VUzzer: Application-aware Evolutionary Fuzzing

Sanjay Rawat*†, Vivek Jain‡, Ashish Kumar‡, Lucian Cojocar*†, Cristiano Giuffrida*† and Herbert Bos*†
*Computer Science Institute, Vrije Universiteit Amsterdam
<s.rawat,lucian.cojocar>@vu.nl; <giuffrida,herbertb>@cs.vu.nl
†Amsterdam Department of Informatics
‡ International Institute of Information Technology, Hyderabad
<vivek.jain,ashish.kumar>@research.iiit.ac.in

***Abstract*—Fuzzing is an effective software testing technique to find bugs. Given the size and complexity of real-world applications, modern fuzzers tend to be either scalable, but not effective in exploring bugs that lie deeper in the execution, or capable of penetrating deeper in the application, but not scalable.**

In this paper, we present an *application-aware* evolutionary fuzzing strategy that does not require any prior knowledge of the application or input format. In order to maximize coverage and explore deeper paths, we leverage control- and data-flow features based on static and dynamic analysis to infer fundamental properties of the application. This enables much faster generation of *interesting* inputs compared to an *application-agnostic* approach. We implement our fuzzing strategy in VUzzer and evaluate it on three different datasets: DARPA Grand Challenge binaries (CGC), a set of real-world applications (binary input parsers), and the recently released LAVA dataset. On all of these datasets, VUzzer yields significantly better results than state-of-the-art fuzzers, by quickly finding several existing and new bugs.

I. Introduction

Fuzzing is a testing technique to *catch bugs early*, before they turn into vulnerabilities. However, existing fuzzers have been effective mainly in discovering superficial bugs, close to the surface of software (*low-hanging bugs*) [13], [17], while struggling with more complex ones. Modern programs have a complex input format and the execution heavily depends on input values conforming to the format. Typically, a fuzzer *blindly* mutates values to generate new inputs. In this (pessimistic) scenario, most of the resulting inputs do not conform to the input format and are rejected in the early stages of the execution. This makes a *traditional random fuzzer* often ineffective in finding bugs that hide deep in the execution.

State-of-the-art fuzzers such as AFL [52] employ *evolutionary algorithms* to operate valid input generation. Such algorithms employ a simple feedback loop to assess how good an input is. In detail, AFL retains *any* input that discovers a new path and mutates that input further to check if doing so leads to new basic blocks. While simple, this strategy cannot effectively select the most promising inputs to mutate from the discovered paths. In addition, mutating an input involves answering two questions: *where* to mutate (which offset in the input) and *what value* to use for the mutation? The problem is that AFL is completely *application-agnostic* and employs a *blind* mutation strategy. It simply relies on generating a huge amount of mutated inputs in the hope of discovering a new basic block. Unfortunately, this approach yields a slow fuzzing strategy, which can only discover deep execution paths by sheer luck. Fortunately, we can increase the efficiency of AFL-like fuzzers manifold by accounting for information that answers the questions above.

In this direction, the use of symbolic and concolic execution has shown promising results [33], [47]. Driller [47] uses concolic execution to enable AFL to explore new paths when it gets stuck on superficial ones. However, fuzzers like AFL are designed to target arbitrarily large programs and, in spite of several advancements, the application of symbolic/concolic techniques to such programs remains a challenge [10]. For example, Driller was benchmarked with 126 DARPA CGC binaries [15] and when AFL got stuck on 41 such binaries, its concolic engine could only generate new meaningful inputs from 13 of such binaries. The results reported in the LAVA paper [17] evidence similar problems with symbolic execution approaches. In another recent study [50], the authors reported that symbolic execution-based input generation (using KLEE) is not very effective at exploring meaningful and deeper paths. In essence, while combining fuzzing with symbolic execution is an interesting research area, this approach also significantly weakens one of fuzzing's original key strengths: *scalability*.

In this paper, we present VUzzer, an *application-aware* evolutionary fuzzer which is both scalable and fast to discover bugs deep in the execution. In contrast to approaches that optimize the input generation process to produce inputs at maximum rates, our work explores a new point in the design space, where we do more work at the front-end and produce fewer but better inputs. The key intuition is that we can enhance the efficiency of general-purpose fuzzers with a "*smart*" mutation feedback loop based on *control- and data-flow* application features without having to resort to less scalable symbolic execution. We show that we can extract such features by lightweight static and dynamic analysis of the application during fuzzing runs. Our control-flow features allow VUzzer to *prioritize* deep (and thus interesting) paths and *deprioritize* frequent (and thus uninteresting) paths when mutating inputs. Our data-flow features allow VUzzer to accurately determine *where* and *how* to mutate such inputs.

Thanks to its application-aware mutation strategy, VUzzer is much more efficient than existing fuzzers. We evalu-

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23404

ated the performance of VUzzer on three different datasets: a) the DARPA CGC binaries [15], a collection of artificially created interactive programs designed to assess bug-finding techniques; b) a set of Linux programs with varying degrees of complexity (`djpeg`, `mpg321`, `pdf2svg`, `gif2png`, `tcpdump`, `tcptrace`) and c) the recently released binaries from the LAVA team [17], a number of Linux utilities with several injected bugs. In our experiments on the different datasets, we outperformed AFL by generating orders of magnitude fewer inputs, while finding more crashes. For example, in *mpg321*[1], we found 300 unique crashes by executing 23K inputs, compared to 883K inputs to find 19 unique crashes with AFL.

Contributions: We make the following contributions:

1) We show that modern fuzzers can be "*smarter*" without resorting to symbolic execution (which is hard to scale). Our application-aware mutation strategy improves the input generation process of state-of-the-art fuzzers such as AFL by orders of magnitude.
2) We present several application features to support *meaningful* mutation of inputs.
3) We evaluate VUzzer, a fully functional fuzzer that implements our approach, on three different datasets and show that it is highly effective.
4) To foster further research in the area and in support of open science, we are open sourcing our VUzzer prototype, available at https://www.vusec.net/projects/fuzzing.

II. Background

In this section, we cover the background required for our discussion of VUzzer in subsequent sections.

A. A Perspective on Fuzzing

Fuzzing is a software testing technique aimed at finding bugs in an application [35], [48]. The crux of a fuzzer is its ability to generate bug triggering inputs. From an input generation perspective, fuzzers can be *mutation-* or *generation-*based. Mutation-based fuzzers start with a set of known inputs for a given application and mutate these inputs to generate new inputs. In contrast, generation-based fuzzers first learn/acquire the input format and generate new inputs based on this format. VUzzer is a mutation-based fuzzer.

With respect to input mutation, fuzzers can be classified as *whitebox, blackbox* and *greybox*. A whitebox fuzzer [21], [22], [26] assumes access to the application's source code—allowing it to perform advanced program analysis to better understand the impact of the input on the execution. A blackbox fuzzer [1], [39] has no access to the application's internals. A greybox fuzzer aims at a middle ground, employing lightweight program analysis (mainly monitoring) based on access to the application's binary code. VUzzer is a greybox fuzzer.

Another factor that influences input generation is the application exploration strategy. A fuzzer is *directed* if it generates inputs to traverse a particular set of execution paths [20]–[22], [26], [38]. A *coverage-based* fuzzer, on the other hand, aims at generating inputs to traverse different paths of the application in the hope of triggering bugs on some of these path [14], [28], [44], [47], [52]. VUzzer is a coverage-based fuzzer.

By definition, a coverage-based fuzzer aims at maximizing the code coverage to trigger paths that may contain bugs. To maximize code coverage, the fuzzer tries to generate inputs such that each input (ideally) executes a different path. Therefore, it is of paramount importance for a fuzzer to account for the gain obtained for each generated input, a property that we term *input gain* (IG). IG is defined as the ability of an input to discover a new path either by executing new basic blocks or increase the frequency of previously executed basic blocks (e.g., loop execution).

Obviously, a coverage-based fuzzer is effective if it frequently generates inputs with non-zero IG. It is not hard to notice that the ability to generate inputs with non-zero IG requires addressing our two questions in Section I (*where* and *what* to mutate). Unfortunately, most existing fuzzers, especially mutation-based ones make little effort to achieve this goal. For example, let us consider the code snippet in Listing. 1.

```
1 ...
2 read(fd, buf, size);
3 if (buf[5] == 0xD8 && buf[4] == 0xFF) // notice the order of CMPs
4     ... some useful code ...
5 else
6     EXIT_ERROR("Invalid file\n");
```

Listing 1. Simple multibyte IF condition.

