The Journal of Systems and Software 85 (2012) 102–111

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

SimFuzz: Test case similarity directed deep fuzzing

Dazhi Zhang, Donggang Liu*, Yu Lei, David Kung, Christoph Csallner, Nathaniel Nystrom, Wenhua Wang

Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, United States

ARTICLE INFO

Article history:
Received 25 August 2010
Received in revised form 4 June 2011
Accepted 13 July 2011
Available online 4 August 2011

Keywords:
Fuzzing
Software testing
Software vulnerability

ABSTRACT

Fuzzing is widely used to detect software vulnerabilities. Blackbox fuzzing does not require program source code. It mutates well-formed inputs to produce new ones. However, these new inputs usually do not exercise deep program semantics since the possibility that they can satisfy the conditions of a deep program state is low. As a result, blackbox fuzzing is often limited to identify vulnerabilities in input validation components of a program. Domain knowledge such as input specifications can be used to mitigate these limitations. However, it is often expensive to obtain such knowledge in practice. Whitebox fuzzing employs heavy analysis techniques, i.e., dynamic symbolic execution, to systematically generate test inputs and explore as many paths as possible. It is powerful to explore new program branches so as to identify more vulnerabilities. However, it has fundamental challenges such as unsolvable constraints and is difficult to scale to large programs due to path explosion. This paper proposes a novel fuzzing approach that aims to produce test inputs to explore deep program semantics effectively and efficiently. The fuzzing process comprises two stages. At the first stage, a traditional blackbox fuzzing approach is applied for test data generation. This process is guided by a novel *test case similarity* metric. At the second stage, a subset of the test inputs generated at the first stage is selected based on the test case similarity metric. Then, combination testing is applied on these selected test inputs to further generate new inputs. As a result, less redundant test inputs, i.e., inputs that just explore shallow program paths, are created at the first stage, and more distinct test inputs, i.e., inputs that explore deep program paths, are produced at the second stage. A prototype tool `SimFuzz` is developed and evaluated on real programs, and the experimental results are promising.

1. Introduction

Software security breaches cost billions of dollars each year and can cause potentially devastating impact on individuals, organizations, and government agencies. Many approaches have been proposed recently to detect or prevent vulnerabilities in programs (Wagner et al., 2000; Godefroid et al., 2008; Cadar et al., 2006, 2008; Cowan et al., 1998; Newsome and Song, 2005; Bhatkar et al., 2003). Security testing can pro-actively detect program vulnerabilities and has become an attractive research area (Ganesh et al., 2009; Godefroid et al., 2008).

Fuzz testing (or fuzzing) was first proposed in Miller et al. (1990) to test the robustness of UNIX utilities. Currently, it is a primary security testing approach due to its cost-effectiveness, and many tools have been developed (Miller et al., 1990; Sutton et al., 2007; Ganesh et al., 2009; Godefroid et al., 2008). The basic idea of fuzzing is quite simple. Generally, certain program inputs are mutated to produce new ones to explore different paths of the program. Here, the mutation means to modify the inputs. Then, the program is executed under these input mutations and the runtime behavior is monitored. If an exception such as a *segmentation fault* is captured, a potential vulnerability is detected. In general, there are two types of fuzzing approaches: blackbox fuzzing and whitebox fuzzing.

Blackbox fuzzing does not require program source code. It randomly or systematically mutates well-formed inputs to generate new inputs and feeds them to the program for testing. It can be remarkably effective for finding vulnerabilities in input validation components of a program since many programs fail to check input formats properly. However, it is hard for blackbox fuzzing to extensively test deep program paths or semantics. The reason is that most of the resulting input mutations are ill-formed. They cannot satisfy required program conditions to reach deep program states. They usually repetitively explore shallow program paths and are rejected by the program. Domain knowledge such as input grammars can be used to direct the fuzzing process and mitigate this limitation (Röning et al., 2002; Sparks et al., 2007). However, it is

* Corresponding author.
E-mail addresses: dazhi.zhang@mavs.uta.edu (D. Zhang), dliu@cse.uta.edu (D. Liu), ylei@cse.uta.edu (Y. Lei), kung@cse.uta.edu (D. Kung), csallner@cse.uta.edu (C. Csallner), nystrom@cse.uta.edu (N. Nystrom), wenhua.wang@mavs.uta.edu (W. Wang).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.07.028

often extraordinarily expensive to obtain proper input grammars in practice due to the availability, diversity, and complexity of input formats (Sutton et al., 2007).

Whitebox fuzzing employs dynamic symbolic execution to systematically test every possible path of a program. At a high level, the program under test is executed both concretely and symbolically. During the execution, every exercised condition (i.e., a predicate) is collected. All these conditions are conjuncted together to form a *path condition* that specifies the condition that must be satisfied to explore this path. Next, certain predicates of the path condition are negated to form a new path condition, and a constraint solver is employed to solve the new path condition and the solution is used to construct the new input. In this way, theoretically every path of the program can eventually be tested. Several dynamic symbolic execution tools have been developed (Godefroid et al., 2005, 2008; Sen et al., 2005; Cadar et al., 2006, 2008), and other techniques such as taint analysis have also been applied during this process (Ganesh et al., 2009). Whitebox fuzzing is powerful for exploring new program branches. However, many branch conditions in real programs contain unsolvable constraints that are beyond the theory of constraint solvers, such as system calls. Complex data structures of the program such as symbolic pointers are also hard to reason about during dynamic symbolic execution. Thus, many program paths can be missed. In addition, it does not scale well to large programs due to the path explosion problem, and it takes a long time to perform the symbolic execution and solve the constraints (Godefroid et al., 2008).

In summary, blackbox fuzzing is simple to conduct but has difficulties when testing deep program semantics without input specification. Whitebox fuzzing is good for generating an input to test a specific path, but still has fundamental challenges in practice.

In this paper, we propose a fuzzing approach that uses lightweight analysis to extensively test deep program semantics *effectively* and *efficiently*. In particular, we propose a novel metric, *test case similarity*, to model the semantic similarity between test cases, and use this metric to guide a two-stage fuzzing process to achieve our goal. At the first stage, this metric is used to avoid blind mutation of inputs. At the second stage, this metric is used to select proper test cases to further test deep program semantics.

Our approach can increase test diversity because we can explore diverse paths near the path explored by well-formed inputs. We agree that a path explored by a produced test case can already have been explored by another well-formed input or test case. We expect that most (or many) test cases generated by our approach will not explore paths explored by well-formed inputs, which is actually confirmed by our experimental results. In addition, it is almost impossible to explore all feasible paths, and the feasible paths are not all equally important. As a result, we focus on exploring deep paths near those that are exercised by well-formed inputs. We believe that our approach is a valuable addition to complement existing techniques such as blackbox fuzzing which have already been effective for testing diverse shallow paths.

Our fuzzing approach integrates techniques from blackbox fuzzing, code analysis, and combination testing to effectively and efficiently test deep program semantics. We have implemented a tool called `SimFuzz` and conducted experiments on real open-source programs to evaluate our approach, and the experimental result is promising.

