018. Association for Computing Machinery.

[50] M. Bynens. In search of the perfect URL validation regex. `https://mathiasbynens.be/demo/url-regex`, 2014.

[51] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[52] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[53] S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, 2015.

[54] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[55] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[56] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011.

[57] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, 2014.

[58] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, ICFP, 2000.

[59] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[60] E. Coppa, C. Demetrescu, and I. Finocchi. Input-Sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[61] N. Coppik, O. Schwahn, and N. Suri. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.

[62] D. Coppit and J. Lian. Yagg: An Easy-to-use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 356–359, New York, NY, USA, 2005. ACM.

[63] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2123–2138, New York, NY, USA, 2017. ACM.

[64] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[65] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[66] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54:69–77, Sept. 2011.

[67] R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013.

[68] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 420–435, New York, NY, USA, 2018. ACM.

[69] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.*, 52(6):422–436, June 2017.

[70] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM.

[71] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[72] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, page 6, USA, 2000. USENIX Association.

[73] O. S. Foundation. OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`, 1999. Accessed Apr 30, 2021.

[74] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

[75] G. Fraser and A. Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[76] S. Frizell. Report: Devastating Heartbleed Flaw Was Used in Hospital Hack. *Time*, 2014. Accessed Apr 30, 2021.

[77] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, 2009.

[78] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 225–234, 2010.

[79] P. Godefroid. Fuzzing: Hack, Art, and Science. *Commun. ACM*, 63(2):70–76, Jan. 2020.

[80] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[81] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[82] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Symposium on Network and Distributed System Security*, NDSS '08, 2008.

[83] P. Godefroid, H. Peleg, and R. Singh. Learn & Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

[84] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring Empirical Computational Complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 395–404, New York, NY, USA, 2007. ACM.

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

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

[87] R. Gopinath, B. Mathis, and A. Zeller. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 2019 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1–12, New York, NY, USA, 2020. Association for Computing Machinery.

[88] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[89] J. Graham-Cumming. Incident report on memory leak caused by Cloudflare parser bug. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`, 2017. Accessed Apr 30, 2021.

[90] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[91] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE, 2009.

[92] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, 2013.

[93] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[94] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[95] S. Hocevar. zzuf. `http://caca.zoy.org/wiki/zzuf/`, 2007. Accessed August 22nd, 2017.

[96] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. Eclemma-jacoco java code coverage library, 2011.

[97] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[98] P. Holser. junit-quickcheck: Property-based testing, JUnit-style. `https://pholser.github.io/junit-quickcheck`, 2014. Accessed January 11, 2019.

[99] M. Höschele and A. Zeller. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.

[100] A. D. Householder and J. M. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. Technical report, Carnegie Mellon University Software Engineering Institute, 2012.

[101] S. Inc. Heartbleed. `https://heartbleed.com/`, 2014. Accessed Apr 30, 2021.

[102] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, Aug. 2020.

[103] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[104] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[105] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, Dec. 2014.

[106] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.

[107] B. Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

[108] H. Krasner. The Cost of Poor Quality Software in the US: A 2018 Report. Technical report, Consortium for Information & Software Quality, 2018.

[109] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pages 28:1–28:8, New York, NY, USA, 2018. ACM.

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

[111] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L.-y. Xia. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 114–129, New York, NY, USA, 2017. ACM.

[112] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[113] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[114] C. Lemieux, R. Padhye, K. Sen, and D. Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[115] C. Lemieux and K. Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, 2018.

[116] M. Leung and C. Commisso. Canadians filing taxes late due to 'Heartbleed' bug won't face penalties: CRA. `https://www.ctvnews.ca/canada/canadians-filing-taxes-late-due-to-heartbleed-bug-won-t-face-penalties-cra-1.1767727`, 2014. Retrieved October 23, 2020.

[117] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[118] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated Graph Sequence Neural Networks. *CoRR*, abs/1511.05493, 2015.

[119] Z. Lin and X. Zhang. Deriving Input Syntactic Structure from Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 83–93, New York, NY, USA, 2008. Association for Computing Machinery.

[120] Z. Lin, X. Zhang, and D. Xu. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering*, 36(5):688–703, 2010.

[121] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[122] V. Livinskii, D. Babokin, and J. Regehr. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[123] A. Löscher and K. Sagonas. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 46–56, New York, NY, USA, 2017. ACM.

[124] A. Loscher and K. Sagonas. Automating Targeted Property-Based Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, volume 00, pages 70–80, Apr 2018.

[125] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 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, pages 190–200, New York, NY, USA, 2005. ACM.

[126] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, Aug. 2019. USENIX Association.

[127] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, Science, and Engineering. *CoRR*, abs/1812.00140, 2018.

[128] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[129] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 548–560, New York, NY, USA, 2019. Association for Computing Machinery.

