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 dataset, 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*) [12], [16], 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 by the software in the early stages of the execution. This makes a *traditional random fuzzer* often ineffective in discovering bugs that are hidden deeper in the execution.

State-of-the-art fuzzers such as AFL [52] employ *evolutionary algorithms* to operate valid input generation. It employs a feedback loop to assess how good an input is: an input is good, if it finds new paths. 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 above questions.

In this direction, the use of symbolic and concolic execution has shown promising results [32], [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 software remains a challenge [9]. For example, Driller was benchmarked with 126 DARPA CGC binaries [14] 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 [16] paper 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 in 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/-concolic 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

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
 ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23404

is much more efficient than existing fuzzers. We evaluated the performance of VUzzer on three different datasets: a) the DARPA CGC binaries [14], 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 [16], a number 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 data sets and show that it is highly effective. To foster further research, we will open source our prototype.

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 [34], [48] in an application. 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 format of input and generate new inputs based on this format. As we shall see, VUzzer is a mutation-based fuzzer.

With respect to input mutation, fuzzers can be *white-box, black-box* and *grey-box*. A white-box fuzzer [20], [21], [25] assumes access to the application's source code—allowing it to perform advanced program analysis to understand better an input's impact on the execution. A black-box fuzzer [1], [38] has no access to program's internals. A middle approach, grey-box fuzzing, employs lightweight program analysis (mainly monitoring), based on the access to binary code of the applications. VUzzer is a grey-box fuzzer.

Another factor that influences input generation is the notion of exploration of the application. A fuzzer is *directed* if it generates inputs to traverse a particular set of execution paths [19]–[21], [25], [37]. 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 [13], [27], [43], [47], [52]. VUzzer is a coverage-based fuzzer.

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

By definition, a coverage-based fuzzer aims at maximizing the code coverage to hit the path that may contain a bug. To maximize code coverage, it tries to generate inputs such that each input (ideally) executes a different path. Any newly generated input that does not result in a new path is useless for the fuzzer. Therefore, it is of paramount importance for a fuzzer to account for the gain from generated inputs, a property that we term *per-input-gain* (PIG). PIG 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 would be effective if it generates inputs with non-zero PIG very frequently. It is not hard to notice that the ability to generate inputs with non-zero PIG addresses two questions, raised earlier in section I- *which offsets and what value to mutate with?* Unfortunately, most of the fuzzers, especially mutation-based ones, are clueless on how to effectively 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 order of CMP
4     .. some useful code ...
5 else { ERROR("Invalid file\n");
6    ...
```

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: 1). AFL has to guess the `FFD8` byte sequence exactly right; 2). 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 likely to explore this new path even if the path leads to an error state. In such case, AFL is trapped 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 may not be the right answer either, because with this new input, AFL again starts mutation randomly. 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 {
6   if(...)   // nested if
7     { ... }
8 }else
9 {
10   ....
11   some code
12   ...
13 }
```

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

Now consider another simple (pseudo) code snippet, in Listing. 2. At line 6, there is another multi-byte $if-condition$ on the input bytes, which is nested in the outer if. As AFL will probably 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, when AFL gets stuck at if at line 6, symbolic execution (like Driller [47]) tries to find new path by sequentially negating path conditions. In this process, it may negate constraints at line 4 to find a new path, which may lead to some error code. AFL, however, has no knowledge of such error code and as a result, it will start exploring that direction. There are several such properties of the code that may hinder the progress of code-coverage based fuzzer.

In order to understand such properties of the code that allow for generating inputs with non-zero PIG, we will walk through a synthetic code snippet that highlights some of the complex constructs (Listing 3). Though 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 inputs, 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) exit (0);
5    fstat(fd,&s);
6    size = s.st_size;
7    if (size > 1000) return -1;
8    read(fd,buf,size);
9    if (buf[1] == 0xEF && buf[0] == 0xFD)// notice the order of CMPs
10        printf("Magic bytes matched!\n");
11   else { ERROR("Invalid file\n"); exit(0); }
12   if (buf[10]==0x25 && buf[11]==0x40) {  // check for %@
13        printf("2nd stop: on the way..\n");
14        if (strncmp(&buf[15],"MAZE",4)==0)   // nested IF condition
15             raise(SIGSEGV); // some bug here
16        else {
17             printf("you just missed me...\n");
18             // do some other task...
19             close(fd); return 0;
20        }
21   } else {
22        ERROR("Invalid bytes");
23        // do some other task...
24        close(fd); return 0;
25   }
26   close(fd); return 0;
27 }
```

Listing 3. Motivating example that illustrate issues in fuzzing

It is interesting to note that state-of-the-art fuzzer AFL [52] could not reach the buggy state after running 24 hrs. What is so special about this toy example and what is missing in fuzzers like AFL? We try to investigate these points by walking through the code given in Listing 3

1) **magic-bytes:** The 2nd and 1st bytes are compared first to validate the input (line 9). If these bytes are not set properly at those offsets, the input is discarded immediately. In this example, what is interesting is that offset 1 is checked first and then offset 0. We see this behavior in real application, like `djpeg` utility. This also explain that it took million 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 keep *guessing* a valid combination of bytes and offsets.
2) **deeper-execution:** In order to go deeper in execution, there is another check at line 12 that compares offsets 10 and 11 (note such offsets may be read from the input and thus may not be fixed for every input to make them 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 15. Again, AFL falls into the loop of guessing valid combination of bytes and offsets. In general, after few iterations of input generation, a large percentage of input will be falling into the *error-handling* code. Any coverage-based fuzzer, like AFL that searches for new basic blocks, is likely to further use such inputs as these input have indeed found new code. However, if we consider the exploration of more meaningful and interesting paths, reusing such input is a trap, which hinders further exploration of the application code.