The rest of this paper is organized as follows. The next section explains some important definitions in the paper. Section 3 presents the main techniques. Section 4 describes the system design and implementation. Section 5 presents the experimental results. Section 6 discusses some limitations and possible counter measures. Section 7 reviews other related work. Section 8 concludes this paper and points out some future directions.

2. Definitions

2.1. Test case similarity

At a high level, a program specifies different actions upon different inputs. These different actions are manifested by different paths of the program. Here, a path is a sequence of exercised statements of the program. Thus, the path appropriately reflects the semantics of an input to a program because it represents how a program interprets this input. If two inputs exercise the same path of a program, they are processed by the same sequence of statements. Thus, they are *semantically* equal to each other. Otherwise, they are semantically different from each other.

We observe that the execution paths of a program under two inputs can be very similar to each other. For example, they can contain the same sequence of statements on certain parts. This indicates that the program takes the same action to respond to partial characteristics of the two inputs. It means the semantics of two test cases can be partially the same from the program's point of view.

Therefore, we propose *test case similarity* (TCS) between two test cases as a metric to represent such similar semantics. This metric is further used to extensively test deep program semantics as discussed below. Basically, it is defined as the edit distance between execution paths of two test cases of a program. Note that edit distance is widely used to measure the amount of difference between two strings. It is defined as the minimal number of edits (insertion, deletion, substitution) needed to transform one string into the other. More over, we *normalize* TCS into range $[0, 1]$, and two test cases are said to be semantically *closer* to each other if their test case similarity is relatively high (close to 1).

2.2. Two-stage fuzzing

We separate the fuzzing process into two stages and use TCS to direct the fuzzing process at each stage.

In the first stage, we propose to perform *incremental* fuzzing to systematically mutate each part of the well-formed inputs. Basically, a certain part of an input is duplicated and concatenated to form a new input. For example, "abc" can be mutated to "ababc". After a mutation, we feed the mutated input to the program under test for testing. Also, we record its execution path to calculate the TCS between the well-formed input and this input mutation. If the result is high, we know that a large part of the execution path is equal to that of the well-formed input. Since the well-formed input exercises deep program semantics, this input mutation is also supposed to exercise deep program semantics, at least with high probability. If the result is low, we switch to mutate other part of the well-formed input. In this way, we restrain the fuzzing process to the direction of testing deep program semantics.

In the second stage, we select test cases from the first stage whose TCS value is high and combine them to form new test inputs. Note that the selected test cases are semantically *close* to the well-formed inputs. Therefore, their combination is more likely to produce test cases to explore paths similar to the path exercised by the well-formed input. This is especially the case when there is little dependency between the combined parts of the inputs.

2.3. Deep program semantics

We say a program input is well-formed if it conforms to the input specification and is not rejected by the input validation logic of the program. Clearly, well-formed inputs satisfy more branch conditions of the program and exercise longer execution paths than ill-formed inputs. Hence, we deem that well-formed inputs exercise *deep* program semantics. We are interested in testing deep program semantics because they are hard or expensive to be

```
int f(char * q){
    char name[5];
    int magic;
    magic = getFourBytes(q);
    if(magic != 1234)
     return -1;
    q = q + 4;
    while(*q != NULL){
      *name++ = *q++;
    }
    return 0;
  }
```

Fig. 1. Sample program.

extensively tested but are prone to be vulnerable. First, although well-formed inputs are able to test them, they are usually designed to test different *functionalities* instead of different *paths* of the program. They do not extensively explore nearby paths. Second, many vulnerabilities such as those vulnerable pointer deferences in the *Wu-ftp*, *sendmail*, and *bind* programs, are triggered only when the involved loops are exercised a certain number of times (Zitser et al., 2004).

3. Main techniques

In this section, we first formally define test case similarity and discuss the algorithm complexity. Then we describe the two-stage fuzzing process in detail.

3.1. Test case similarity

Given a program P and two test cases I_1 and I_2, we define the *test case distance* between I_1 and I_2 as the edit distance between their paths and denote it as TCD(P, I_1, I_2). Here, a path is specified by a sequence of exercised branch conditions of program P under a test case I, and we denote it as $L(P, I)$. Note that we assign a unique value to each branch of the program. If a program has n branch conditions in source code, we use $2 * n$ unique values to represent their *true* and *false* branches. Hence, TCD(P, I_1, I_2) = ED($L(P, I_1)$, $L(P, I_2)$), where ED is a function that computes the edit distance between two paths. We assume that the cost of each insertion, deletion, or substitution of an element in the path is 1.

Given two test cases I_1 and I_2 of a program P, we select one of them as the *template* test case. We denote the test case similarity as TCS(P, I_1, I_2), where the first parameter specifies the program, the second parameter specifies the template test case, and the third parameter specifies the other test case. Note that the selection of template test case is arbitrary by definition, and later we will see that `well-formed` inputs are selected as templates in our fuzzing process. Suppose the test case distance TCD(P, I_1, I_2) = d, and the path length of the template test case is s. We define the *test case similarity* between I_1 and I_2 under program P as

$$\mathrm{TCS}(P, I_1, I_2) = \frac{s - d\prime}{s}, \quad d\prime = MIN(d, s).$$

It is clear that the minimal value of test case similarity is 0, and the maximal value is 1. Intuitively, the larger the test case similarity, the higher the semantic similarity between the two test cases.

Let us look at a concrete C program P in Fig. 1. Its input is a piece of data that has a magic value *1234* positioned in the beginning. It occupies 4 bytes and is used to indicate the validity of the input. The program first checks the magic value, then copies the data to a local buffer if it is correct. There are two conditions c_1: *magic* ! = 1234 and c_2: $^*q$! = *NULL* in the program. We use $c_{i:T}$ and $c_{i:F}$ to represent the *true* and *false* branches of the ith condition c_i(i = 1, 2), respectively. Thus, the four condition signatures are $c_{1:T}$, $c_{1:F}$, $c_{2:T}$, $c_{2:F}$. Suppose we have a well-formed input I_1 : { "1234*aa*" } and we select it as the template test case. It is clear that its path condition is $\langle c_{1:F}, c_{2:T}, c_{2:T}, c_{2:F}\rangle$. If we mutate it to two test cases I_2 : { "3091*aa*" } and I_3 : { "1234*aaaa*" }, then their path conditions are $\langle c_{1:T}\rangle$ and $\langle c_{1:F}, c_{2:T}, c_{2:T}, c_{2:T}, c_{2:T}, c_{2:F}\rangle$ respectively. According to the definition of test case distance, we can derive that TCD(P, I_1, I_2)=4 and TCD(P, I_1, I_3)=2. Therefore, the test case similarity is $(4-4)/4 = 0$ between I_1 and I_2 and is $(4-2)/4 = 0.5$ between I_1 and I_3. It means that I_3 is semantically closer to I_1.

There is a vulnerability in Fig. 1; the buffer of *name* can be overflowed when the length of the parameter content is greater than 5. Note that the input mutation I_2 strays far away from the deep program semantics; while I_3 explores deeper program states, we will continue the mutation on the data part of the input. Using the incremental mutation strategy as shown in Section 3, we will detect this vulnerability on the next mutation.