In this code, buf contains tainted data from the input. On this simple code, AFL runs for hours without making any progress to go beyond if condition. The reason for this rather pessimistic behavior is twofold: (i) AFL has to guess the `FFD8` byte sequence exactly right; (ii) AFL has to find the right offsets (4 & 5) to mutate. As AFL is a coverage-based fuzzer, for an input which fails the if condition and thus results in a new path (the $else$ branch), AFL may focus on exploring this new path even if the path leads to an error state. In such case, AFL gets stuck in the $else$ branch. Symbolic execution-based solutions such as Driller [47] may help AFL by providing an input with the *right byte* at the *right offset*. However, this is not a definitive solution, because, with this new input, AFL again starts mutating at random. In the process, it may try mutating these offsets again, wasting processing power and time.

```
1 ...
2 read(fd,buf,size);
3 ...
4 if (...) {
5     if(...) // nested IF
6         ...
7 } else {
8     ...
9 }
```

Listing 2. Nested-level conditions and *deeper* paths

Now consider another simple (pseudo) code snippet, in Listing. 2. At line 5, there is another multibyte if condition on the input bytes, which is nested in the outer if. As AFL will likely fail to satisfy the branch constraint, it will generate inputs that traverse the $else$ branch. As there is code to explore in the $else$ branch, AFL will not be able to prioritize efforts to target the if branch. It is hard to impart such knowledge to AFL even via symbolic execution. As a result, any bug inside the nested if code region may remain hidden. In another case,

[1] http://linux.die.net/man/1/mpg321

when AFL gets stuck at the if condition at line 5, symbolic execution-based approaches such as Driller [47] will try to find new paths by sequentially negating path conditions. In this process, they may negate constraints at line 4 to find a new path, which may lead to some error handling code. AFL, however, has no knowledge of such error handling code and, as a result, it will start exploring in that direction. In general, there are several complex real-world code constructs that may hinder the progress of coverage-based fuzzers.

In order to understand such code constructs, we will walk through a more complex code snippet presented in Listing 3. Although VUzzer does not require source code, we use high-level C code for better illustration. The code reads a file and, based on certain bytes at *fixed* offsets in the input, it executes certain paths.

```
1 int main(int argc, char **argv){
2   unsigned char buf[1000];
3   int i, fd, size, val;
4   if ((fd = open(argv[1], O_RDONLY)) == -1)
5       exit(0);
6   fstat(fd, &s);
7   size = s.st_size;
8   if (size > 1000)
9       return -1;
10  read(fd, buf, size);
11  if (buf[1] == 0xEF && buf[0] == 0xFD) // notice the order of CMPs
12      printf("Magic bytes matched!\n");
13  else
14      EXIT_ERROR("Invalid file\n");
15  if (buf[10] == '%' && buf[11] == '@') {
16      printf("2nd stop: on the way...\n");
17      if (strncmp(&buf[15], "MAZE", 4) == 0) // nested IF
18          ... some bug here ...
19      else {
20          printf("you just missed me...\n");
21          ... some other task ...
22          close(fd); return 0;
23      }
24  } else {
25      ERROR("Invalid bytes");
26      ... some other task ...
27      close(fd); return 0;
28  }
29  close(fd); return 0;
30 }
```

Listing 3. Motivating example that illustrates issues in existing fuzzers

It is interesting to note that, when we ran the code snippet in Listing 3 with AFL [52], we could not reach the buggy state within 24 hours. What is so special about this code snippet and what is missing in fuzzers like AFL? We address this question by considering the following code properties:

1) **Magic bytes:** The second and the first byte are first compared to validate the input (line 11). If these bytes are not properly set at certain input offsets, the input is discarded immediately. In our example, offset 1 is checked first and offset 0 next. We observed this behavior in real applications, such as the `djpeg` utility. This also explains that it took millions of inputs for AFL to generate a valid jpeg image[2]. As AFL is not application-aware, it has no idea of such bytes and offsets. It will simply keep on *guessing* a valid combination of bytes and offsets.
2) **Deeper execution:** In order to go deeper in the execution, one has to get past another check at line 15, which compares offsets 10 and 11 (note that such offsets may be read from the input and thus vary across inputs, unlike the case of magic bytes). Irrespective of the result of this check, a new path is taken. However, the `true` branch will lead to buggy spot at line 18. Again, when processing this example, AFL spends a long time guessing the valid combination of bytes and offsets. In general, after a few iterations of input generation, a large percentage of inputs will be falling into the *error-handling* code. AFL and any other coverage-based fuzzer that searches for new basic blocks are likely to further explore from such inputs as these inputs have indeed found new code. However, if we consider the exploration of more meaningful and interesting paths, reusing such inputs yields no benefit and hinders further exploration of the application code.
3) **Markers:** In order to reach the buggy spot at line 18, there is a branch constraint to satisfy at line 17. It should be noted that such bytes may not be present at fixed offsets, but rather work as markers for certain fields in many input formats, such as JPEG, PNG, or GIF. Miller & Peterson show that the presence (and absence) of such markers has a direct impact on the executed code [36]. As often such markers are multibytes, AFL struggles with generating such bytes to execute certain paths.
4) **Nested conditions:** In the context of coverage-based fuzzing, each path is important. However, reaching certain paths may be more difficult than others. For example, in order to reach line 18, an input has to satisfy the check at line 17, which is only triggered when the constraint at line 15 is satisfied. Therefore, in order to increase the chance of reaching line 18, we need to fuzz any input that reaches line 15 more often. In the case of AFL, even if it passes or fails constraints at line 15, it discovers new paths in both cases and it tries to fuzz inputs corresponding to both the branches with equal probability. In this process, it spends a long time mutating the input executing the *easier path* and thereby minimizing the chances of reaching line 18. This is the result of spending less time in satisfying the constraint at line 17. Clearly, this strategy is not able to prioritize efforts to focus on the *interesting* path. A better strategy would be to optimize efforts based on the *control-flow* characteristics of the application.

Interestingly, one of the LAVA authors noted similar issues with AFL in a recently published post [16]. This supports our observation that black/greybox fuzzers like AFL tend to be *application-agnostic*, which makes them significantly less effective at discovering *hard-to-reach* bugs.

We note that some of the issues discussed above can be, in principle, handled by symbolic/concolic-based approaches [9], [20], [22], [33] such as Driller [47]. Driller combines AFL and concolic execution to explore deeper execution paths. With symbolic execution, we may be able to learn magic bytes quickly and assist AFL in crossing the first hurdle at line 11. However, AFL will get stuck again in the following lines. Moreover, this combination is again agnostic to nested conditions and thus path prioritization remains an issue. A more general and practical problem is the poor scalability of symbolic execution-based solutions. Although not an issue in this small motivating example, real-world applications are complex enough to drive symbolic execution into a state-explosion scenario. This is evident from the results presented in the Driller paper [47]: out of 41 binaries from DARPA CGC, Driller's concolic engine could generate new meaningful

[2]http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html

inputs only for 13 binaries. Another study [50] empirically established that symbolic execution is not suitable for finding inputs that explore interesting paths. Therefore, in spite of promising results on CGC binaries, the poor scalability of symbolic/concolic execution-based approaches is still a strong limiting factor on real-world applications.

B. Evolutionary Input Generation

Despite some pitfalls, AFL is a very promising fuzzer. The success of AFL is mainly attributed to its *feedback loop*, i.e, incremental input generation. In the case of our motivating example, it is almost impossible to generate an input that will reach the buggy state in one mutation. This motivated us to adopt an *evolutionary fuzzing* strategy, that is a fuzzing strategy that relies on an evolutionary algorithm (EA) for input generation. In the following, we briefly describe the main steps that a typical EA follows (see Algorithm 1). In the later sections, we will refer to these steps while detailing the main components of VUzzer.

Algorithm 1 Pseudo-code of a typical evolutionary algorithm

```
INITIALIZE population with seed inputs
repeat
    SELECT1 parents
    RECOMBINE parents to generate children
    MUTATE parents/children
    EVALUATE new candidates with FITNESS function
    SELECT2 fittest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution
```

Every EA starts with a set of initial inputs (seeds), which undergo the evolutionary process as follows. With some selection probability, one or two inputs (parents) are selected (SELECT1 state). Such inputs are then processed by two genetic operators, namely *crossover* (RECOMBINE state) and *mutation* (MUTATE state). In crossover, two inputs are combined by choosing an offset (*cut-point*) and exchanging the corresponding two parts to form two children. In mutation, we apply several mutation operators (like addition, deletion, replacement of bytes) on a single parent input to form one child. With this strategy, we get a new set of inputs which undergo the evaluation state (EVALUATE). In this state, we monitor the execution of each new input based on a set of properties. These properties are used in a *fitness function* to assess the suitability of the input. We choose the input with the highest fitness score for the next generation of inputs. The whole process continues until a termination condition is met: either the maximum number of generations is reached or the objective is met (in case of fuzzing, a crash is found).

III. Overview

To address the challenges mentioned in the previous section, we propose VUzzer, an application-aware evolutionary fuzzer. Figure 1 provides an overview of its main components. As VUzzer is an evolutionary fuzzer, there is a feedback loop to help generate new inputs from the old ones. When generating new inputs, VUzzer considers features of the application based on its execution on the previous generation of inputs. By considering such features, we make the feedback loop "smart" and help the fuzzer find inputs with non-zero IG with high frequency.

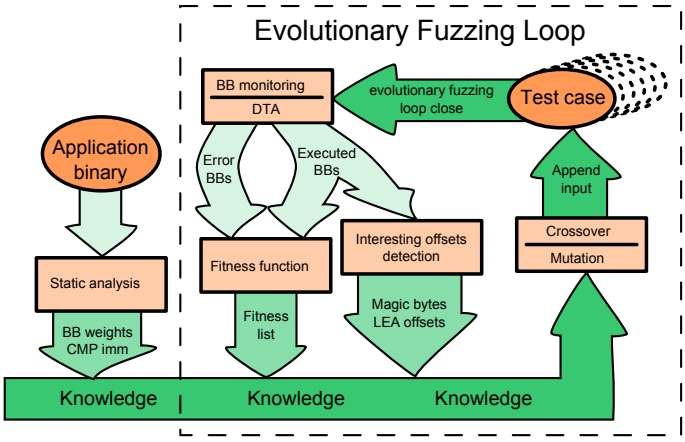

Fig. 1. A high-level overview of VUzzer. BB: basic block, CMP imm: `cmp` instruction with one immediate operand, DTA: dynamic taint analysis, LEA: *load effective address* instruction.