3) **markers:** In order to reach buggy spot at line 15, there is a branch constraint to satisfy- `MAZE` at line 14. It should be noted that such bytes may not be present at fixed offsets, but rather work as indicators for certain fields in many input formats, such as JPEG, PNG, GIF. Miller & Peterson pointed out in [35] that presence (and absence) of such markers have direct impact on executed code! As often such markers are multi-bytes, AFL has tough time in generating such bytes to execute certain paths.
4) **nested-conditions:** In the context of code-coverage based fuzzing, each path is important. However, reachability of certain paths may be more difficult than others. For example, in order to reach line 15, any input has to satisfy a check at line 14, which is only triggered when constraint at line 12 is satisfied. Therefore, in order to increase the chance of reaching line 15, we need to fuzz any input that reaches line 12 more often. In the case of AFL, even if it passes or fails constraints at line 12, 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 lot of time in mutating input executing *easier path* and thereby minimizing the chance of reaching line 15 as a result of spending less time in satisfying constraint at line 14. Clearly, it is not able to prioritize efforts for *interesting* path. A better strategy would be to optimize efforts based on the *control-flow* characteristics of the application.

Interestingly, in a recently published post (at the time of writing this paper) from one of the authors of LAVA [16], the similar observations (pitfalls) are noted about AFL [15] This supports our observation that black/greybox fuzzers, like AFL tend to be *application agnostic*, which makes them not much effective in discovering *hard-to-reach* bugs.

In general, above mentioned difficulties can be handled by symbolic/concolic based approaches [8], [19], [21], [32], which has been explored in a recent approach called 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 to assist AFL crossing the first hurdle at line 13. However, AFL will continue to fall into the trap at lines 17, 19. Moreover, this combination is again agnostic to nested-condition and thus path prioritization remains an issue. A more general and practical problem is the scalability issue of symbolic execution based solutions. Though in this small motivating example, we could not present a scenario, real-world applications are complex enough to make symbolic execution go into state-explosion state. This is evident from the results, presented in Driller [47]- out of 41 binaries from DARPA CGC, Driller's concolic engine

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

could generate new meaningful inputs only from 13 binaries. In yet another study [50], it was 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 performance of symbolic/concolic execution based approaches on real-world applications is yet to be optimistic.

B. Evolutionary Input Generation

In spite of 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 motivating example, it is almost impossible to generate an input that will reach line 22 in one mutation. This motivate us to follow *evolutionary fuzzing approach* for input generation. Therefore, the overall technique for input generation, employed by VUzzer, is inspired from an evolutionary algorithm- genetic algorithm (GA). In the following paragraph, we very briefly describe main steps that a typical GA follows (see Algorithm 1). In the later sections, we will refer to these steps while building 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 GA 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), which are, then, subjected to 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 thus created 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. In this way, we get new set of inputs which undergo evaluation state (EVALUATE). In this state, we monitor the execution of each new input on the basis of a set of properties. These properties are used in a *fitness function* to access the input's suitability. We choose the input with high fitness score for the next generation of inputs. This whole process continues until a termination condition is met: either the maximum number of generations passed or objective is met (in case of fuzzing, a crash is found).

III. Overview

We now describe our technique to address the challenges, mentioned in the previous section, and in the process, explain the high-level design. Fig. 1 provides an overview of the 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

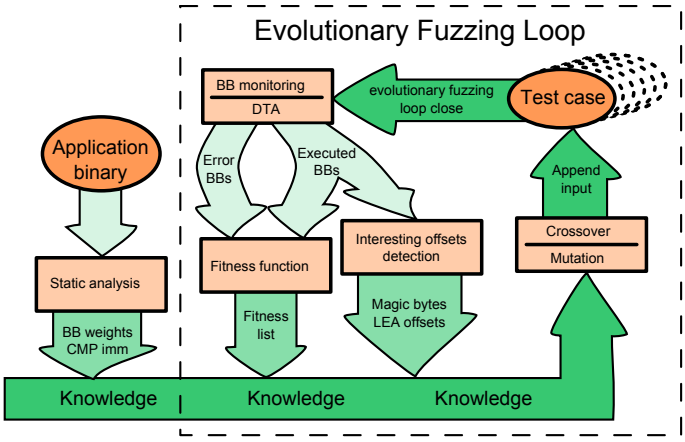

Fig. 1. A high-level view of the functioning of VUzzer. Legends:- BB: basic-block, CMP imm: CMP instruction with one operand as immediate, DTA: dynamic taintflow analysis, LEA: *load effective address* instruction

loop "smart" and help the fuzzer find inputs with non-zero PIG with higher frequency.

A. Features

The two main components 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. Fig. 1 shows that VUzzer continuously pumps this information back into the evolutionary mutation and crossover operators to help it generate better inputs in the next generation. We first introduce the features and then discuss the static analyzer and main fuzzing loop.

Dataflow features provide information about the relationship of input data with the computation in the application. VUzzer extracts them using standard 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, it can determine which offsets are interesting to mutate and what values to use at those offsets (providing partial answers to the question of Section I). It 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, it monitors the `lea` instruction to check if the `index` operand is tainted. If so, it infers that the value at that offset is `int` and mutates the input accordingly. Besides these two simple, but powerful features, many others are possible.