Now we analyze loop effects on test case similarity. Suppose a test input exercises a path L of a program. We assume that the length of L is s and it unfolds a loop r for n times. In other words, the number of other branch conditions in L is $s - n$. Suppose an input mutation exercises the same path but unfolds loop L for n' times. Thus, the edit distance between the paths of these two inputs is $|n' - n|$. From the definition of test case similarity we can see that the TCS value is 0 if $|n' - n| \geq s$. Specifically, we can derive that if any of the following two conditions is satisfied: (I) $n' = 0$ and $s = n$ (II) $n' >= 2n + (s - n)$, the TCS value becomes 0. Therefore, for any loop of a path, if it is exercised n times by a template input, the input mutations that explore it from 1 to $2n$ times do not make the TCS value become 0. For example, suppose there is a test case I : { "1234$a \ldots a$" } that exercises the loop in Fig. 1 100 times. Then, I is semantically close to those inputs that explore the loop from 0 to 200 times.

3.1.1. Complexity

Since the execution path of a program can be huge, the complexity of the algorithm to calculate the test case similarity is crucial to the performance of our fuzzing approach.

From the definition, it is clear that the complexity to evaluate the test case similarity is equal to the complexity to evaluate the edit distance between two sequences. The problem of computing the edit distance between two sequences has been studied extensively (Myers, 1986; Andoni and Onak, 2009). The classical dynamic programming algorithm takes $O(N^2)$ time and space. It has been reduced to $O(ND)$ in Myers (1986) where N is the sum of the input lengths and D is the size of the minimum distance between the inputs. Moreover, approximate algorithms have also been proposed to further reduce the complexity to nearly linear (Andoni and Onak, 2009). In our experiment, we use the algorithm in Myers (1986), which turns out to be quite efficient as shown in Section 5.

In addition, we also apply several optimization methods such as caching, which will be discussed in Section 4, to further improve the performance. As a result, our fuzzing approach can quickly generate test cases and is almost as efficient as blackbox fuzzing.

3.2. Incremental mutation

Our framework supports any mutation strategy such as random mutation (i.e., randomly changing the input values). In this paper, we are more interested in detecting memory corruption vulnerabilities such as buffer overflow and pointer out-of-boundary operations. For this purpose, we propose an *incremental* mutation strategy based on the following observations.

First, we have checked the description of the top 100 buffer overflow vulnerabilities from searching the results of the National Vulnerability Database (NVD) (Anon., 2011a), SecurityFocus (Anon., 2011b), and SecurityTracker (Anon., 2011c). We found that nearly

80 percent of these vulnerabilities are explicitly stated or indirectly caused by an extreme value (of an internal variable) that is derived from a single external parameter. For example, the description of vulnerability CVE-2009-4841 in NVD explicitly states that "*. . . allows remote attackers to execute arbitrary code via a long argument to the DiskType method*". Second, intuitively, a slight mutation of the test input is more likely to produce an input that explores a nearby path compared to a significant mutation.

Based on the above observations, we propose to divide a well-formed input into small segments and *slowly increase* the amount of mutation on each segment. The segment size is from 1 to the size of the well-formed input. In practice, a larger segment size can be set initially. When more test cases are intended to be generated, a smaller segment size can be used. Once the mutation of one segment results in a test case that is *far away* from the well-formed input, we know that the resulting input mutation has quite different semantics, e.g., it has become ill-formed. Thus, continue increasing the mutation of the current segment is more likely to produce test cases that are also semantically different from the well-formed input. So, we start the mutation on the next segment. Whether a test case is semantically close or far away from the well-formed input can be checked by performing the test case similarity analysis. It is important to point out that our approach does not depend on any input specification or domain knowledge of the input parameters. We simply treat each segment as an input parameter.

Algorithm 1. Incremental mutation

```
input: P, I, incvec, segsize, v1, quote1
output: testcases
begin
  length = lengthOf(I);
  num = length/segsize;
  sum1 = 0;
  for i = 0; i < num; i++ do
    for j = 0; j < incvec.size(); j++ do
      sum1++;
      if (sum1 >= quote1) then
        return;
      end
      I' = mutate(I, i, j);
      testing(I');
      v = TCS(P, I, I');
      update(I', v);
      testcases.add(I');
      if v < v1 then
        break;
      end
    end
  end
end
```

The incremental mutation algorithm is described in Algorithm 1. In addition to the program under test and the well-formed input, the tester needs to set several testing parameters for this strategy: the segment size *segsize*, a sequence of incremental numbers *incvec*, the test case similarity threshold $v1$, and the maximum number of new test cases *quote*1. Specifically, given a program P and a well-formed input I, we divide I into a number of segments where each segment is *segsize* (or less than *segsize* for the last segment) bytes long. Then we apply heuristics to incrementally mutate each segment using a sequence of incremental numbers in the vector *incvec*. For example, if *incvec* is a serial of incremental numbers ($m_1, m_2, \ldots, m_k$) and the mutation strategy for a segment is *content duplication*, then the ith ($1 \le i \le k$) round of mutation on a segment will result in a new segment whose length is m_i * *segsize* and whose content is the repetition of the original content for m_i times. After producing an input mutation, we feed it to the program for testing and calculate its test case similarity. We will stop mutating the current segment and switch to the next segment once the resulting similarity is lower than threshold $v1$. The total number of input mutations is limited by the parameter *quote*1. Note that we test the whole program instead of a function. The inputs to a program are always block(s) of data. We do not need to consider the internal structure of the inputs, and we do not need to consider what data structures (e.g., pointer) are used in the program to process the inputs. We simply treat each well-formed input as a block of data, such as a file or a network packet.

3.3. Two-stage fuzzing

Given a program P, a well-formed input I, and testing parameter values, the two-stage fuzzing procedure is described in Algorithm 2.

Algorithm 2. Two-stage fuzzing

```
input P, I, v1, v2, quote1, quote2, strategy
output testcases
begin
  /*Stage I*/
  testcases = mutation(P, I, v1, quote1, strategy);
  /*Stage II*/
  sum2=0;
  candidates = select_test_cases(testcases, v2);
  for i = 0; i < candidates.size(); i++ do
    if (sum2 >= quote2) then
      break;
    end
    for j = i + 1; j < candidates.size(); j++ do
      I' = combine(I, testcases[i], testcases[j]);
      testing(I');
      update(I', TCS(P, I, I'));
      testcases.push_back(I'); sum2++;
      if (sum2 >= quote2) then
        break;
      end
    end
  end
  /*Optional Stage*/
  if sum1 + sum2 < quote2 then
    generate_remain_tests(quote2-sum2);
  end
end
```

3.3.1. Stage I

In this stage, we perform mutation using the specified mutation strategy such as random mutation or incremental mutation discussed above. The type of mutation is specified in the testing parameter *strategy*. Note that random mutation produces *unpredictable* test cases in the first stage. As a result, if random mutation is used, the test case similarity analysis is not applied. Remember that the test case similarity value between each input mutation and the well-formed input is recorded after this stage.