A. Features

The two main components of VUzzer are a static analyzer (shown on the left) and the main (dynamic) fuzzing loop (shown on the right). We use these components to extract a variety of *control-* and *data-flow* features from the application. Figure 1 shows that VUzzer continuously pumps this information back into the evolutionary mutation and crossover operators to help generate better inputs in the next generation. We first introduce the features and then discuss the static analyzer and the main fuzzing loop.

Data-flow features: Data-flow features provide information about the relationship between input data and computations in the application. VUzzer extracts them using well-known techniques such as taint analysis and uses them to infer the structure of the input in terms of the types of data at certain offsets in the input. As an example, it finds input bytes that determine branches ("branch constraints") by instrumenting each instruction of the `cmp` family of the x86 ISA to determine which input bytes (offsets) it uses and against which values it compares them. In this way, VUzzer can determine which offsets are interesting to mutate and what values to use at those offsets (providing partial answers to the questions in Section I). VUzzer is now able to mutate more sensibly by targeting such offsets more often and by using the intended values at those offsets to satisfy branch constraints. Doing so solves the problem of magic bytes, without resorting to symbolic execution.

Likewise, VUzzer monitors the `lea` instruction to check if the `index` operand is tainted. If so, it can determine that the value at the corresponding offset is of type `int` and mutate the input accordingly. Besides these two simple, but powerful features, many others are possible.

Control-flow features: Control-flow features allow VUzzer to infer the *importance* of certain execution paths. For example, Figure 2 shows a simplified CFG of the code in Listing 3. Inputs that exercise `error` blocks are typically not interesting. Therefore, identifying such *error-handling* blocks may speed up the generation of interesting inputs. We show how we detect error-handling code in the following sections. For now, we assume that we can heuristically identify the basic

blocks containing error handlers.

Another example concerns the reachability of nested blocks. Any input that reaches block F is more likely to descend deeper into the code than an input that reaches block H, since the latter is not nested. We use control-flow features to *deprioritize* and *prioritize* paths. As enumerating all the possible paths in the application is infeasible, we implement this metric by assigning *weights* to individual basic blocks. Specifically, basic blocks that are part of *error-handling* code get a negative weight, while basic blocks in *hard-to-reach* code regions obtain a higher weight.

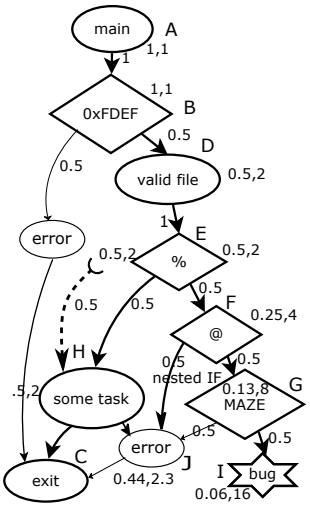

Fig. 2. A high-level CFG of the code shown in Listing 3. The number at each edge denotes the probability of the corresponding branch outcome. The number at each node denotes the overall probability of reaching the corresponding basic block. For example, if all edge probabilities are 0.5 and the program can reach a node N_2 either from N_0 directly or indirectly via N_1, the node probability of $N_2 \leftarrow 0.5 + 0.5 * 0.5 = 0.75$. The number next to the node probability is the assigned weight.

Figure 1 shows that a single iteration of fuzzing consists of several steps. VUzzer expects an initial pool SI of *valid* inputs, called seed inputs. The first step is to perform an intraprocedural static analysis to derive a few control-flow and data-flow features (Section III-B), which is followed by the main evolutionary fuzzing loop. In the remainder of this section, we walk through all the steps to describe the whole process.

B. The static analyzer

At the beginning of the fuzzing process, we use a lightweight intraprocedural static analysis to (i) obtain `immediate` values of the `cmp` instructions by scanning the binary code of the application and (ii) compute the weights for the basic blocks of the application binary.

The presence of many immediate values from `cmp` instructions in the application's code typically indicates that the application *expects* the input to have many of these values at certain offsets. For example, the analysis for the program in Listing 3 yields a list L_{BB} of weights for each basic block and a list L_{imm} of byte sequences containing {`0xEF, 0xFD, %, @, MAZE`}. To determine the basic block weights, we model the CFG of each function as a *Markov model* and compute the probability p_b of reaching each basic block b in a function. We then calculate the weight w_b of each basic block b as $1/p_b$. Thus, the lower the probability of reaching a basic block, the higher the weight. Using this model, the probability and the weight of each basic block is shown next to each node in Figure 2 (see Section IV-A3). We observe that, for example, the probability of reaching basic block G is less than that of reaching basic block F, which in turn has lower probability than basic block H. VUzzer uses these lists in later steps of the fuzzing loop.

C. The main fuzzing loop

We describe the main fuzzing loop by using the steps in Algorithm 1. Before the main loop starts, we execute the application with the set of seed inputs SI to infer an initial set of control-flow and data-flow features. For all the inputs in SI, we run dynamic taint analysis (DTA) to capture common characteristics of valid inputs. Specifically, we do so for the magic-byte and error-handling code detection mentioned earlier. Using these features, we generate an initial population of inputs as part of the INITIALIZE step in Algorithm 1. Note that our magic-byte detection ensures that these new inputs cross the first such check of the application. As DTA has a high overhead, we use it as sparingly as possible after the main loop starts.

Input execution: We execute the application with each of the inputs from the previous step and generate the corresponding traces of executed basic blocks. If any of the inputs executes previously unseen basic blocks, we taint the input and use DTA to infer its structural properties by monitoring the data-flow features of the application.

Fitness calculation: In the EVALUATE step of Algorithm 1, we calculate the fitness of each input as the weighted sum of the frequencies of executed basic blocks. We distribute the weights over the basic blocks using the weights list L_{BB}. Basic blocks that belong to error-handling code get a negative weight—for now we still assume that we can identify such basic blocks. The intuition behind this fitness calculation is to provide high scores to inputs that execute basic blocks with higher weights and thereby prioritize the corresponding paths, while also executing certain basic blocks with high frequencies to catch large loops. For example, let us consider two paths p_1 and p_2, executed by two inputs i_1 and i_2 respectively such that $p_1 = A \rightarrow B \rightarrow D \rightarrow E \rightarrow H \rightarrow J$ and $p_2 = A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow J$. For simplicity, let us assume the error-handling basic block J gets weight -1 and the frequency of execution of each basic block is 1. Using the weights from Figure 2, the weighted sums of the frequencies of p_1 and p_2 are 7 (1+1+2+2+2-1) and 9 (1+1+2+2+4-1). Hence, input i_2 gets a higher fitness score and will participate more in generating new inputs than i_1. This step eventually generates a sorted list L_{fit} of inputs in decreasing order of their fitness scores.

Genetic operators and new input generation: This is the final and most important functionality in our fuzzing strategy, encompassing the SELECT, RECOMBINE, and MUTATE steps in Algorithm 1. Together, these substeps are responsible for generating *interesting* inputs. In each iteration of the main loop, we generate a new generation of inputs by combining

and mutating the inputs from SI, all tainted inputs, and the top n% of L_{fit}. We refer to this set as the *ROOT* set.

Specifically, we generate new inputs via crossover and mutation. First, we randomly select two inputs (parents) from ROOT and apply crossover to produce two new inputs (children). With a fixed probability, these two inputs further undergo mutation. Mutation uses several sub-operations, such as deletion, replacement, and insertion of bytes at certain offsets in the given input. The mutation operator makes use of the data-flow features to generate new values. For example, when inserting or replacing bytes, it uses characters from L_{imm} to generate byte sequences of different lengths. Similarly, various offsets from current inputs' parents are selected for mutation. Hence, if any magic bytes exist, they will be replaced at the proper offsets in the resulting inputs.

This loop of input generation continues until we meet a termination condition. Currently, we terminate when we find a crash or when VUzzer reaches a pre-configured number of generations.

For easier reference, Table I provides a list of symbols that we use throughout the paper. In the next section, we elaborate

TABLE I. GLOBAL SYMBOLS AND THEIR MEANING.

Symbol	Description
DTA	dynamic taint analysis
SI	set of seed inputs (valid inputs).
L_{BB}	list of basic block weights, obtained by static analysis of the application binary.
L_{imm}	list of immediate values from cmp instructions, obtained by static analysis of the application binary.
L_{fit}	Sorted lists (in decreasing order) of fitness scores of inputs, obtained in the fitness calculation step.
O_{other}	set of all tainted offsets, other than the ones which are placeholder for magic bytes. This set is obtained by DTA.
L_{lea}	set of offsets that taint the $index$ operand of `lea` instructions.

on the algorithms that we use to derive relevant information about the input structure by using control-flow and data-flow features.

IV. DESIGN AND IMPLEMENTATION

In this section, we detail the techniques to calculate several primitives discussed in the previous section. The section also presents implementation details of VUzzer.