Control-flow features allow the fuzzer to infer the *importance* of certain execution paths. For example, Fig. 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

later in this paper. For now, we assume that we heuristically identify the basic blocks of 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 it is not nested. We use control-flow features to *deprioritize* and *prioritize* paths. As enumerating all possible paths in the application is infeasible, we implement this metric by assigning *weights* to individual basic block. Specifically, basic blocks that are part of *error-handling* get a negative weight, while basic blocks in *hard-to-reach* code regions obtain a higher weight.

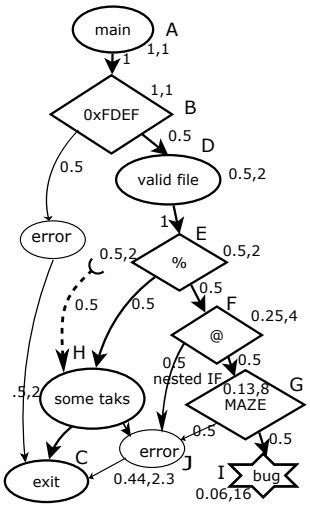

Fig. 2. A high-level CFG of the code shown in Listing 3. The probability at each edge denotes the probability of this branch outcome. The probability at the node indicates the total probability of reaching this node. So if all edge probabilities are 0.5 and the program can reach a node N_2 either from 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 weight.

Fig. 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- and data-flow features (Sec. 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 light-weight intraprocedural static analysis to 1) obtain `immediate` values of the `cmp` instructions by scanning the binary code of the application and 2) compute the weights for basic blocks from the binary of the application.

The presence of many of such immediate values of the `cmp` instructions in the application's code typically indicates that it *expects* input to have 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`}. For weights of basic blocks, 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 calculare the weight w_b of basic block b as $1/p_b$. Thus, the lower the probability of reaching a basic block, the higher will be its weight. Using this model, the probability and weight of each basic block is shown next to each basic block in Fig. 2 (see Section IV-A3). We can see that the probability of reaching basic block G is less than that for 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 explain the main fuzzing loop by means of the steps in Algorithm 1. Before the main loop starts, we execute the application with inputs from SI to infer an initial set control- and data-flow features. For all inputs in the set of seed inputs 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 of 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 of 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 dataflow-only 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 block using the weights list L_{BB}. Basic block that belong to error-handling code gets 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 such 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 resp. s.t. $p_1 = A \rightarrow B \rightarrow D \rightarrow E \rightarrow H \rightarrow J$ and $p_2 = A \rightarrow B \rightarrow D \rightarrow E \rightarrow F \rightarrow J$. For simplicity, that error-handling basic block J gets a weight -1 and that the frequency of execution of each basic block is 1. Using the weights from Fig. 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). Thus 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 technique, encompassing the SELECT, RECOMBINE, and MUTATE steps of Algo. 1. Together, these substeps are responsible for generating *interesting* inputs. In every 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 *ROOT*.

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 length. Similarly, various offsets from current inputs parents are picked for mutation. Thus, if any magic bytes exist, they will be replaced at the proper offset 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 threshold number of generations.

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

TABLE I. GLOBAL SYMBOLS AND THEIR MEANING.

Symbol	Description
DTA	dynamic taintflow analysis
SI	set of seed inputs (valid inputs).
L_{BB}	list of weights of basic block, obtained by static analysis of the application's binary.
L_{imm}	list of immediate values of cmp instructions, obtained by static analysis of the application's binary.
L_{fit}	Sorted lists (in decreasing order) of fitness scores of inputs, obtained in 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 $index$ operand of lea instruction.

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

IV. DESIGN AND IMPLEMENTATION

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

A. Design Details

1) Dynamic TaintFlow Analysis (DTA): DTA is the heart of VUzzer as it plays major role in *evolving* new inputs. This is also the technique that makes VUzzer different from other existing fuzzers. DTA is used to monitor the flow of *tainted* input (e.g., network packets, file) within the application. DTA can answer the queries, like during program execution, which memory locations, 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 an executed CMP instruction `CMP op1, op2` (`op1` and `op2` can either be register, memory or immediate), DTA determines if `op1` and/or `op1` are tainted by a set of offsets. DTA is able to track taint at 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 by T^i_j the set of offsets that taint j^{th} byte of i^{th} operand of a given cmp. We also record 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 set of values of untainted operand of cmp. For `lea` instruction, DTA tracks only the $index$ register. L_{lea} contains all of the offsets that taint such $indices$.

2) Magic-Byte Detection: Based on our understanding of file formats that have magic-bytes, we postulate that *magic-byte is a **fixed** sequence of bytes at a **fixed** offset in the input string*. We have verified this postulation on several file formats that have magic-bytes, for example, jpeg, gif, pdf, elf, ppm. As VUzzer assumes the availability of few valid inputs for a given application, we leverage results of DTA on these inputs at the beginning of fuzzing. As application expect input to contain magic-byte, DTA's result on `cmp` instructions will have corresponding check for magic-byte. The code from Listing 3 expects a magic-byte `0xFDEF` in the beginning of the input file. Thus, 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 executions corresponding to each of these valid inputs. Conversely, if for a set of valid inputs, we get $cmp_i = (o_i, v_i)$ present in DTA's result for all the inputs, v_i is a part of magic-byte at offset o_i. It can be noted that this algorithm of detecting magic-bytes has false positive. This may happen if all of the initial valid inputs share identical values at the same offsets. However, this will still be useful for generating inputs to go beyond the very initial check for magic-bytes with a reduced possibility 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, if the corresponding $value$ depends on multiple offsets per byte, we do not consider such offsets to be candidate for magic-byte. For example, for a given cmp, 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 value of 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) sequence of bytes. We denote by O_{other} a set of all such offsets.