3.3.2. Stage II

In this stage, we employ a combination testing strategy to further explore nearby paths based on the test case similarity analysis. Specifically, we select all mutations of the well-formed input from the first stage whose test case similarity is greater than threshold $v2$. These test cases are then combined to form new test inputs for testing. These new test cases are also likely to explore nearby paths, especially when the combined mutated parts are independent of each other. Multiple combination strategies have been proposed (Grindal et al., 2005). We use a simple technique to conduct a 2-way combination on two mutations of a well-formed input. Specifically, suppose the content of a segment (for incremental mutation) or a byte (for random mutation) of the well-formed input is X, and the

corresponding mutation value of two input mutations are Y and Z, respectively. The combination result of X, Y, and Z is

$$combination(X, Y, Z) = \begin{cases} X, & X = Y, Y = Z \\ Y, & X = Z, X \; != Y \\ Z, & X = Y, X \; != Z \\ Y \text{ or} Z, & X \; != Y, X \; != Z \end{cases}$$

Note that we randomly select Y or Z as the combination result when $X \; != Y$ and $X \; != Z$.

This combination capability is important in identifying vulnerabilities that are only reachable under combination conditions. In contrast, a blackbox fuzzing approach has difficulty to conduct such combination testing without specific domain knowledge because of the large number of mutations and the unguided selection of mutations for combination. It is infeasible to exhaustively combine all test cases in practice.

3.3.3. Optional stage

There is an optional stage in Algorithm 2. The reason is that it is possible that the number of test cases generated in stage I and II does not reach the sum of *quote*1 and *quote*2 under strategies such as incremental mutation. The optional stage is designed for this situation. Specifically, we have several options to generate the remaining number of test cases. For example, we can simply stop the test case generation, or we can change the values of the testing parameters to continue producing the required number of test cases. Note that this stage is necessary especially when the initial testing parameters are *strict* and not enough test cases can be generated.

4. System design and implementation

The system architecture of `SimFuzz` is shown in Fig. 2. It takes three inputs, (1) the source code of the program under test, (2) a set of well-formed inputs, and (3) the testing configuration.

4.1. Components

The testing configuration is a configuration file in which the user can specify testing parameters as discussed above. Also, information such as database settings is also specified in this file since we store the test case information in a database. The transformer component instruments the program under test with condition-logging code to record the execution paths into a file. Currently, `SimFuzz` supports the testing of C programs. It uses CIL (Necula et al., 2002) to perform the program transformation. The mutator component mutates a given input using the specified mutation strategies such as random mutation or the incremental mutation strategy discussed in Section 3. Here, random mutation means to randomly select a certain number of bytes of the original input and change their values to form new inputs. The parameters used during input mutation are specified in the testing configuration file. The monitor component runs the transformed program under each input mutation and captures any abnormal signal such as *segmentation faults* during testing. Note that other runtime memory corruption detection techniques such as TaintCheck (Newsome and Song, 2005) can be also applied. Currently, we use the gcc boundary checking tool (Jones and Kelly, 1997) to compile the transformed program and monitor any out-of-bounds operation. After testing each input mutation, the analyzer calculates the test case similarity, which is used to guide both the mutator and the combinator. The testing driver component organizes all these components and makes the fuzzing process fully automatic.

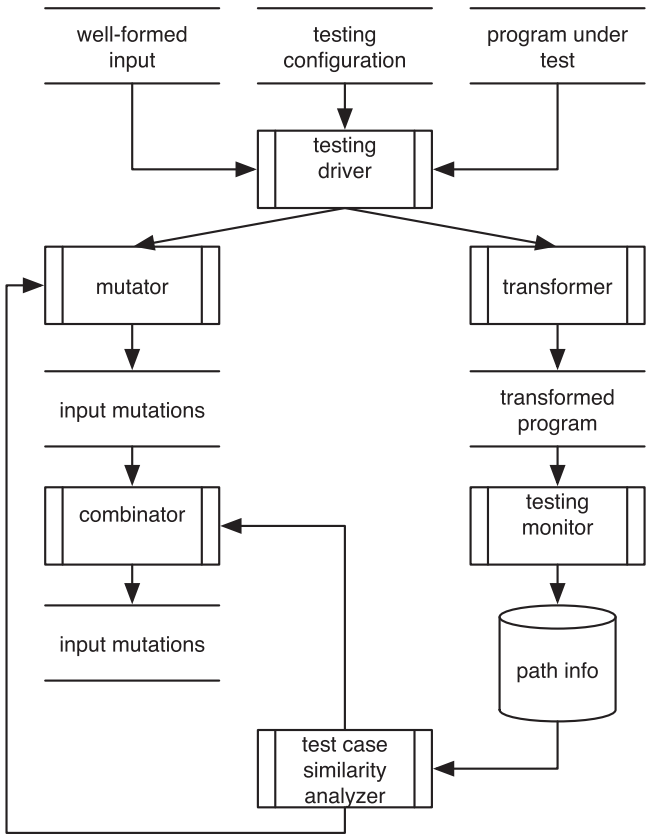

Fig. 2. `SimFuzz` architecture.

4.2. Optimizations

To improve `SimFuzz` performance, we use several optimization techniques. First, we use the algorithm in Myers (1986) to compute the edit distance. This algorithm turns out to be very efficient in our experiments. Second, we note that redundant test cases may explore the same path exercised before. Hence, we use a cache to store recent path signatures and the test case similarity values to accelerate the test case similarity calculation. Third, we note that `SimFuzz` does not need the exact edit distance value if we are certain that it is greater than the path length of the template test case. Also, the edit distance between two paths must be greater than or equal to the difference of their path length. Therefore, we can safely use the path length of the template test case as the edit distance when the path length difference is greater than the path length of the template test case.

5. Experiments

We did experiments to evaluate whether our approach is effective and efficient for extensively testing deep program semantics. We evaluate the coverage, explored path depth, and the vulnerability detection rate to show the effectiveness, and we use testing time to show the efficiency. We also use other possible approaches to perform the testing and compare the result with that of our approach.

5.1. Experimental setup

The experiments were conducted on a 2 G Intel Core 2 CPU 4300 1.80 GHz desktop with 2GB RAM running the Ubuntu 8.10 operating system.

Table 1
Testing configurations.

Config ID	Strategy	Stages	Test case similarity directed?
1	Random	1	No
2	Incremental	1	No
3	Random	2	No
4	Incremental	2	No
5	Incremental	1	Yes
6	Random	2	Yes
7	Incremental	2	Yes

Table 2
Coverage information.

Config ID	B-Cov	S-Cov	P-Cov	♯Distinct paths
1	33.24%	50.32%	24.22%	1235
2	31.47%	47.69%	18.33%	935
3	33.00%	53.53%	13.37%	682
4	31.47%	47.55%	19.80%	1010
5	31.84%	48.99%	50.50%	2317
6	33.24%	50.32%	24.22%	1235
7	31.70%	48.44%	63.94%	2976

5.1.1. Program under test

We conduct testing on the open source C program expat-2.0.1 (Anon., 2011d), an XML parser library that has 16,055 lines of code (calculated by *cloc* (Anon., 2011e)) and is widely used (e.g., in Mozilla).

5.1.2. Well-formed inputs

The program under test takes XML files as input. We randomly select 100 of the 149 *valid* xml files in the */ibm/valid/* directory of the standard XML W3C Conformance Test Suites (Anon., 2011f) as our well-formed inputs. We use the latest version (20080827) of this test suite.