A. Design Details

1) Dynamic Taint Analysis (DTA): DTA is the core of VUzzer as it plays a major role in *evolving* new inputs. This is also the technique that sets VUzzer apart from existing fuzzers. DTA is used to monitor the flow of *tainted* input (e.g., network packets, files, etc.) within the application. DTA can determine, during program execution, which memory locations and registers are dependent on tainted input. Based on the granularity, DTA can trace back the tainted values to individual offsets in the input. VUzzer uses DTA to trace *tainted* input offsets used at `cmp` and `lea` instructions. For every executed `cmp` instruction `cmp op1, op2` (`op1` and `op2` can either be register, memory, or immediate operands), DTA determines if `op1` and/or `op2` are tainted by a set of offsets. Our DTA implementation is able to track taint at the byte level. For a given tainted operand *op*, DTA provides taint information for each byte of *op*. Symbolically, if *op* is denoted as b_3, b_2, b_1, b_0, then DTA provides taint information for each byte b_i separately. We denote the set of offsets that taint the j^{th} byte of the i^{th} operand of a given `cmp` instruction as T^i_j. We also record the values of these operands. Symbolically, we represent a tainted `cmp` instruction as $cmp_i = (offset, value)$, where $offset$ and $value$ are the sets of offsets from tainted input and the set of values for the untainted operand of the `cmp` instruction. For each `lea` instruction, DTA tracks only the $index$ register. L_{lea} contains all the offsets that taint such $indexes$.

2) Magic-byte Detection: Based on our understanding of file formats that have magic bytes, we postulate that a *magic byte is a **fixed** sequence of bytes at a **fixed** offset in the input string*. We have verified this assumption on several file formats that have magic bytes, for example, jpeg, gif, pdf, elf, and ppm. As VUzzer assumes the availability of a few valid inputs for a given application, we use the results of DTA on these inputs at the beginning of fuzzing. As applications expect the input to contain magic bytes, DTA's results on `cmp` instructions will contain the corresponding check for magic bytes.

For example, the code from Listing 3 expects a magic byte `0xFDEF` in the beginning of the input file. Hence, DTA will capture two `cmp` instructions—`cmp reg, 0xFD` with `reg` tainted by offset 0 and `cmp reg, 0xEF` with `reg` tainted by offset 1. If we have a set of valid inputs for this program, we can observe these two `cmp` instructions in all the corresponding executions. Conversely, if for a set of valid inputs we get $cmp_i = (o_i, v_i)$ in DTA's results for all the inputs, v_i is a part of the magic byte at offset o_i.

It should be noted that the algorithm we use to detect magic bytes can incur false positives. This may happen if all the initial valid inputs share identical values at the same offsets. Nonetheless, this will still be useful for generating inputs that go beyond the very initial check for magic bytes with a reduced probability of exploring different paths. To avoid this situation, we prefer to start with a *diverse* but valid set of inputs.

During magic-byte detection, for a given cmp_i instruction, if the corresponding $value$ depends on multiple offsets per byte, we do not consider such offsets to be magic-byte candidates. For example, for a given `cmp` instruction, if DTA detects that $|T^i_j| > 1$, we exclude such offsets ($\in T^i_j$) from any further consideration for magic-byte placeholders. Such a case indicates that the value of the corresponding operand may be *derived* from tainted values at those offsets $\in T^i_j$. The dependence on multiple bytes breaks the assumption that magic bytes are *fixed* (constant) sequences of bytes. We denote the set of all such offsets as O_{other}.

3) Basic Block Weight Calculation: From a coverage-based fuzzing perspective, every feasible path is important to traverse. A simple fuzzing strategy is to spend equal efforts to generate inputs for all feasible paths. However, due to the presence of control structures, the reachability of some paths may not be the same as that of other paths. This situation arises very frequently if we have *nested* control structures [41]. Therefore, any input that exercises such *hard-to-reach* code should be rewarded more compared to other inputs.

We incorporate this reward by assigning higher weights to basic blocks that are contained within nested control structures. As enumerating all the paths at the interprocedural level has trouble scaling, we constrain our analysis at the intraprocedural

level, i.e., we calculate weights for each basic block within the containing function. Later, we gather and add the weights of all the basic blocks in a path that is executed by a given input. With this strategy, we *simulate* the score of an interprocedural path by stitching several intraprocedural path scores together.

If we consider that the transition of an input at a particular basic block to the next basic block is dependent on some probability, we can derive a probabilistic model called Markov process for the input behavior from the control-flow graph (CFG). A Markov process is a stochastic process in which the outcome of a given trial depends only on the current state of the process [30]. We model the CFG of a function as a Markov process with each basic block having a probability based on its *connections* with other basic blocks.

For a given basic block, we assign equal probability to all its *outgoing edges*. Hence, if $out(b)$ denotes the set of all outgoing edges of basic block b, then $\forall e_{b*} \in out(b), prob(e_{b*}) = 1/|out(b)|$. The transition probability (likelihood) of a basic block b is calculated as follows:

$$prob(b) = \sum_{c \in pred(b)} prob(c) * prob(e_{cb}) \tag{1}$$

where $pred(b)$ is the set of all the predecessors of b. We employ a fixed-point iteration algorithm to compute the probability associated with each basic block in the CFG. The root basic block of the CFG is initialized with a probability of 1. Loops are handled by assigning a fixed probability of 1 to each backedge, thereby neglecting the effect of the backedge itself (i.e., we flatten the loop to speed up fixed point calculation). From Equation 1, the weight of each basic block b is given by:

$$w_b = \frac{1}{prob(b)} \tag{2}$$

4) Error-Handling Code Detection: As noted earlier, during fuzzing, the majority of mutated inputs will be executing a path ending up in some error state. Deprioritizing such execution paths is a key step towards increasing the chances of creating interesting inputs faster. Our error-handling detection heuristic relies on the availability of valid inputs, which is a prerequisite of VUzzer. As our error-handling detection depends on the dynamic behavior of the application, it detects error-handling basic blocks in an incremental manner.

Initial analysis: For each valid input $i \in SI$, we collect a set BB(i) of basic blocks that are executed by i. Let $Valid_{BB}$ denote the union of all such sets of executed basic blocks by all valid inputs. We then create a set of *totally random* inputs, denoted as TR. For each input in this set, we collect its execution trace in terms of basic blocks. A basic block from such a set of executions is assumed to be a error-handling basic block (i.e., belonging to error-handling code) if it is present in each execution of inputs from TR and it is not present in $Valid_{BB}$. The intuition is that since SI is a set of valid inputs, no error-handling code will be triggered. Therefore, $Valid_{BB}$ will contain *only* basic blocks that correspond to valid paths. And since TR is a set of totally random inputs, they will be very likely caught by error-handling code during the execution.

Ours is a very conservative error-handling basic block detection strategy as we may miss few basic blocks if certain inputs are caught by different error-handing code. Nonetheless, note that we will never classify a basic block corresponding to a valid path as an error-handling basic block. More formally, let

$$Valid_{BB} = \cup_{i \in SI} \text{BB}(i), \text{ then}$$
$$EHB = \{b : \forall k \in TR, b \in BB(k) \& b \notin Valid_{BB}\}$$

where EHB is a set of error-handling basic blocks.

Incremental analysis: We observe that since our error-handling detection strategy is based on the dynamic behavior of the application, not all error-handling code may be triggered during the *initial analysis*. As inputs evolve, they explore more paths and thus encounter new error-handling code. For this reason, we initiate an *incremental analysis* during later iterations of fuzzing. In our experimental setup, we observed that, as we proceed with more iterations of fuzzing, the number of new error-handling code instances decreases. This reflects the intuition that software has a finite number of error-handling code instances, which are reused in different parts of the application. Therefore, we run our incremental analysis less frequently as we execute more iterations.

The intuition behind our *incremental analysis* is the observation that as fuzzing proceeds, the *majority* of newly generated inputs will end up triggering some error-handling code. At a given iteration, let I be the set of inputs generated in the iteration. Let the *majority* be quantified by $n\%$ of $|I|$. Our (offline) experiments show that $n = 90$ is a reasonable choice. Let $BB(I)$ be the set of all the basic blocks executed by inputs in I. We classify a basic block b from $BB(I)$ as an error-handling basic block if it is associated to at least $n\%$ of the inputs from I and it is not in the $Valid_{BB}$ set. More formally, let $\mathcal{P}(I)$ denote the power set of I. Then

$$EHB = \{b : \forall k \in \mathcal{P}(I), \text{ s.t. } |k| > |I| * n/100,$$
$$b \in BB(k) \& b \notin Valid_{BB}\}$$

Weight calculation for error-handling basic blocks: After detecting error-handling basic blocks (EHBs), we want to deprioritize paths that contain such blocks. We achieve this by *penalizing* the corresponding inputs so that such inputs have less chances of participating in next generation. For this purpose, each EHB is given a negative weight, which impacts the fitness score of the corresponding inputs (see Section IV-A5). However, this strategy is alone insufficient since, as EHBs are a small minority when compared to the total number of basic blocks executed by an input, such a small quantity will have negligible impact. We solve this problem by defining an *impact coefficient* μ (a tunable parameter) that decides *how many (non-error handling) basic blocks may be nullified by a single error-handling basic block*. Intuitively, this parameter determines that, once an input enters error-handling code, the contribution of any of the corresponding basic blocks when calculating fitness scores must be reduced by a factor μ. For a given input i, we use the following formula for weight calculation purposes.

$$w_e = -\frac{|\text{BB}(i)| \times \mu}{|\text{EHB}(i)|} \tag{3}$$

where $|\text{BB}(i)|$ is number of all the basic blocks executed by input i, $|\text{EHB}(i)|$ is the number of all the error-handling basic blocks executed by i, and $0.1 \leq \mu \leq 1.0$.