3) Basic-block Weight Calculation: From a coverage-based fuzzing perspective, every feasible path is important to traverse. A general strategy is to make equal effort to generate inputs for all feasible paths. However, due to the presence of control structures, reachability of some path may not be the same as that of others. This situation arises much frequently, if we have *nested* control structure [40]. Therefore, any input that exercises such *hard-to-reach* should be rewarded more as compared to other inputs. We calculate this reward by assigning higher weights to basic blocks that are contained within nested control structures. As enumerating all the paths at interprocedural level is too expensive and may not be even scalable, we adhere our analysis to be at intraprocedural level, i.e., we calculate weights for each basic block in a single function. Later, we gather and add the weights of all basic blocks that falls on a path that is executed by an input. In this way, we *simulate* the score of an interprocedural path by stitching several intraprocedural path scores.

If we consider the transition of an input at a particular

basic block in the control flow graph to the next basic block is dependent on some probability, we can develop a probabilistic model called Markov process for input behavior from the CFG. A Markov process is stochastic process in which the outcome of a given trial depends only on the current state of the process [29]. We model the CFG of a function as markov process with each basic block having a probability based on its *connections* with other basic block. For a given basic block, we assign equal probability to all its *outgoing edges*. Thus, 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 probability associated with each basic block in the CFG. The root basic block of CFG is initialized with a probability of 1. Loops are handled by assigning a fixed probability 1 to each backedge, thereby neglecting the effect of backedge (i.e. we flatten the loop to speed up fixed point calculation). From the Eqn. 1, the weight of a basic block b is given by:

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

4) Error-Handling Code Detection: As noted earlier, during fuzzing process, a majority of mutated inputs will be executing a path that will end up in some error state. Deprioritizing such execution paths is a key step towards increasing the chances of creating interesting inputs faster. The 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 block 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 $Valid_{BB}$ denotes 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 termed as error-handling basic block (i.e., belongs 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 at work is that since SI is a set of valid inputs, no error-handling code will be invoked. Therefore, $Valid_{BB}$ will have *only* basic blocks that correspond to valid path. And since TR is a set of totally random inputs, they will be caught by error-handling code during execution. This is a very conservative detection of error-handling basic block as we may miss few basic block if certain inputs are caught by different error-handing code, but we will not classify a basic block, corresponding to valid path as error-handling basic block. More formally, let

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

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

Incremental Analysis: we observe that since our error-detection is based on the dynamic behavior of the application, not all error-handling code may be hit during *initial analysis*. As inputs evolve, they explore more paths and thus encounter new error-handling code. We initiate *incremental analysis* during later iterations of fuzzing. In our test experimental setup, we observe that as we proceed with more iterations of fuzzing, number of new error-handling code decreases, which should be a general phenomenon as software have finite number of error-handling code, which are reused in different parts of the application. Therefore, the rate of running this analysis *decreases* as we execute more iterations. The heuristic behind *incremental analysis* is the observation that as fuzzing proceeds, a *majority* of newly generated inputs will end-up in 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) experimentation shows that $n = 90$ is a reasonable choice. Let $BB(I)$ is the set of all the executed basic blocks by inputs in I. We classify a basic block b from $BB(I)$ as error-handling basic block if it is present in at least $n\%$ of inputs from I and it is not in $Valid_{BB}$ 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 Block: After detecting EHB, we want to deprioritize paths that contain these error-handling basic blocks. We achieve this by *penalizing* inputs so that such inputs have less chance of participating in next generation of inputs. Therefore, basic block in EHB is given a negative weight, which impacts the fitness score of inputs (see next section IV-A5). We can simply give a weight of -1 to such basic blocks, but as EHBs are in minority when compared to total basic blocks executed by an input, such a small quantity will have negligible effect. We solve this problem by defining a *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, it says that once an input enters a error-handling code, the effect (contribution) of may of its basic blocks in calculating fitness score is reduced by a factor μ. For a given input i, we use the following formula for weight calculation.

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

where, $|BB(i)|$ is number of all executed basic blocks by input i, $|EHB(i)|$ is the number of all 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 the basis of applying 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 depends on its fitness score. VUzzer assess the fitness of an input in two ways. If the execution of an input results in hitting a new basic block, which is not from EHB set, the input qualifies for participation for the next generation. This is similar to AFL (however, note the difference by considering the use of EHB set). However, this measure of fitness considers all

newly discovered paths equal. In previous sections, we have mentioned our views against this monolithic consideration. The important (and thus the fitness) of an inputs depends on the path that it executes and which, in turn, depends on the weights of the basic blocks on that path. Therefore, we define fitness f_i of an input i as function that captures the effect of basic block's weights.

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

where $BB(i)$ is the set of basic-blocks executed by input i; $Freq(b)$ is the execution frequency of basic-block b by i; W_b is the weight of basic-block b (by using Eqn. 2); l_i is the length of input i and MAX_LEN is a pre-configured limit on input length. This is used to address the phenomenon of input *bloating*. In the parlance of genetic algorithm both of these fitness criteria 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: Input generation in VUzzer consists of two parts- crossover and mutation, which are not mutually exclusive, i.e. crossover is followed by mutation with a fixed probability.