Note that taking well-formed input as *seed files* has been adopted in both traditional blackbox fuzzing and white-box fuzzing (Godefroid et al., 2008). The required well-formed inputs are usually not hard to obtain for many of the programs that manipulate standard input formats, such as server programs that process standard protocols or parser and reader programs that accept standard inputs (e.g., doc, pdf, xml, mpeg4, etc.). In addition, remember that we only need such input data and do not need to understand the underlying input grammar as in Röning et al. (2002) and Godefroid et al. (2008).

5.1.3. Testing configuration

We did comparison experiments between `SimFuzz` and traditional blackbox fuzzing from different perspectives. These testing configurations are specified in Table 1. The first column is the configuration ID; the second column specifies the mutation strategy used; the third column specifies how many stages are involved for this configuration; the last column specifies if our test case similarity analysis is applied during testing. Note that for the random mutation strategy, we use the *zzuf* (Hocevar, 2008) fuzzing tool to randomly mutate the given input.

Other testing parameters for these testing configurations are as follows: *segsize*=4, *incvec* = {2, 4, 7, 10, 13}, *quote*1 = 25, *quote*2 = 25, $v1 = 0.9$, $v2 = 0.9$. Note that for certain testing strategies, some of the above parameters are not used. For example, the random mutation strategy does not need the *segsize*, *incvec*, and $v1$ parameters. In addition, if a given testing configuration only involves one stage, we use *quote*1 + *quote*2 as the limitation of the maximum number of input mutations for a well-formed input. In other words, we have the same limitation on the maximum number of input mutations for all these testing configurations. There is no exact method to decide the values of these initial testing parameters. However, we can control the *strictness* of our fuzzing approach. For example, if we want to generate more test cases, we can set smaller *segsize*, larger *quote*1, *quote*2, smaller $v1$ and $v2$, and slower increasing values in *incvec*. The effects of these values are specific to applications. In practice, a tester can first set stricter values to produce fewer test cases, then gradually relax them to produce more.

We evaluate the performance of different testing configurations through the comparison of metrics such as branch coverage and statement coverage obtained using *gcov*, and we also evaluate the number of distinct paths under each testing configuration.

5.2. Results and discussion

5.2.1. Coverage

Table 2 shows the coverage under every testing configuration. The first column is the configuration ID. The second column is the branch coverage, computed as the number of tested branches divided by the total number of branches in the program. The third column is the statement coverage, computed as the number of tested statements divided by the total number of statements in the program. The fourth column is the P-coverage, which measures the percentage of distinct paths. Specifically, it is the ratio between the number of distinct paths exercised by all generated test cases and the total number of exercised test cases. The last column is the total number of distinct paths that were tested.

Note that for configuration 5 and 7, the total number of created test cases is 4488 and 4554, respectively. For other configurations, the total number of created test cases is 5000. We will explain the reason later.

5.2.1.1. Discussion. First, we analyze the effect of the test case similarity analysis on coverage. We consider the following three groups of configurations: (i) configurations 2 and 5, (ii) configurations 3 and 6, and (iii) configurations 4 and 7. The only difference between the configurations in the same group is whether test case similarity analysis is applied during the fuzzing process. As we can see, in each group, if the test similarity analysis is applied, the generated test cases will explore 1.8 to 3.2 times more new paths. The fact that configuration 2, 3, and 4 have low P-coverage indicates that many of the input mutations are redundant and do not explore new branches or paths. On the contrary, our approach mitigates such redundancy by using test case similarity analysis. At the same time, our approach achieves nearly the same branch coverage and statement coverage. This indicates that the test case similarity analysis does not over-reduce the blackbox fuzzing effort under reasonable configurations. We do not aim to improve branch coverage, and we noticed that our technique does not help that. This illustrates a fundamental limitation of blackbox fuzzing approaches, i.e., it is hard to achieve high branch coverage since the possibility of a randomly generated value satisfying a condition like $i == 0$ is extremely low. However, once more powerful blackbox fuzzing approaches (e.g., input specification-based approaches) are available, our approach can directly use them and further amplify its effectiveness.

Second, we analyze the effect of the two-stage fuzzing strategy. We consider the following three groups of configurations: (i) configurations 1 and 3, (ii) configurations 2 and 4, and (iii) configurations 5 and 7. The difference between the configurations in the same group is the number of stages. We can see that for groups i and ii, the two-stage strategy does not improve the P-coverage. The reason is that groups i and ii did not apply the test case similarity analysis, and therefore the two-stage strategy does not help them. However, for group iii, we can see that the P-coverage is further improved. It indicates that the combination of semantically *close* test cases can be every effective in generating test cases that explores new paths. Thus, the two-stage strategy is useful when combined with test case similarity analysis.

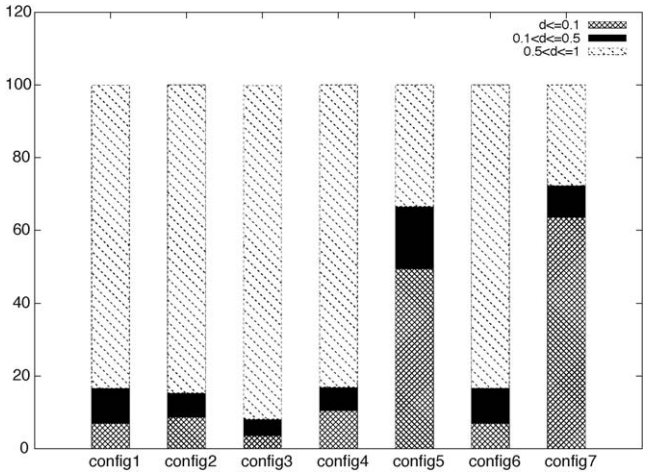

Fig. 3. Path depth distribution.

Third, we compare the *random* and *incremental* mutation strategies. We consider the following three groups of configurations: (i) configurations 1 and 2, (ii) configurations 3 and 4, and (iii) configurations 6 and 7. The difference between the configurations in the same group is the mutation strategy. We can see that for groups i and ii, the incremental mutation does not improve the P-Coverage. For group iii, the P-coverage is greatly increased. Again, the reason is that group iii is directed by the test case similarity analysis. Thus, incremental mutation is useful when combined with test case similarity analysis.

In addition, we can see that configuration 6 did not perform as well as configuration 5 and 7. For configuration 6 and 7, we have discussed the reason above. For configuration 5 and 6, we can see that even when we only perform one-stage incremental mutation, we get better results than two-stage random mutation although both of them are under the test case similarity analysis. The reason is that random mutation produces more redundancy in the first stage compared to the incremental mutation strategy, since test case similarity analysis does not make sense to *direct* random mutation in the first stage. In other words, test case similarity analysis is not applied in the first stage if the random mutation strategy is used. Therefore, in the second stage, there are fewer *close* test cases that satisfy the combination standard for configuration 6 in our experiments.