5) Fitness Calculation: Fitness calculation is one of the most important components of evolutionary algorithms. This is crucial to implement the feedback loop, which fuels the next step of input generation. Once a new input is generated, the chances of its participation in generating new inputs depend on its fitness score.

VUzzer assess the fitness of an input in two ways. If the execution of an input results in discovering a new non-EHB basic block, the input qualifies for participation in the next generation. This is similar to AFL (with the additional use of the EHB set). However, this measure of fitness considers all newly discovered paths equal, which is problematic, as explained earlier. The importance (and thus the fitness) of an input depends on the *interestingness* of the path that it executes, which, in turn, depends on the weights of the corresponding basic blocks. Therefore, we define fitness f_i of an input i as a function that captures the effect of all the corresponding basic block weights.

$$f_i = \begin{cases} \frac{\sum_{b \in \text{BB}(i)} \log(\text{Freq}(b))W_b}{\log(l_i)} \text{BBNum} & \text{if } l_i > \text{LMAX} \\ \sum_{b \in \text{BB}(i)} \log(\text{Freq}(b))W_b & \text{otherwise.} \end{cases} \quad (4)$$

where $\text{BB}(i)$ is the set of basic blocks executed by input i, $\text{Freq}(b)$ is the execution frequency of basic block b when executed by i, W_b is the weight of basic block b (by using Equation 2), l_i is the length of input i, and LMAX is a pre-configured limit on input length. LMAX is used to address the phenomenon of input *bloating*. In the parlance of genetic algorithms, both of the fitness criteria (i.e. ability to discovering new basic block and a higher f_i) correspond to the notion of *exploration* and *exploitation*—discovering a new basic block indicates a new direction (i.e., exploration) and a higher f_i indicates higher execution frequencies (among other factors) of basic blocks (i.e., exploitation in the same direction).

6) Input Generation: VUzzer's input generation consists of two parts, crossover and mutation, which are not mutually exclusive, that is, crossover is followed by mutation with a fixed probability.

Crossover: Crossover is a simple operation wherein two parent inputs are selected from the previous generation and two new child inputs are generated. Figure 3 illustrates the process of generating two child inputs from two parent inputs.

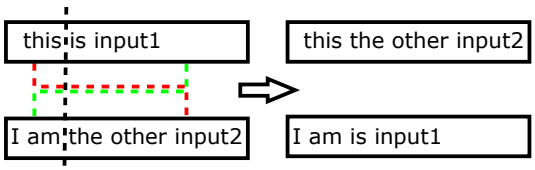

Fig. 3. Crossover operation in VUzzer. In this single-point crossover, we select the 5^{th} offset as cut-point. For the first parent (`this is input1`), this strategy breaks it into two parts: i_1^1=`this` and i_1^2= `is input1`. For the second parent, we again get two parts: i_2^1=`I am` and i_2^2= `the other input2`. Now, we form two children by using $i_1^1 \mid i_2^2$ and $i_1^2 \mid i_2^1$, that is, `this the other input2` and `I am is input1`.

Mutation: Mutation is a more complex operation, which involves several suboperations to change a given parent input into the corresponding child input. The process is detailed in the following steps:

Step 1: Randomly select tainted offsets from the set O_{other} and insert strings at those offsets. The strings are formed by bytes obtained from the set L_{imm}.
Step 2: Randomly select offsets from the set L_{lea} and mutate such offsets in the string from Step 1 by replacing them with *interesting* integer values, such as, 0, MAX_UINT, negative numbers.
Step 3: For all the tainted `cmp` instructions for the parent input, if the values of $op1 \neq op2$, replace the value at the tainted offset in the string from Step 2 with the value of `op2` or else with a fixed probability replace the tainted byte by a random sequence of bytes.
Step 4: Place the magic bytes at the corresponding offsets as determined by our magic-byte detector.

B. Implementation Details

The core functionality of VUzzer is implemented in Python 2.7. Some of the implemented analyses, for example *incremental analysis* for error-handling basic block detection, are memory intensive and therefore we also make use of efficient data structures provided by more recent versions, such as BitVector[3]. VUzzer internally consists of two main components, comprising static and dynamic analyses, as further detailed below.

Static analysis: VUzzer implements both of the static analyses (constant string extraction and basic block weight calculation) within IDA [27]. The analysis is written in Python using IDAPython [18].

Dynamic analysis: VUzzer implements both dynamic analyses (basic-block tracing and DTA) on the top of the Pin dynamic analysis framework [31]. For basic block tracing, we implemented a pintool to record each basic block, along with its frequency, encountered during the execution. Our pintool can selectively trace basic blocks executed by certain libraries on-demand. Selective library monitoring allows us to reduce the execution trace overhead and focus on the intended application code.

Our DTA implementation is based on DataTracker, proposed by Stamatogiannakis *et.al* [46], which in turn is based on LibDFT [29]. As LibDFT can only handle 32-bit applications, the current VUzzer prototype can only be used to fuzz 32-bit applications (also used in our evaluation). Note that this is not a fundamental limitation and we are, in fact, in the process of implementing 64-bit support in VUzzer. Any updated version will be made available at https://www.vusec.net/projects/fuzzing.

To make it suitable for our purposes, we also made several changes to DataTracker:

- In DataTracker, taint tags associated with each memory location are modeled as tuples: `<ufd, file_offeset>`, i.e., unique file descriptor and the offset of the file associated with that descriptor. Each of these tuples is 64 bit long (32 bits for `ufd` and 32

[3]https://pypi.python.org/pypi/BitVector/3.4.4

bits for `file_offset`). Each memory location has a set of these tuples associated with it to determine the offsets and the files by which the memory location is tainted. We changed this to a EWAHBoolArray type[4] which is a compressed bitset data type. Since we only need data-flow information from one (input) file, we modified DataTracker to propagate taint only through that file. Thus, in our modified version, the taint tags associated with each memory location are modeled as a EWAHBoolArray which only contains offsets. As a result, our implementation is at least 2x faster and uses several times less memory than DataTracker.
- We added instrumentation callbacks for the `cmp` family of instructions like `CMP, CMPSW, CMPSB, CMPSL` and the `lea` instruction to catch byte-level taint information for the operands involved in the computations.
- We rewrote hooks for each implemented system call and also added hooks for some extra system calls such as pread64, dup2, dup3, mmap2, etc. To evaluate our performance on the DARPA dataset [15], we also implemented hooks for DECREE-based system calls, which are different from normal Linux system calls.

Crash triage: Once fuzzing starts producing crashes, it may continue to produce more crashes and there should be some mechanism to differentiate crashes due to different bugs (or the same bug but different instances). In order to determine the uniqueness of a crash, VUzzer uses a variant of *stack hash*, proposed by Molnar *et.al.* [37]. In our pintool, we implemented a ring buffer that keeps track of the last 5 function calls and the last 10 basic blocks executed before we get a crash. We calculate the hash of this buffer and each time a new crash is encountered, we compare the newly generated hash with the older ones to determine if the reported crash is a new unique one.

V. Evaluation

In order to measure the effectiveness of our proposed fuzzing technique, this section presents an evaluation of VUzzer. To expose VUzzer to a variety of applications, we chose to test VUzzer on three different datasets A. DARPA CGC binaries [15], B. miscellaneous applications with binary format as used in [43], and C. a set of *buggy* binaries recently generated by LAVA [17].

We ran our experiments on an Ubuntu 14.04 LTS system equipped with a 32-bit 2-core Intel CPU and 4 GB RAM. For the DARPA CGC dataset, the (provided) environment is a VM with a customized OS called DECREE. We want to emphasize that our main evaluation goal is to show how effective VUzzer is in identifying bugs (that may be buried deep in the execution) with much fewer inputs than state-of-the-art fuzzers such as AFL. Our current VUzzer prototype is not as optimized for fast input execution as AFL and we therefore seek no comparison in this direction.

A. DARPA CGC Dataset

As part of *Cyber Grand Challenge*, DARPA released a set of binaries that run in a customized OS called DECREE. There are 131 binaries in total, with various types of bugs injected in them. However, we could not run VUzzer on all of them for the following reasons:

- All of the binaries are interactive in nature by accepting inputs from `STDIN`. Once started, many of them present a menu to choose an action, including the option to *quit*. Furthermore, in many cases, there are multiple menus (in a different state of the program) with different options to quit. As VUzzer requires a step to generate totally random inputs (error-handling code detection, Section IV-A4), executing such inputs puts the application in a loop, looking for *valid* options, including the option to quit. This causes the application to run forever. This is an interfacing problem and not a fundamental limitation of our fuzzing method.
- Some of these binaries are compiled with *floating-point instructions*, which are not handled by LibDFT and thus VUzzer cannot get correct data-flow information.
- As VUzzer is based on Pin [32], we followed the given procedure to run pintools in DECREE[5]. However, we could not run some of the binaries with Pin.
- Some of the binaries involve interaction with other binaries, which is not handled by VUzzer.