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

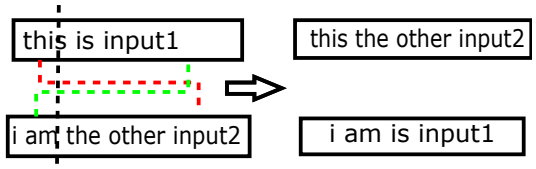

Fig. 3. Crossover operation in VUzzer: In this single-point crossover, we select 5^{th} offset as cut-point. For the first parent `this is input1`, this breaks it in two parts- i_1^1=`this` and i_1^2= `is input1`. For second parent, we again get two parts- i_2^1=`i am` and i_2^2= `the other input2`.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

b) mutation: is more complex operation. It involves several sub-operations to change the given input (parent) to yield child input. The given (parent) input undergoes several steps in the following order to generate a new input.

step 1 randomly select tainted offsets from the set O_{other} and insert strings at those offsets. These strings are formed by bytes obtained from the set L_{imm}.

step 2 randomly select offsets from the set L_{lea} and mutate these offsets in the string from step 1 by replacing them with *interesting* integer values,like, 0, MAX_UINT, negative numbers.

step 3 For all the tainted `cmp` instructions for the parent input (input that is being mutated), if the values of $op1 \neq op2$, replace the value at tainted offset in string from step 2 by 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 corresponding offset as determined by magic-byte detector.

B. Implementation Details

The main 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 BitVector representation for date[3].As VUzzer consists of two main components- static and dynamic analysis, we implement them as per the details given below.

Static Analysis: VUzzer implements both of the static analyses (constant string extraction and basic block weight calculation) within IDA [26]. The analysis is written in python using IDAPython [17].

Dynamic Analysis: VUzzer implements dynamic analysis for two purposes- basic-block monitoring to get execution trace and taintflow analysis. Both of these analyses are build on the top of dynamic analysis framework Pin [30]. For basic block monitoring, we implemented a pintool to record each basic block, along with its frequency of the given process. This pintool can selectively monitor basic blocks, executed by certain libraries on-demand. Selective library monitoring allows us to reduce the execution trace and to focus only on the intended code.

Our implementation of dynamic taintflow analysis is based on the technique DataTracker, proposed by Stamatogiannakis *et.al* [46], which in turn is based on LibDFT [28]. As LibDFT can only handle 32-bit applications, VUzzer is used to fuzz only 32-bit applications, which, by no means, is a limitation of the technique, presented in the paper. However, we made several changes to DataTracker to make it suitable for our purpose.

- 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 this tuple is 64 bit long (32 bit for `ufd` and 32 bit for `file_offset`). For each memory location it has a set of these tuples associated with it which determines the offsets and the files by which this memory location is tainted. We changed this to a EWAHBoolArray type[4] which is compressed bitset data type. Since we only need taintflow from one 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 ony contains offsets. As a result, our implementation is at least 2X faster and uses several times less memory than DataTracker.

[3]https://pypi.python.org/pypi/BitVector/3.4.4
[4]https://github.com/lemire/EWAHBoolArray

- We added instrumentation callbacks for `cmp` family of instructions like `CMP`, `CMPSW`, `CMPSB`, `CMPSL` and `lea` instruction to catch byte level taint information for the operands involved in the computation.
- We rewrote hooks for each implemented system call and also added hooks for some extra system calls such as pread64, dup2, dup3, mmap2, etc. For comparing our fuzzer performance on DARPA dataset [14] we also implemented hooks for DECREE based system calls, which are different from normal Linux syscalls.

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 same bug but different instance). In order to determine uniqueness of crash, VUzzer uses a variant of *stack hash*, proposed by Molnar *et.al.* [36]. In our pintool, we implement a ring buffer that keeps track of last 5 function calls and 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 compared the newly generated hash with the older ones to determine if the reported crash is an unseen one.

V. Evaluation

In order to measure the effectiveness of our proposed fuzzing technique, in this section, we present the result of experimentation with VUzzer. To expose VUzzer to a variety of applications, we choose to test VUzzer on three different sets of applications A. DARPA CGC binaries [14], B. miscellaneous applications with binary inout format as used in [42] and C. a set of *buggy* binaries recently proposed in LAVA [16]. All of the experimentations are run on Ubuntu 14.04 LTS, 32-bit, 2-core, 4 GB RAM, except DARPA CGC as the whole environment is provided as a VM running a customized OS called DECREE. We want to emphasize that our main objective of evaluating VUzzer is to show its effectiveness in identifying bugs (that may be buried deeper in the execution) in much lesser number of generated inputs. Our implemented prototype of VUzzer is not as optimized for faster input execution as AFL and therefore, we do not seek any comparison in this direction.

In the current version of VUzzer, we performed all the experimentations on 32-bit applications. This is due to the limitation of LIBDFT [28] that we use for taintflow analysis. However, we are working on a taintflow analysis implementation for 64-bit applications and VUzzer will be open sourced with 32-and 64-bit versions. The status will be updated on https://www.vusec.net/projects/fuzzing/.

A. DARPA CGC Dataset

As a 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 type of bugs inserted 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 chose an action, including the option to *quit*. Further in many cases, there are multiple menus (in a different state of the program) with different options to quit. As VUzzer involves a step of generating totally random inputs (error-handling code detection, section IV-A4), executing such inputs puts the application in a loop, looking for *valid* option, including the option to quit. This makes the application to run forever. This problem is an interfacing problem and it is not fundamental to the fuzzing method.
- Some of these binaries are compiled with *floating point instructions*, which are not handled by LibDFT and thus VUzzer does not get correct taintflow information.
- As VUzzer is based on Pintool [31], we followed the given procedure to run pintool in DECREE[5]. However, for few of the binaries, we could not run them with pintool.
- Few of the binaries involve interaction with other binaries, which is not handled by VUzzer.