In summary, we can see that test case similarity plays a critical role in improving P-coverage. In addition, the two-stage and incremental fuzzing strategy can further improve the P-coverage when combined with test case similarity analysis. Note that for configurations 5 and 7, the total number of test cases is less than the sum of *quote*1 and *quote*2. The reason is that `SimFuzz` stops fuzzing on a given segment if the fuzzing on such segment produces a test case that has low similarity, regardless of whether or not enough test cases have been produced.

5.2.2. Path depth

Typically, if the length difference between the execution paths of two test cases is small, the two test cases do not necessarily explore similar paths. On the contrary, if two test cases are semantically close to each other, the length difference between their execution paths will be quite small. Fig. 3 shows the length information of the paths exercised by the test cases generated from each testing configuration. For each input mutation, if it has path length b, and the well-formed input has path length a, then we calculate their length difference as $d=|a-b|/a$. The figure shows the percentage of test cases where $d \leq 0.1$, $0.1 < d \leq 0.5$, $0.5 < d \leq 1$, respectively. We can see that configurations 5 and 7 also generate more test cases with $d \leq 0.1$ than other configurations. In other words, the test cases generated from these two configurations will be more effective in exploring new paths that are semantically close to the well-formed test inputs.

Table 3
Path depth.

Config ID	Average distinct path depth	Average path depth
1	2287	1201
2	2824	1299
3	2368	730
4	2887	1339
5	5101	4013
6	2286	1199
7	5488	4462

Table 4
Performance.

Config ID	Testing time (min)	Estimated SymExe time (min)
1	64	1244
2	70	1536
3	45	1288
4	74	1570
5	158	2496
6	68	1243
7	172	2724

The average path depth is also shown in Table 3. And we can see that test case similarity directed fuzzing produces deeper program execution paths compared to straight fuzzing.

5.2.3. Overhead evaluation

Column 2 in Table 4 shows the total testing time for each test configuration, including test data generation and the run time of the target program. As we can see, fuzzing processes that are directed by test case similarity analysis (Config ID 5, 6, 7) are around 2 times slower than corresponding straight fuzzing processes. Remember that Config 6 uses the random mutation strategy and produces fewer mutated data that are qualified for combination in stage 2. Therefore, it has less testing time than Config 5 and Config 7.

To compare the overhead with a whitebox fuzzing approach, we select the state-of-art whitebox fuzzing tool `SAGE` (Godefroid et al., 2008). We were not able to conduct our testing directly under `SAGE` since it is not public available. In addition, the applications tested in Godefroid et al. (2008) are internal MS programs and are not clearly specified. Therefore, we *estimate* the symbolic execution time of `SAGE` based on experimental results in Godefroid et al. (2008). In Godefroid et al. (2008), the authors applied `SAGE` to 7 applications and the symbolic execution time, the number of test cases, the average path depth (constraints) are provided. Based on these data, we calculate that it takes 0.0064 s on average for `SAGE` to perform symbolic execution on one constraint. Therefore, we can estimate the symbolic execution time when `SAGE` explores the same distinct paths as `SimFuzz` does. Note that here we use *distinct* paths because `SAGE` always explores new paths during symbolic execution. Column 3 in Table 4 is the estimated time. As we can see, to explore the same set of distinct paths, `SAGE` is estimated to take at least 16 times more time than `SimFuzz`. In other words, we can finish our testing in hours while `SAGE` may take days.

However, we emphasize that this is only a roughly comparison. Also, `SAGE` is a powerful tool to explore new *branches* while `SimFuzz` is designed to explore deep program paths efficiently.

5.2.4. Vulnerability detection simulation

Our experimental result above has shown that `SimFuzz` is able to explore deeper program paths efficiently. Therefore, it can test more program semantics and potentially find more bugs. To better

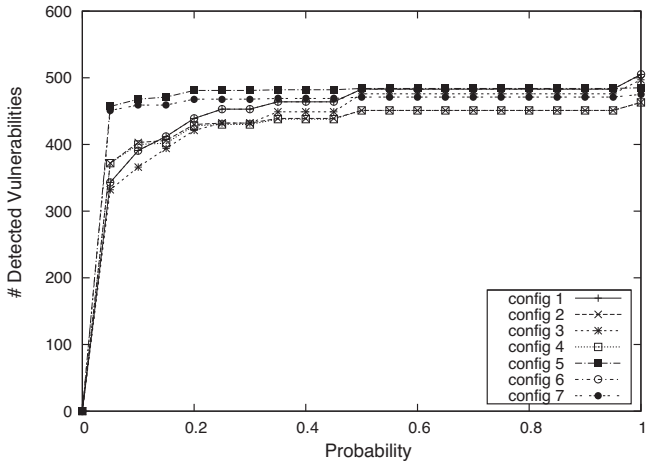

Fig. 4. Vulnerability detection.

demonstrate bug detection ability, we simulate the vulnerability distribution in the program to evaluate `SimFuzz` from another aspect.

First, we instrument a vulnerable statement (a call to a self-defined dummy function) at each branch of the program with a probability p to simulate the static vulnerability distribution in the program. We inject 2368 vulnerable statements into the program with $p = 0.3$. Second, a vulnerability is often path specific and the execution of a vulnerable statement does not necessarily trigger the vulnerability. In other words, a test case may only trigger (detect) a vulnerability when it drives the program to execute specific paths to reach the vulnerable point. In addition, even if two test cases exercise the same path, it is possible that one triggers the vulnerability (so that we can detect it during testing) and the other does not. For example, one test case may not contain a long enough string to trigger a buffer overflow while the other does not. To simulate such a scenario at runtime, we trigger a vulnerability with probability q when an injected vulnerable statement is exercised. In other words, if an execution path contains a vulnerability, a test case that exercises such a path will trigger such vulnerability at a probability of q. Fig. 4 shows the number of detected vulnerabilities under each testing configuration for various values of q ($0 \le q \le 1$). Note that although $p = 0.3$ may be high in practice, the value of q actually controls whether a vulnerability is triggered or not when the vulnerable statement is exercised. And we showed the distribution of the number of detected vulnerabilities under all possible values of q.

5.2.4.1. Discussion. We note that configurations 5 and 7 detected more vulnerabilities than other configurations. Both of them are directed by the test case similarity analysis. Note that configuration 5 allows us to detect slightly more vulnerabilities when compared to configuration 7. The reason is that with configuration 5, we explore fewer distinct paths. In other words, each path is exercised by more test cases, which leads to a higher probability of triggering the vulnerability in this path. The testing strategies that do not use the test case similarity analysis detect fewer vulnerabilities even though they can hit vulnerable statements more times due to redundant test cases.

Fig. 4 also shows that the number of detected vulnerabilities increases with q, and the performance of different testing configurations becomes close to each other. The reason is that these configurations result in similar branch coverage. A larger q indicates a higher probability of triggering a vulnerability when the corresponding vulnerable statement is exercised. In an extreme case where $q = 1$, the vulnerability detection rate becomes the same as the branch coverage since a vulnerability is triggered every time when the vulnerable statement is exercised.

5.2.5. Summary

In our experiment, we conducted testing using different settings for other parameters such as *segsize* and *incvec*. The results are consistent with what we have presented above. In addition, we have also evaluated the performance overhead of our approach as shown in Fig. 4. On average, the fuzzing process directed by test case similarity analysis is only 1.5 times slower. Thus, in summary, we believe that our approach is efficient and effective in practice.