[130] P. M. Maurer. Generating test data with enhanced context-free grammars. *Ieee Software*, 7(4):50–55, 1990.

[131] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.

[132] Microsoft. Gated Graph Neural Network Samples. https://github.com/Microsoft/gated-graph-neural-network-samples, 2017. Accessed October 17th, 2018.

[133] B. Miller, M. Zhang, and E. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[134] B. P. Miller, G. Cooksey, and F. Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, page 46–54, New York, NY, USA, 2006. Association for Computing Machinery.

[135] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[136] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, 1995.

[137] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223, 1976.

[138] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[139] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[140] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu. DifFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 176–187, Piscataway, NJ, USA, 2019. IEEE Press.

[141] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[142] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. 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, pages 1475–1482, New York, NY, USA, 2018. ACM.

[143] I. Ogrodnki. 900 SINs stolen due to Heartbleed bug: Canada Revenue Agency. `https://globalnews.ca/news/1269168/900-sin-numbers-stolen-due-to-heartbleed-bug-canada-revenue-agency/`, 2014. Retrieved October 23, 2020.

[144] OpenSSL. OpenSSL Security Advisory [07 Apr 2014]. `https://www.openssl.org/news/secadv/20140407.txt`, 2014. Retrieved October 23, 2020.

[145] OpenSSL. OpenSSL Security Advisory [26 Sep 2016]. `https://www.openssl.org/news/secadv/20160926.txt`, 2016. Retrieved October 23, 2020.

[146] OW2 Consortium. ObjectWeb ASM. `https://asm.ow2.io`, 2018. Accessed August 21, 2018.

[147] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, 2007.

[148] R. Padhye, C. Lemieux, and K. Sen. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, 2019.

[149] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, 2019.

[150] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[151] M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-based Testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. ACM.

[152] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[153] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[154] J. Pereyda. BooFuzz. `[https://github.com/jtpereyda/boofuzz`, 2016. Accessed May 4th, 2020.

[155] N. Perlroth. Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant. *The New York Times*, 2014. Accessed Apr 30, 2021.

[156] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632. IEEE, 2017.

[157] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[158] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart Greybox Fuzzing. *CoRR*, abs/1811.09447, 2018.

[159] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, Aug. 2020.

[160] S. Poeplau and A. Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium*, NDSS '21, 2021.

[161] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, NDSS '17, 2017.

[162] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association.

[163] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1410–1421, New York, NY, USA, 2020. Association for Computing Machinery.

[164] T. Ringer, D. Grossman, D. Schwartz-Narbonne, and S. Tasiran. A Solver-aided Language for Test Input Generation. *Proc. ACM Program. Lang.*, 1(OOPSLA):91:1–91:24, Oct. 2017.

[165] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005.

[166] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[167] K. Serebryany. libFuzzer. `http://llvm.org/docs/LibFuzzer.html`, 2016. Accessed August 25th, 2017.

[168] K. Serebryany, V. Buka, and M. Morehouse. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator, 2017.

[169] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[170] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 1–13, New York, NY, USA, 1999. ACM.

[171] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 561–578, New York, NY, USA, 2014. ACM.

[172] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, 2016.

[173] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, chapter 5, pages 120–124. MIT Press, 2018.

[174] R. Swiecki. honggfuzz. `http://honggfuzz.com/`, 2016. Accessed August 7, 2018.

[175] L. Szekeres. FuzzBench: 2020-09-07 Sample Report . `https://web.archive.org/web/20210503193514/https://www.fuzzbench.com/reports/sample/index.html`. Accessed May 3rd, 2021.

[176] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-Aware Greybox Fuzzing. In *41st International Conference on Software Engineering*, ICSE '19, 2019.

[177] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[178] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 765–777, New York, NY, USA, 2020. Association for Computing Machinery.

[179] V. Wüstholz and M. Christakis. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1398–1409, New York, NY, USA, 2020. Association for Computing Machinery.

[180] V. Wüstholz and M. Christakis. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 789–800, New York, NY, USA, 2020. Association for Computing Machinery.

[181] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[182] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[183] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786, 2019.

[184] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association.

[185] M. Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl`, 2014. Accessed August 18th, 2017.

[186] M. Zalewski. Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78). `https://web.archive.org/web/20210428205228/http://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html`, 2014. Accessed May 2, 2020.

[187] M. Zalewski. Pulling JPEGs out of thin air. `https://web.archive.org/web/20210123014427/https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`, 2014. Accessed May 2, 2020.

[188] M. Zalewski. FidgetyAFL. `https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ`, 2016. Accessed August 23rd, 2017.

[189] M. Zalewski. American Fuzzy Lop Technical Details. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017. Accessed August 18th, 2017.

[190] D. Zaparanuks and M. Hauswirth. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[191] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.