After considering the above mention obstacles, 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 fuzzer[6]. It has the same fuzz engine as that of AFL, but a different mechanism to get execution trace. Our choice of using AFLPIN over AFL is to have an identical interfacing mechanism with the SUT, i.e. passing input to pintool via file descriptor 0 (`STDIN`). VUzzer found crashes in 29 of these binaries, whereas AFLPIN found 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 crash. The most important result was the number of executed inputs per crash in both of these fuzzers. We ran both the fuzzers for a maximum of 6 hours. In Fig. 4, we see plot for number of executions for the cases where both of the fuzzers found crashes (total 13).

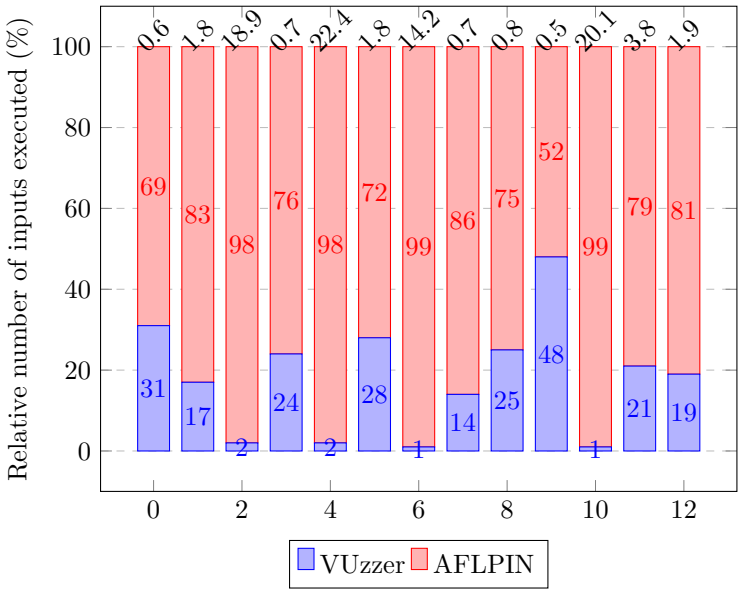

Fig. 4. Relative number of inputs executed for each CGC Binaries, wherein both VUzzer and AFLPIN find crash. The numbers above the bars are the total number of inputs (in thousands) executed.

While fuzzing a specific binary `NRFIN_00015`, we observed the importance of computing fitness score f_i in a very

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

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

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

We note that results on this dataset is not very encouraging, specially in the view of results, reported in Driller [47]. We further investigate the results and found some issues that may interfere with the performance of VUzzer on CGC.

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

In the view of aforementioned points, we opine that VUzzer is not suitable for interactive program, mainly because ot its 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 few linux utilities [16] for testing fuzzer and symbolic execution based solution. We use a part of the dataset, referred to as **LAVA-M** in [16] for evaluating VUzzer. This set consists of 4 linux utilities- `base64, who, uniq` and `md5sum`, each injected with multiple faults (in the same binary per utility). The paper reports results of evaluating a coverage-based fuzzer (FUZZER) and symbolic execution and SAT (SES) based approach on these buggy applications. For the benefit of readers, we restate the results from the original paper in Table II. The last column of Table II shows results with VUzzer. The number shown are the total unique bugs identified by VUzzer. In the case of `md5sum`, we could not run VUzzer as it was crashing on the first round of input generation and thus not allowing program to parse more of any input. Each injected fault in LAVA binaries has an ID and this ID is printed on the shell before binary crashes due to that fault. This allows us to precisely identify faults triggered by VUzzer. Table III provides IDs of the faults, triggered by VUzzer per LAVA binary.

For LAVA dataset, we highlight certain interesting points. Most of the LAVA injected faults are based on the *artificially* injected path conditions, like `lava` to reach a particular path and trigger the bug. This is very well captured by, thanks to its dataflow features, VUzzer. For example, during `base64` fuzzing, we learned that first four bytes should be either of `'val` and `lav'` to follow a particular path. Similarly, we discovered that few last bytes can be any of the following to take different paths: `las[,lat\x1b,Wsal` etc. It should be noted that most of the path constraint, injected by LAVA, are multibytes. Such constraints pose a serious problem for AFL to traverse deeper in the execution (as also noted in [15]). Another interesting point is the performance of VUzzer on `who`. The fuzzer used in LAVA study failed to find even a single bug, whereas VUzzer found several unique crashes.

TABLE II. LAVA-M CORPUS: PERFORMANCE OF VUZZER W.R.T. OTHER 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)

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

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

In all, on both of the *artificial* datasets, we showed very encouraging and *not-so-encouraging* results. Without going into the merits and demerits of VUzzer w.r.t. these datasets, we move further to evaluate VUzzer on real-world programs that have also been analyzed by other fuzzers.

C. Various Applications (VA) Dataset

We use a set of the real programs (`djpeg/eog`, `tcpdump`, `tcptrace`, `pdf2svg`, `mpg321`, `gif2png`) to evaluate the performance of VUzzer. Rebert *et al.* also evaluated these programs to report several bugs [42] and therefore, we included these program in our study. For each of these programs, we use *vanilla release* with Ubuntu 14.04. We would like to mention that through these utilities, we also targeted some well known libraries, like `libpcap, libjpeg, libpoppler` and `libpng`. Each applications is fuzzed not for more than 24 hrs. 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 VA dataset.