After considering the obstacles mentioned above, we are left with a total of 63 binaries. In order to make a comparison with AFL, we also ran AFLPIN, a pintool-based AFL implementation[6]. AFLPIN has the same fuzzing engine as AFL, but a different mechanism to get execution traces. Our choice to use AFLPIN instead of AFL is to have an identical interfacing mechanism with the SUT, that is, passing input to the pintool via file descriptor 0 (`STDIN`).

VUzzer found crashes in 29 of the CGC binaries, whereas AFLPIN found only 23 crashes. As each CGC is also accompanied with a patched version, we verified each bug found by VUzzer by running the patched version of the binary to observe no further crashes. The most important result was the number of executed inputs per crash in both of these fuzzers. We ran both fuzzers for a maximum of 6 hours. Figure 4 depicts the number of executions for the cases where both of the fuzzers found crashes (13 in total), evidencing that VUzzer can significantly prune the search space compared to AFL.

While fuzzing a specific binary `NRFIN_00015`, we observed the importance of computing the fitness score f_i in a discrete manner. The vulnerability in this binary is a typical case of buffer overflow in a `loop`. We observed that after generation 18, there was no new BB discovered, but f_i kept increasing, indicating typical *loop* execution behavior. At generation 63 (total executions 13K), we reach the boundary of the buffer. AFLPIN could not detect this crash.

We note that our current results on this dataset are modest, especially in the view of the results reported in Driller [47]. We further investigated the results and found some peculiarities that may interfere with the performance of our current VUzzer prototype on CGC.

[4]https://github.com/lemire/EWAHBoolArray

[5]https://github.com/CyberGrandChallenge/cgc-release-documentation/blob/master/walk-throughs/pin-for-decree.md

[6]https://github.com/mothran/aflpin

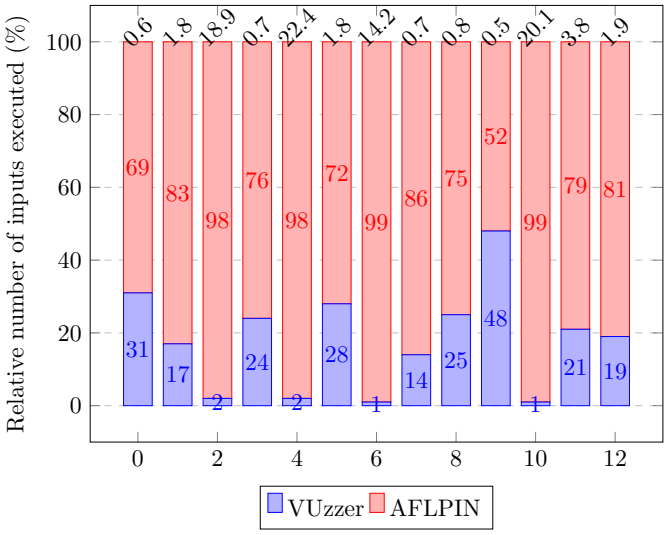

Fig. 4. Relative number of inputs executed for each of the CGC Binaries, wherein both VUzzer and AFLPIN find crashes. The numbers above the bars are the total number of inputs (in thousands) executed.

- In several binaries, a buggy state is reached only by performing a very *specific* set of actions from a given menu. For exmaple, in the `CROMU_00001` application, one has to perform: `login A -> send many msg to user B -> login B -> check msg .` Currently, VUzzer does not have capabilities to repeat a sequence.
- The notion of *valid* input is blurry. Recall that we use a *whole* session, provided in the form of XML files by every CGC binary, as one input. Therefore, there is essentially no notion of *invalid* inputs. Because of this, we cannot exploit the full power of VUzzer.
- Related to the above point is the issue of *interesting* offsets. As the CGC binaries are interactive, the input is essentially a sequence to explore application state, which may vary from one input to another. For example, one of the binaries allows a user to load a file. The bug is triggered while processing the file. The corresponding file loading menu can appear anywhere in the input and therefore the offsets in the file are relative to where it was loaded in the input, making it difficult to automatically reason over offsets.

In light of the aforementioned issues, we believe that VUzzer is not suitable for interactive programs, mainly because ot its poor interfacing mechanism with such programs.

B. LAVA Dataset

In a recent paper, Dolan-Gavitt *et.al.* developed a technique to inject *hard-to-reach* faults and created buggy versions of a few Linux utilities [17] for testing fuzzing- and symbolic execution-based bug finding solutions. We used the **LAVA-M** dataset [17] to evaluate VUzzer. This dataset consists of 4 Linux utilities—`base64`, `who`, `uniq`, and `md5sum`—each injected with multiple faults (in the same binary for each utility). The LAVA paper reports results on the evaluation of a coverage-based fuzzer (FUZZER), symbolic execution, and a SAT-based approach (SES) on these buggy applications.

To improve readability, we restate the results from the original LAVA paper in Table II. The last column in Table II shows the results produced by VUzzer. The numbers shown are the total unique bugs identified by VUzzer. In the case of `md5sum`, we could not run VUzzer as it crashed on the first round of input generation without allowing the program to parse more of any input. Each injected fault in the LAVA binaries has an ID and the ID is printed on standard output before each binary crashes due to that fault. This allowed us to precisely identify the faults triggered by VUzzer. Table III reports the IDs of the faults triggered by VUzzer for each LAVA binary.

TABLE II. LAVA-M DATASET: PERFORMANCE OF VUZZER COMPARED TO PRIOR APPROACHES.

Program	Total bugs	FUZZER	SES	VUzzer (unique bugs, total inputs)
`uniq`	28	7	0	27 (27K)
`base64`	44	7	9	17 (14k)
`md5sum`	57	2	0	1*
`who`	2136	0	18	50 (5.8K)

A few interesting points emerge from our LAVA dataset results. Most of the LAVA injected faults are based on *artificially* injected path conditions, like `lava` to reach a particular path and trigger the bug. This is very well captured by VUzzer, thanks to its data-flow features. For example, during `base64` fuzzing, we learned that the first four bytes should be either `'val` or `lav'` to follow a particular path. Similarly, we discovered that the last few bytes should contain any of the following values to take different paths: `las[`, `lat\x1b`, `Wsal`, etc. It should be noted that most of the path constraints injected by LAVA are multibyte constraints. Such constraints pose a serious problem for AFL to traverse deeper in the execution (as also noted in [16]). Another interesting point is the performance of VUzzer on `who`. The fuzzer used in the LAVA paper failed to find even a single bug, whereas VUzzer found several unique crashes.

TABLE III. FAULT IDS OF BUGS DETECTED BY VUZZER ON THE LAVA-M DATASET.

Program	Fault IDs
`uniq`	468, 318, 293, 170, 130, 443, 171, 393, 169, 368, 112, 322, 166, 227, 371, 472, 321, 215, 222, 297, 372, 396, 446, 397, 471, 296, 447
`base64`	1, 843, 817, 386, 786, 805, 576, 276, 222, 806, 284, 841, 584, 235, 278, 583, 788
`md5sum`	-
`who`	4159, 4343, 3800, 83, 1188, 60, 137, 138, 1960, 59, 1458, 1, 159, 5, 1803, 1314, 79, 475, 18, 4, 9, 1804, 1816, 10, 7, 3, 58, 985, 179, 14, 319, 2617, 81, 22, 2, 63, 4364, 8, 672, 341, 26, 255, 20, 75, 474, 6, 4358, 4362, 587, 89

Overall, on both *artificial* datasets, VUzzer reported encouraging results, although, as expected, it did struggle with interactive programs in the DARPA CGC dataset. We now move on to evaluate VUzzer on real-world programs that have also been considered by other fuzzers.

C. Various Applications (VA) Dataset

We use a dataset of real-world programs (`djpeg/eog`, `tcpdump`, `tcptrace`, `pdf2svg`, `mpg321`, `gif2png`) to evaluate the performance of VUzzer. Rebert *et al.* also evaluated these programs to report on several bugs [43] and we therefore included these programs in our evaluation for

comparison purposes. For each of these programs, we use the *vanilla release* in Ubuntu 14.04. We remark that by, evaluating these utilities, we also targeted some well-known libraries, such as `libpcap`, `libjpeg`, `libpoppler`, and `libpng`. Each program is fuzzed for maximum 24 hours. In order to highlight the performance of VUzzer, we also ran AFL on these applications. Table IV shows the results of running VUzzer and AFL on the VA dataset, with VUzzer significantly outperforming AFL for both number of unique crashes found and number of inputs required to trigger such crashes.

TABLE IV. VA DATASET: PERFORMANCE OF VUZZER VS. AFL.

Application	VUzzer		AFL	
	#Unique crashes	#Inputs	#Unique crashes	#Inputs
mpg321	337	23.6K	19	883K
gif2png+libpng	127	43.2K	7	1.84M
pdf2svg+libpoppler	13	5K	0	923K
tcpdump+libpcap	3	77.8K	0	2.89M
tcptrace+libpcap	403	30K	238	3.29M
djpeg+libjpeg	1[7]	90K	0	35.9M

Figure 5 details the distribution of crashes over a period of 24 hours. The x-axis of each plot shows the cumulative sum of crashes, sampled at each 2 hours. As shown in the figure, for almost every application, VUzzer keeps finding crashes during the later iterations of fuzzing, whereas AFL quickly exhausts its efforts after a few initial iterations. This is due to the fact that, at later stages, AFL is not able to find new (deeper) paths, whereas VUzzer is able to learn branch constraints as it explores new paths and thus it is able to find crashes in later stages of fuzzing. Another interesting point to note in Figure 5 is that, in comparison to AFL, VUzzer in not only able to find crashes with much fewer inputs, but this also happens in much less time (see the position of the vertical line in Figure 5). We want to again remark that we have not optimized VUzzer for fast input execution. We believe that there exist several techniques to enhance the execution speed of VUzzer, for example, using an AFL-like *fork-server* within a single fuzzing iteration or distributing concurrent fuzzing workers across multiple cores or machines.