6. Limitations and countermeasures

The current implementation of our approach needs source code of the program under test. However, the test case similarity definition is not bound to program source code in principle. It is possible to employ binary rewriting tools, e.g., Pin (Luk et al., 2005), to add condition logging codes into the program image which can be used to calculate test case similarity. Thus, `SimFuzz` can be extended to the binary level.

It is possible that the first stage mutation only gives a small number of qualified test cases for combination in the second stage, although it did not happen in our experiments. We believe that for programs that process strings or text, the first stage mutation is likely to produce more test cases that are similar to the well-formed inputs, especially when the segment size is small.

In our experiments, we randomly inject and trigger vulnerabilities to evaluate the performance of vulnerability detection of our approach. However, such simulation may not reflect the vulnerability distribution in real programs. This can be a threat to our evaluation result. Although we did extensive testing under various types of configurations, we applied SimFuzz on a single application. This can also be a threat to the validity of our approach.

At last, more information can be obtained from source code using program analysis techniques to further improve our fuzzing process. For example, through code analysis, we can select more important paths for testing, such as paths containing security critical function calls. We can also perform data flow analysis to identify dependent parts of input data for mutation (Ganesh et al., 2009). These techniques can further improve our approach.

7. Related work

Runtime detection techniques monitor and protect runtime system integrity (Newsome and Song, 2005; Cowan et al., 1998; Bhatkar et al., 2003). For example, StackGuard (Cowan et al., 1998) adds a *canary* word to monitor the modification of return addresses in the stack. TaintCheck (Newsome and Song, 2005) tracks the flow of tainted data and raises alarms whenever they are used at critical points such as the target of a jump instruction. Address randomization techniques (Bhatkar et al., 2003) randomize program addresses. Thus, many software attacks will crash the program and thus be detected. These techniques provide powerful mechanisms to improve the runtime security of a system. However, they introduce additional runtime overhead.

Prevention techniques identify program vulnerabilities offline to eliminate them before deployment. Static analysis methods scan source code for vulnerabilities (Viega et al., 2000; Wagner et al., 2000); they usually produce many false positives due to over-approximation. Significant effort is often required to evaluate the analysis results. Software testing does not generate false positives and is widely used in practice. However, automatic test data generation is difficult, and sometimes domain knowledge such as input specification is required. Recently, dynamic symbolic execution has

been proposed to help this process (Godefroid et al., 2005; Sen et al., 2005; Cadar et al., 2006). Basically, a program is executed both concretely and symbolically to collect path constraints. A constraint solver is then employed to solve these constraints to generate test data for new paths. This technique is promising but still has many limitations such as unsolvable constraints, path explosion, and imprecision on complex data structures.

Fuzzing was first proposed as a blackbox testing method to test the reliability of UNIX utilities (Miller et al., 1990). It has been widely used in penetration testing and security testing to detect software vulnerabilities. Many fuzzing frameworks or tools have been released to fuzz diverse program inputs such as file formats and network protocols (Sutton et al., 2007; Röning et al., 2002). For example, PROTO (Röning et al., 2002) is a specification based approach. It uses context-free grammars (BNF) with extensions to model input syntax, inserts anomalies into valid specification to simulate faults, and tests the security of protocol implementations. One disadvantage is that it is time-consuming to acquire proper input grammars, especially for complex input formats. Our approach provides a mechanism to select desired input mutations in terms of path exploration without using any knowledge about the input specification.

Software fault injection techniques introduce different kinds of faults into a system to detect vulnerabilities, and is similar to the fuzzing approaches. The work in Du and Mathur (2000) simulates environment faults to a program and tests the behavior of the target program. It can extensively test the fault-tolerance property of a system. Our approach has a different testing purpose: we aim to further explore the nearby paths that are missed by traditional fuzzing.

Evolution algorithms have been proposed to improve fuzzing. The work in Sparks et al. (2007) extends blackbox fuzzing using a genetic algorithm, which uses the past branch profiling information to direct the input generation in order to cover specified program regions or points in the control flow graph. Our approach can also be extended in a similar way. For example, we can further select combination test cases from the second stage to produce the next generation input mutations. However, our approach has the following benefits. First, we do not need input grammar which is used in Sparks et al. (2007) for input mutation. Second, our approach does not need a set of pre-defined attack points such as the call site of *strcpy* to conduct testing. Although such target oriented mutation may save testing effort, it has drawbacks that only known or chosen attack points are targeted during testing while the missed or unknown attack points (vulnerabilities) are ignored. Work in Grosso et al. (2008) also uses genetic algorithms to generate test data for buffer overflow detection. The algorithm in `SimFuzz` is similar to the genetic algorithm in the sense of test data generation and path selection for combination. Unlike (Grosso et al., 2008), the purpose of `SimFuzz` is to automatically explore deeper program semantics.

Concolic testing performs dynamic symbolic execution and employs constraint solving techniques to automatically generate test data to explore different paths (Godefroid et al., 2005; Sen et al., 2005; Cadar et al., 2006, 2008). In general, these approaches execute the program both concretely and symbolically, and systematically collect and negate path conditions to generate test cases for new branches. These techniques have several limitations due to the path explosion problem, the frequent constraint solving requests, the unsolvable constraints, and the inaccurate symbolic execution on complex structures. Researchers have proposed to use input grammars to perform grammar-based fuzzing (Godefroid et al., 2008), combine dynamic symbolic execution and random testing (Majumdar and Sen, 2007), use function summaries (Godefroid, 2007), model loops more efficiently (Saxena et al., 2009), etc. However, those fundamental challenges are still wide open problems.

BuzzFuzz (Ganesh et al., 2009) uses dynamic taint analysis to identify inputs that propagate their values to a set of attack points in the program and only performs fuzzing on the identified inputs to reduce the testing cost. This approach requires a set of pre-defined attack points. Another drawback of this approach is that it is difficult to capture indirect (e.g., control) dependencies accurately. For example, out-of-bound pointer deference is an important attack point that can lead to DoS attacks but is often indirectly dependent on inputs. In addition, dynamic taint analysis makes the system 20–30 times slower, meaning that less testing can be conducted with limited testing resources.

Basis path testing uses cyclomatic complexity metrics (McCabe, 1976) to test linearly independent paths of a program. A linearly independent path is any path of the program that introduces at least one new edge (in the control flow graph) that is not included in any other linearly independent paths. Given a program, the number of test cases that cover all basis paths is deterministic. However, our approach focuses on the extensive testing of deep program states. We usually explore paths that are missed by basis path testing approaches. For example, if a program has a statement "*while*($a > 0$)$s1$;", there are only two basis paths: one covers the *false* branch; another covers the *true* branch a certain time (or 1 time) and then covers the false branch to leave the loop. However, our approach will test different paths more extensively.

8. Conclusion

In this paper, we propose a novel metric, test case similarity, to model semantic similarity between test cases. We then propose a two-stage fuzzing framework to automatically and efficiently explore *deep* program semantics given a set of well-formed inputs. Our experimental results show that our approach can efficiently explore deeper program paths, and has the potential to detect more path specific vulnerabilities.