In Figure 5, we show 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. We can see that for almost every application, VUzzer keeps finding crashes during the later iterations of fuzzing, whereas AFL gets exhausted after few initial iterations. This may be due the

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

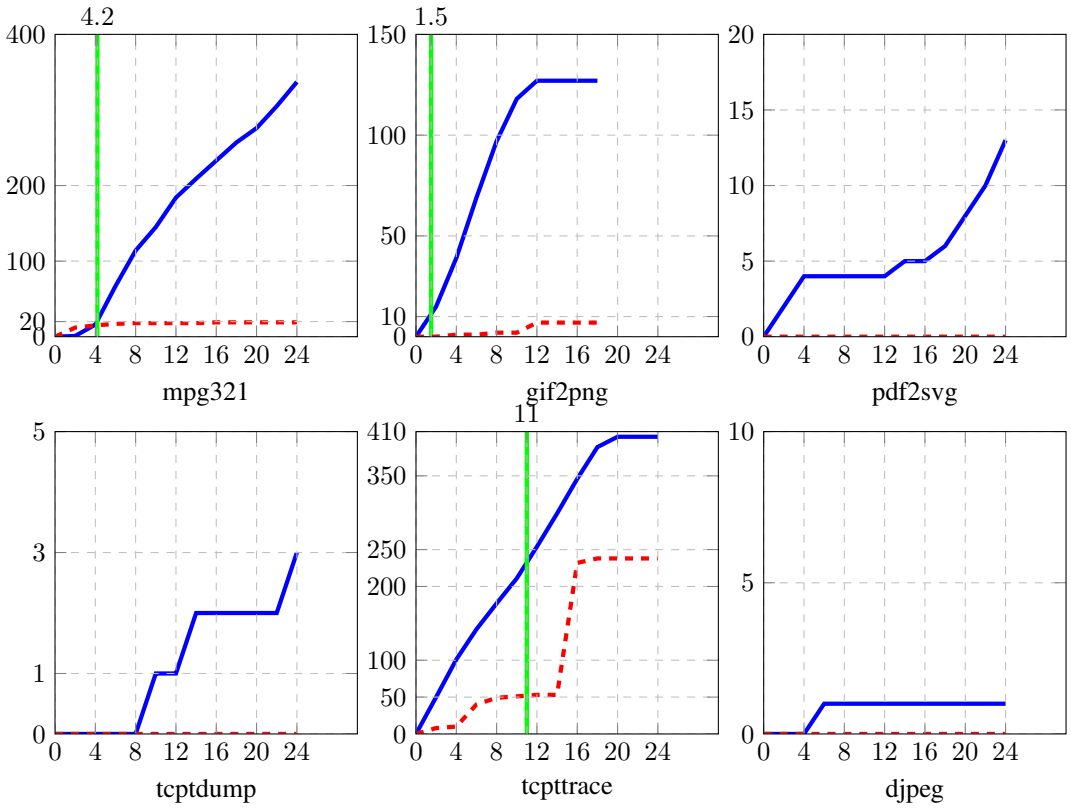

Fig. 5. Distribution of crashes over a time period of 24 hrs. X-axis: cummulative sum of creahes. Y-axis: time (over 24 hrs). Blue line: VUzzer; Red dashed line: AFL. Vertical green line: Time taken by VUzzer to find crashes equal to those found by AFL during a complete run.

TABLE IV. RESUTLS OF RUNNING VUZZER AND AFL ON VA DATASET.

Application	VUzzer		AFL	
	Unique crash	# of Exec.	Unique crash	# of Exec.
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

cause that at later stages, AFL is not able to find new (deeper) paths, whereas VUzzer is able to learn branch constraints as it explore new paths and thus it is able to find crashes in later stage of fuzzing. Another

D. Crash-Triage Analysis

Fuzzers tend to generate a large number of crashes. Fixing every bug associated with crashes 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 effort of patching bugs is invested in the bugs that are critical.

!Exploitable [18], 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 (if it is a read or write) and signals triggered by the application. While this analysis is simple and fast, it is not sound but still provides hints on the severity of a crash. We use the !Exploitable tool to rank the crashes found by VUzzer on this dataset. The results are shown in Table V as percentages.

TABLE V. PERCENTAGES OF EXPLOITABLE BUGS DISCOVERED BY VUZZER 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

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

With !Exploitable, we obtained the same results on a 64 bit system running the same version of the tested applications. This shows that even if VUzzer was not tested on a 64 bit environment its results are meaningful across architectures. This is an interesting observation for fuzz testing in general.

To further analyze the quality of the bugs discovered by VUzzer, we measure the distance between the crash and the library involved. 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 adhere to 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 of last library call, when a crash in observed in the application code.

The distance from a crash to 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 effect) which caused the crash might be processed in a library call. Second, we count the number of stack frames from a library call to crash. For example, libraries that are using output function hooks that reside in the main application (e.g. `tcpdump`, `tcptrace`, `mpg321`) are covered by this heuristics. Table VI shows the results of the analysis.

TABLE VI. DISTANCE FROM THE CRASH TO THE LIBRARY CALLS.

	#instructions	binaries (#stack frames)
gif2png	20554.00	gif2png (0); libc (5)
mpg321	733.04	**libid3tag**, vdso (0); libmad (3.1); libc (3.9); mpg321 (5.5)
pdf2svg	626.11	vdso (0); 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 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 errors. `gif2png` crashes always inside the application. This is confirmed by both metrics, the stack frame distance is high as well the number of instructions. The `pdf2svg` crashes the majority of time in `libpoppler`. The stack frame distance is 3 because the signal gets *routed* from `vdso` through the standard library. `tcpdump` and `tcptrace` are using the same library but because `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 correspond to *zero-day* vulnerabilities and we are working in reporting these bugs 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 (introduction and overview), we have already highlighted some of the major differences w.r.t. ate-of-the-art fuzzers like AFL. In this section, we survey over the recent research work in the direction of fuzzing. This enable 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 to input generation is a well explored area in software engineering [7], [33]. There have been attempts to use genetic algorithms for input generation to discover vulnerabilities in applications [24], [41], [45]. The difference lies in the fact that these approaches assume the knowledge of application *a priori* to focus on the path that leads to vulnerable part of the program, which makes them more closer to *directed* fuzzing approaches and therefore, our technique deviates from them substantially. As in the case of AFL, feedback loop in these approaches does not try to relate application behavior with the input structure to enhance input generation.