D. Crash-Triage Analysis

Fuzzers tend to generate a large number of crashes. Fixing every bug associated with a crash is a time-consuming but lucrative process. The only information provided to a software developer is the version number of the application and the crash itself. Naturally, the bug patching efforts are invested in the bugs that are more (security) critical.

!Exploitable [19], a tool proposed by CERT, is built on top of GDB and uses heuristics to asses the exploitability of a crash caused by a bug. The heuristics are based on the crash location, the memory operation (read or write), and the signals triggered by the application. While this analysis is not sound, it is simple, fast, and provides hints on the severity of a crash. We use the !Exploitable tool to rank the crashes found by VUzzer on this dataset. Table V presents our results.

As shown in the table, most of the cases were marked as *Unknown* due to the simplicity of the !Exploitable tool. None of the cases were marked as *Probably Exploitable*. Finally, every crash discovered by VUzzer in `tcptrace` seems to be *Exploitable*. We investigated one of the crashes in `tcptrace` and there is a seemingly obvious way to exploit it: the vulnerability is an out-of-bounds write to a heap buffer. The bound and the data that are written are tainted (i.e., attacker-controlled).

[7]No crash, but infinite loop resulting in an out-of-memory error.

TABLE V. PERCENTAGES OF EXPLOITABLE BUGS DISCOVERED BY VUZZER AS REPORTED BY !EXPLOITABLE TOOL.

	Unknown	Exploitable	Probably Not Exploitable
gif2png	100.0	0.0	0.0
mpg321	100.0	0.0	0.0
pdf2svg	87.5	0.0	12.5
tcpdump	100.0	0.0	0.0
tcptrace	0.0	100.0	0.0

To further analyze the quality of the bugs discovered by VUzzer, we measured the distance between the crash and the library involved (if any). A bug located in a library will likely be included in any application that uses that library, hence these bugs are of high priority. We need to also keep in mind, that these are unknown bugs and therefore many of them could be *zero-day*. As we found a large number of unique crashes, reporting the most important ones early is a priority and therefore we rely on an automated analysis to approximate the *severity* of a bug. In short, if a crash happens in a library, then it is a serious bug to report. However, sometime a bug manifests itself in the user application, but the real cause of the bug lies in a library used by the application. We, therefore, also measure the distance from the last library call, when a crash in observed in the application code.

The distance between the crash and a library is measured by two metrics. First we count the number of instructions executed between the crash and the last library call. The intuition is that the computation (and its side effects) which ultimately caused the crash might originate in a library call. Second, we count the number of stack frames between the crash and the last library call. As an example, libraries using output function hooks that reside in the main application (e.g. `tcpdump`, `tcptrace`, `mpg321`) are covered by such heuristics. Table VI presents the results of our analysis.

TABLE VI. DISTANCE BETWEEN CRASHES AND LIBRARY CALLS.

	#Instructions	#Stack frames
gif2png	20554.00	gif2png (0); libc (5)
mpg321	733.04	**libid3tag** (0); libmad (3.1); libc (3.9); mpg321 (5.5)
pdf2svg	626.11	libc (1); **libpoppler** (3); **libpoppler-glib** (8); pdf2svg (9);
tcpdump	293.50	tcpdump (0); **libpcap** (5.7)
tcptrace	1134.53	tcptrace (0); **libpcap** (2); libc (7)

All crashes in `mpg321` happened inside the (`libid3tag`) library. The `libid3tag` library is heavily patched (patch level is 10) by the distro maintainers. This shows that this library is known to contain many bugs. `gif2png` always crashed inside the application. This is confirmed by both metrics with high figures. `pdf2svg` crashed in `libpoppler` most of the time. The stack frame distance is 3 because the signal gets *routed* from Linux' `vdso` through the standard library. `tcpdump` and `tcptrace` use the same (`libpcap`) library but, since `tcpdump` displays the content of the network flow, it has a higher distance from the library.

Based on the aforementioned analysis, we believe many of the crashes reported by VUzzer uncover *zero-day* vulnera-

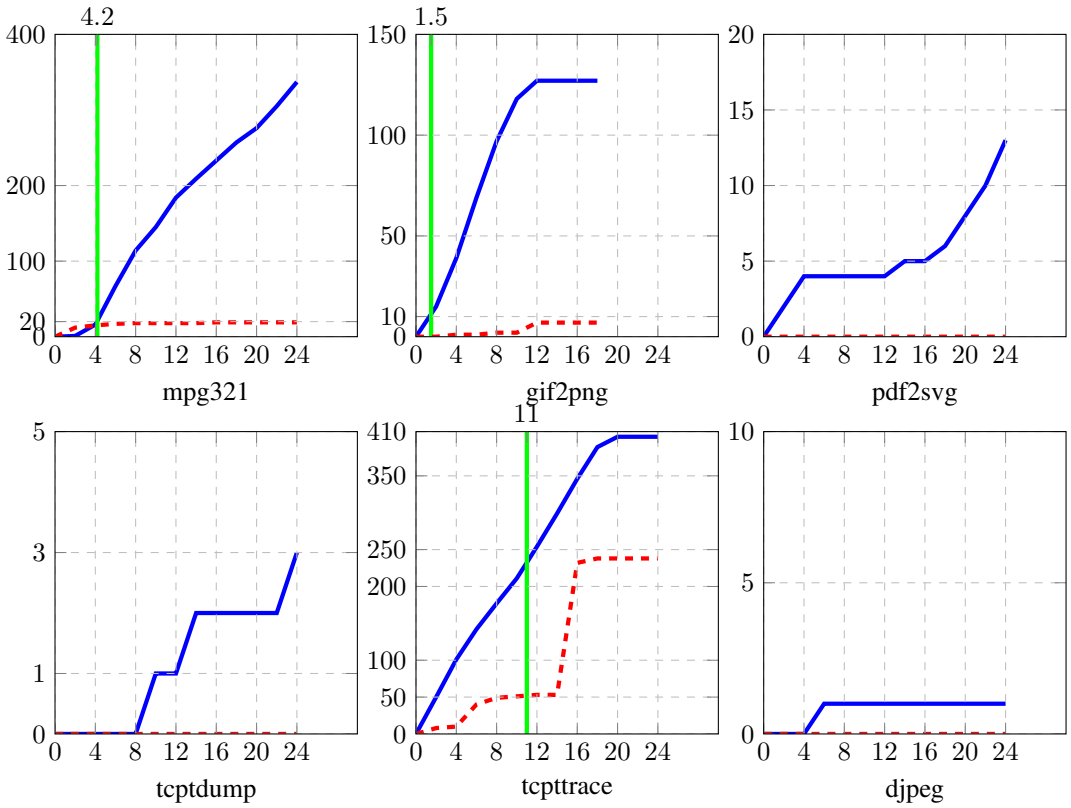

Fig. 5. Distribution of crashes over a time period of 24 hours. X-axis: cummulative sum of creahes. Y-axis: time (over 24 hours). Blue line: VUzzer. Red dashed line: AFL. Vertical green line: Time taken by VUzzer to find the same number of crashes as those found by AFL during a complete run.

bilities and we are currently in the process of performing responsible disclosure to the open-source community. Table VII provides information on some of the bugs that we have analyzed and reported so far.

VI. Related Work

In the previous sections, we have already highlighted some of the major differences between VUzzer and state-of-the-art fuzzers like AFL. In this section, we survey additional recent research work in the area of fuzzing. This enables us to highlight some of the features and differences with respect to existing work.

A. Search-based Evolutionary Input Generation

The use of evolutionary algorithms for input generation purposes is a well-explored research area in software engineering [7], [34]. There have been attempts to use evolutionary algorithms for input generation to discover vulnerabilities in applications [25], [42], [45]. The difference lies in the fact that these approaches assume *a-priori* knowledge of the application to focus on the paths leading to vulnerable parts of the program. This property makes these approaches closer to *directed* fuzzing and, therefore, our fuzzing strategy deviates from them substantially. Unlike VUzzer, and similar to AFL, the feedback loop used by these approaches does not attempt to relate application behavior with the input structure to enhance input generation.

B. Whitebox Fuzzing Approaches

Whitebox fuzzing is one of the earliest attempts to enhance the performance of *traditional random* fuzzing by considering the properties of the application. There exist a number of approaches to make fuzzing more efficient, for example, by applying symbolic execution and dynamic taint analysis to solve branch constraints [20]–[24], [26]. Although VUzzer differs from these approaches in a number of ways, the fundamental difference remains the use of symbolic execution. Similar to VUzzer, BuzzFuzz, proposed by Ganesh *et.al.* [20] makes use of dynamic taint analysis, but for an entirely different purpose. BuzzFuzz is a directed fuzzer and, therefore, it does not try to learn constraints for every path. It instead uses taint analysis to detect bytes that influence *dangerous spots* in the code, like library call arguments, and mutate these bytes in the input to trigger exceptional behavior. Most of these approaches also require the availability of source code to perform analysis.