In the future, we plan to improve `SimFuzz` in several aspects. We would like to extend `SimFuzz` to support binary level test case similarity analysis and do more experiments on other C programs. We are also interested in applying this approach to web applications to detect other security vulnerabilities such as XSS. In this paper, we randomly insert vulnerabilities to evaluate the performance. It is also interesting to apply non-parametric statistical tests during the evaluation.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1017305.

References

Andoni, A., Onak, K., 2009. Approximating edit distance in near-linear time. In: Proceedings of the Annual ACM Symposium on Theory of Computing.
Anon., 2011a. Available: http://nvd.nist.gov (Online).
Anon., 2011b. Available: http://www.securityfocus.com (Online).
Anon., 2011c. Available: http://www.securitytracker.com (Online).
Anon., 2011d. Available: http://expat.sourceforge.net (Online).
Anon., 2011e. Available: http://cloc.sourceforge.net (Online).
Anon., 2011f. Available: W3C XML http://www.w3.org/XML/Test (Online).
Bhatkar, S., DuVarney, D., Sekar, R., 2003. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium.
Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D., 2006. EXE: automatically generating inputs of death. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS).
Cadar, C., Dunbar, D., Engler, D., 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex system programs. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).
Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium.

Du, W., Mathur, A., 2000. Testing for software vulnerability using environment perturbation. In: Proceedings of International Conference on Dependable Systems and Networks (DSN).
Ganesh, V., Leek, T., Rinard, M., 2009. Taint-based directed whitebox fuzzing. In: Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE).
Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223.
Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. In: ACM SIGPLAN Notices.
Godefroid, P., Levin, M.Y., Molnar, D., 2008. Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed Systems Security (NDSS).
Godefroid, P., 2007. Compositional dynamic test generation. In: Proceedings of the Symposium on Principles of Programming Languages (POPL).
Grindal, M., Offutt, A.J., Andler, S.F., 2005. Combination testing strategies: a survey. In: Software Testing, Verification, and Reliability.
Grosso, C.D., Antoniol, G., Merlo, E., Galinier, P., 2008. Detecting buffer overflow via automatic test input data generation. Computers and Operations Research 35, 3125–3143.
Hocevar, S., 2008. "Hot fuzzing with zzuf," Hacker Space Festival (HSF). Vitry sur Seine, France.
Jones, R., Kelly, P., 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the International Workshop on Automated Debugging.
Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwoood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (SIGPLAN).
Majumdar, R., Sen, K., 2007. Hybrid concolic testing. In: Proceedings of the international conference on Software Engineering (ICSE).
McCabe, T.J., 1976. A complexity measure. In: Proceedings of the 2nd International Conference on Software Engineering.
Miller, B.P., Fredrikson, L., So, B., 1990. An empirical study of the reliability of Unix utilities. Communications of the ACM 32 (12).
Myers, E.W., 1986. An o(nd) difference algorithm and its variations. Algorithmica 1, 251–266.
Necula, G.C., McPeak, S., Rahul, S., Weimer, W., 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proceedings of the International Conference on Compiler Construction.
Newsome, J., Song, D., 2005. Dynamic taint analysis for automatic detection analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium (NDSS).
Röning, J.J., Laakso, M., Takanen, A., 2002. Protos: systematic approach to eliminate software vulnerabilities. http://www.ee.oulu.fi/research/ouspg, May.
Saxena, P., Poosankam, P., McCamant, S., Song, D., 2009. Loop-extended symbolic execution on binary programs. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).
Sen, K., Marinov, D., Agha, G., 2005. CUTE: a concolic unit testing engine for c. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering.
Sparks, S., Embleton, S., Cunningham, R., Zou, C., 2007. Automated vulnerability analysis: leveraging control flow for evolutionary input crafting. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC).
Sutton, M., Greene, A., Amini, P., 2007. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional.
Viega, J., Bloch, J., Kohno, Y., McGraw, G., 2000. Its4: A static vulnerability scanner for C and C++code. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC).
Wagner, D., Foster, J., Brewer, E., Aiken, A., 2000. A first step towards automated detection of buffer overrun vulnerabilities. In: Proceedings of the Network and Distributed System Security Symposium (NDSS).
Zitser, M., Lippmann, R., Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 97–106.

Dazhi Zhang has successfully defended his PhD thesis on July 15, 2011. He received B.S. and M.S. from the Jilin University, China. He began his Ph.D study in the computer science and engineering department of the University of Texas at Arlington in 2007. His research interest is software security, especially using automatic approaches to detect program vulnerabilities.

Donggang Liu received his BS degree from the Department of Computer Science at Beijing Institute of Technology, China, in 1998, MS degree from Institute of Computing Technology, Chinese Academy of Science, China, in 2001, and PhD degree from the Department of Computer Science at North Carolina State University, in 2005. He works as an assistant professor in the Department of Computer Science and Engineering, UT Arlington. His research interests include network and distributed system security. Currently, He focuses on wireless security and system security.

Yu Lei is an Associate Professor in Department of Computer Science and Engineering at University of Texas at Arlington. He received his PhD degree in Computer Science from North Carolina State University. His research interests are in the area of automated software analysis, testing, and verification, with a current focus on software security and combinatorial testing.

Professor David Kung received his Ph.D. and his M.S. in computer science from the Norwegian Institute of Technology, Trondheim, Norway. He received his B.S. in mathematics from Beijing University, China. His research interests include object-oriented software testing, agent-oriented software engineering, and design patterns.David Kung received his Ph.D. and his M.S. in computer science from the Norwegian Institute of Technology, Trondheim, Norway. He received his B.S. in mathematics from Beijing University, China. His research interests include object-oriented software testing, agent-oriented software engineering, and design patterns.

Christoph Csallner is an Assistant Professor in the Computer Science and Engineering Department at the University of Texas at Arlington (UTA). He received his Dipl.-Inf. degree from the University of Stuttgart and his M.S. and Ph.D. from Georgia Tech. His research interests are in software engineering and program analysis.

Nathaniel Nystrom joined the University of Lugano as an assistant professor in 2011. Prior to his current position, he was an assistant professor in the Department of Computer Science and Engineering at the University of Texas at Arlington and was a postdoctoral researcher at IBM T.J. Watson Research Center in Hawthorne, NY. He received his Ph.D. in Computer Science from Cornell University in 2007. He also holds B.S. (1995) and M.S. (1998) degrees in Computer Science from Purdue University and an M.S. (2004) in Computer Science from Cornell. His research interests include programming languages, compilers, tools, and methodologies for constructing safe, secure, and efficient systems.

Wenhua Wang received the BS degree in telecommunication engineering from Huazhong University of Science and Technology, Wuhan, Hubei, China, in 2002, the MS degree in software engineering from Tsinghua University, Beijing, China, in 2005, and the PhD degree in computer science from the University of Texas at Arlington, Arlington, Texas, USA, in 2010. Currently, he is a software quality assurance engineer in Marin Software company, San Francisco, California, USA. He focuses on automated software testing and security testing. He is a member of the IEEE.