B. Whitebox Fuzzing Approaches

Whitebox fuzzing approaches are 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 interesting approaches to make fuzzing more efficient, for example, by applying symbolic execution and taintflow analysis to solve branch constraints [19]–[23], [25]. Though VUzzer differs from these approaches in a number of ways, the main fundamental difference remains the application of symbolic execution. Similar to VUzzer, BuzzFuzz, proposed by Ganesh *et.al.* [19] makes use of taintflow analysis, but entirely for a different purpose. It is directed fuzzer and therefore, it does not try to learn every path constraints. It rather uses taintflow to detect bytes that influence *dangerous spots* in the code, like library calls arguments and mutate those bytes in the input to trigger exceptional behavior. Most of these approaches 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 [38] and Radamsa [39] have discovered bugs in real-world applications. However, throughout the paper, we have made out point about the limitations of such fuzzers.

Recently, symbolic and concolic execution based fuzzing approaches have dominated the area of *smart* fuzzing [11], [37], [47], [51]. Mayhem [11], a system from CMU to find automatically exploitable bugs in binary code, uses several program analysis techniques, including symbolic and concolic execution to reason about application behavior for a given input, which is, in principle, similar to the objective of VUzzer. However, as the goal of VUzzer differs from that of Mayhem, VUzzer does not require heavy-weight program analysis techniques and instead *infers* important properties of the input just by applying heuristics based on light-weight analyses. Similarly, Driller [47] uses hybrid concolic execution technique([32]) to assist fuzzer by solving branch constraints for deeper path explorations. In [27], Kargén *et.al.* proposed an interesting approach to generate fuzzed inputs. For a given application that is being tested, the technique modifies other *input producer* application by injecting faults that influence the output. In this way, 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 taintflow analysis to *infer* checksum-handling code, which further assists fuzzer to execute application by bypassing this

TABLE VII. REPORTED BUGS FOUND BY VUZZER.

Program	Error Type	Already fixed?	Bug Reported
tcpdump	Out-of-bound Read	Yes	No
mpg321	Out-of-bound 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-bound Read	No	Yes [5]
gif2png	Out-of-bound Read	No	Yes [6]

check. VUzzer can definitely benefit by using this technique while fuzzing.

There have been other several techniques to enhance the fuzzing [10], [42], [51]. VUzzer can benefit from these approaches, multiple ways. For example, Seed selection [42] can help VUzzer to start with a *good* set of seed inputs.

VII. CONCLUSIONS

This paper takes the position that fuzzing is a light-weight *scalable* software testing technique to find bugs and applying *heavy-weight* and *non-scalable* techniques, like symbolic execution based approaches, may not be the only way to improve the performance of a coverage-based fuzzer. After studying several existing general-purpose (black/graybox) fuzzers, including *state-of-the-art* AFL fuzzer, we note that they tend to be *application agnostic*, which makes them less effective in discovering deep rooted bugs. The downside of being application agnostic is the 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 centric features allows us to *prioritize* and *deprioritize* certain paths, thereby making inputs generation a controlled process. We achieve this by assigning weights to basic blocks and make fitness of the input dependent of this. By using dynamic taintflow analysis, we monitor several dataflow features of the application, which provides us an ability to infer *type* system of the input. For example, it provides information on which offsets in the inputs are used at several branch conditions, what values are used as branch constraints etc. We use these interesting properties of the input in our feedback loop to generate new inputs. We have implemented the technique in a prototype, called VUzzer and evaluated it on several applications. In this process, we also compared it with a *state-of-the-art* fuzzer AFL to highlight its performance. In almost every test case, VUzzer was able to find bugs within an order of magnitude less inputs as compared to AFL.

We find it a very interesting direction to infer input properties by analyzing application behavior and use these properties to enhance fuzzing process. We see enough opportunities to further enhance the technique proposed in this paper by considering more advance approaches, like Howard [44] to *infer* more properties of the input format, thereby making fuzzing *great* again!

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] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proc. of the 13th ACM CCS'06*. ACM, 2006, pp. 322–335.

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

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

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

[12] 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 *Proc. ACSAC '10*. New York, NY, USA: ACM, 2010, pp. 251–260.

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

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

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

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

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

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

[19] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. of the IEEE 31st Int. Conf. on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[20] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proc. of the 2nd int. workshop on Random testing*. New York, NY, USA: ACM, 2007, pp. 1–1.

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

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

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

[24] 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.

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

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

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

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

[29] 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.

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

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

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

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

[34] 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.

[35] 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

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

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

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

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

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

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

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

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

[44] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proc. NDSS'11*. Internet Society, 2011.

[45] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *Proc. ACSAC*. 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 *Proc. IPAW 2014*. 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 *Proc. NDSS'16*. Internet Society, 2016.

[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 *Proceedings of the 2010 IEEE Symposium S&P 2010*. 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 *Proc.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 *Proc. CCS'13*. New York, NY, USA: ACM, 2013, pp. 511–522.

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