C. Blackbox/Graybox Fuzzing Approaches

In spite of being simple and fully application agnostic, blackbox fuzzers, like, Peach [1], Sulley [39], and Radamsa [40] have discovered bugs in real-world applications. However, throughout the paper, we have already discussed the limitations of such fuzzers.

Recently, symbolic and concolic execution-based fuzzing approaches have dominated the area of "*smart*" fuzzing [12], [38], [47], [51]. Mayhem [12], a system from CMU to automatically find exploitable bugs in binary code, uses several

TABLE VII. ANALYSIS OF NEW BUGS FOUND BY VUZZER.

Program	Bug Type	Already fixed?	Reported?
tcpdump	Out-of-bounds Read	Yes	No
mpg321	Out-of-bounds Read	No	Yes [2]
mpg321	Double free	No	Yes [3]
pdf2svg	Null pointer deref (write)	Seems to be fixed in poppler 0.49	No
pdf2svg	Abort	Seems to be fixed in poppler 0.49	No
pdf2svg	Assert fail (abort)	Yes [4]	No
tcptrace	Out-of-bounds Read	No	Yes [5]
gif2png	Out-of-bounds Read	No	Yes [6]

program analysis techniques, including symbolic and concolic execution, to reason about application behavior for a given input. This is similar in spirit to VUzzer. However, since the goal of VUzzer differs from that of Mayhem, VUzzer does not require heavyweight program analysis techniques and instead *infers* important properties of the input just by applying heuristics based on lightweight program analysis. Similarly, Driller [47] uses hybrid concolic execution techniques [33] to assist fuzzing by solving branch constraints for deeper path explorations. In [28], Kargén *et.al.* propose a different approach to generate fuzzed inputs. For a given application that is being tested, their approach modifies another *input producer* application by injecting faults that influence the output. Using this strategy, the buggy program generates *mutated* inputs. However, it is not clear if these mutated inputs indeed affect the way application consumes these inputs. TaintScope [49]—a checksum-aware fuzzer—uses taint analysis to *infer* checksum-handling code, which further helps fuzzing bypass checksum checks. VUzzer can also benefit from this (complementary) technique while fuzzing. In a very recent work [8] (concurrent to our work), the authors of AFLFAST proposed a markov-model based technique to identify *low-frequency* paths to focus fuzzing efforts in that direction. The heuristic, also used by VUzzer partially, is to deprioritize paths that are executed by maximum number of inputs. VUzzer's error-handling basic-block detection technique is similar to this, albeit much light-weight. VUzzer applies other data- and control-flow features to speed-up the input generation.

There have been several other techniques to enhance fuzzing [11], [43], [51]. VUzzer can also benefit from these approaches, in multiple ways. For example, Seed selection [43] can help VUzzer start with a *good* set of seed inputs.

VII. CONCLUSIONS

This paper argues that the key strength of fuzzing is to implement a *lightweight*, *scalable* bug finding technique and applying *heavyweight* and *non-scalable* techniques, like symbolic execution-based approaches, is not the definitive solution to improve the performance of a coverage-based fuzzer. After studying several existing general-purpose (black/graybox) fuzzers, including the *state-of-the-art* AFL fuzzer, we note that they tend to be *application agnostic*, which makes them less effective in discovering deeply rooted bugs. The key limitation of application-agnostic strategies is their inability to generate *interesting* inputs faster. We address this problem by making fuzzing an *application-aware* testing process.

We leverage control- and data-flow features of the application to *infer* several interesting properties of the input. Control-flow features allows us to *prioritize* and *deprioritize* certain paths, thereby making input generation a controlled process. We achieve this by assigning weights to basic blocks and implement a weight-aware fitness strategy for the input. By using dynamic taint analysis, we also monitor several data-flow features of the application, providing us with the ability to infer structural properties of the input. For example, this provides us with information on which offsets in the input are used at several branch conditions, what values are used as branch constraints, etc. We use these properties in our feedback loop to generate new inputs.

We have implemented our fuzzing technique in an open-source prototype, called VUzzer and evaluated it on several applications. We also compared its performance with that of AFL, showing that, in almost every test case, VUzzer was able to find bugs within an order of magnitude fewer inputs compared to AFL. This concretely demonstrates that inferring input properties by analyzing application behavior is a viable and scalable strategy to improve fuzzing performance as well as a promising direction for future research in the area.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments. We would also like to thank the LAVA team for sharing the LAVA corpus privately with us much before the official public release. This work was supported by the European Commission through project H2020 ICT-32-2014 SHARCS under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI Dowsing and NWO 628.001.006 CYBSEC OpenSesame.

REFERENCES

[1] "Peach fuzzer," http://www.peachfuzzer.com/.

[2] 2016, http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=844634.

[3] 2016, https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=844626.

[4] 2016, https://bugs.freedesktop.org/show_bug.cgi?id=85141.

[5] 2016, https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=844719.

[6] 2016, https://gitlab.com/esr/gif2png/issues/1.

[7] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based grey-box fuzzing as markov chain," in *CCS'16*. New York, NY, USA: ACM, 2016, pp. 1032–1043.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *CCS'06*. ACM, 2006, pp. 322–335.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[11] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *S&P'15*, May 2015, pp. 725–741.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEE S&P'12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394.

[13] S. Clark, S. Frei, M. Blaze, and J. Smith, "Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities," in *ACSAC'10*. New York, NY, USA: ACM, 2010, pp. 251–260.

[14] B. Copos and P. Murthy, "Inputfinder: Reverse engineering closed binaries using hardware performance counters," in *PPREW'15*. New York, NY, USA: ACM, 2015, pp. 2:1–2:12.

[15] DARPA CGC, "Darpa cyber grand challenge binaries," https://github.com/CyberGrandChallenge.

[16] B. Dolan-Gavitt, "Fuzzing with afl is an art," http://moyix.blogspot.nl/2016/07/fuzzing-with-afl-is-an-art.html.

[17] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *IEEE S&P'16*. IEEE Press, 2016.

[18] Elias Bachaalany, "idapython: Interactive disassembler," https://github.com/idapython.

[19] J. Foote, "Cert triage tools," 2013.

[20] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE'09*. IEEE Computer Society, 2009, pp. 474–484.

[21] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Int. workshop on Random testing*. New York, NY, USA: ACM, 2007, pp. 1–1.

[22] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.

[23] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS'08*. Internet Society, 2008.

[24] ——, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[25] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers & Operations Research*, vol. 35, no. 10, pp. 3125–3143, 2008.

[26] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX SEC'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64.

[27] Hex-Rays, "Ida: Interactive disassembler," https://www.hex-rays.com/products/ida/.

[28] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *FSE'15*. New York, NY, USA: ACM, 2015, pp. 782–792.

[29] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *SIGPLAN/SIGOPS VEE '12*. New York, NY, USA: ACM, 2012, pp. 121–132.

[30] H. Kobayashi, B. L. Mark, and W. Turin, *Probability, Random Processes, and Statistical Analysis: Applications to Communications, Signal Processing, Queueing Theory and Mathematical Finance*. Cambridge University Press, Feb. 2012.

[31] 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 *PLDI'05*. New York, NY, USA: ACM, 2005, pp. 190–200.

[32] ——, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[33] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE'07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426.

[34] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing*, vol. 5, no. 3, pp. 315–331, 2005, application Reviews.

[35] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[36] C. Miller and Z. N. J. Peterson, "Analysis of mutation and generation-based fuzzing," 2007. [Online]. Available: https://www.defcon.org/images/defcon-15/dc15-presentations/Miller/Whitepaper/dc-15-miller-WP.pdf

[37] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Sec'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 67–82.

[38] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *CODASPY '15*. New York, NY, USA: ACM, 2015, pp. 87–97.

[39] OpenRCE, "Sulley fuzzing framework," https://github.com/OpenRCE/sulley.

[40] OUSPG, "Radamsa fuzzer," https://github.com/aoh/radamsa.

[41] P. Piwowarski, "A nesting level complexity measure," *SIGPLAN Not.*, vol. 17, no. 9, pp. 44–50, Sep. 1982.

[42] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *EC2ND'10*. IEEE Computer Society, 2010.

[43] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *USENIX Sec'14*. Berkeley, CA, USA: USENIX Association, 2014, pp. 861–875.

[44] K. Serebryany, "Libfuzzer: A library for coverage-guided fuzz testing (within llvm)," at: http://llvm.org/docs/LibFuzzer.html.

[45] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *ACSAC'07*. IEEE, 2007, pp. 477–486.

[46] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking inside the black-box: Capturing data provenance using dynamic instrumentation," in *IPAW'14*. Springer, 2015, pp. 155–167.

[47] 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 *NDSS'16*. Internet Society, 2016, pp. 1–16.

[48] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 1st ed. Norwood, MA, USA: Artech House, Inc., 2008.

[49] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *IEEE S&P'10*. IEEE Computer Society, 2010.

[50] X. Wang, L. Zhang, and P. Tanofsky, "Experience report: How is dynamic symbolic execution different from manual testing? a study on klee," in *ISSTA'15*. New York, NY, USA: ACM, 2015, pp. 199–210.

[51] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS'13*. New York, NY, USA: ACM, 2013, pp. 511–522.

[52] M. Zalewski, "American fuzzy lop," at: http://lcamtuf.coredump.cx/afl/.