Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection

Hector D. Menendez and David Clark,

Abstract—

The utility of a test set of program inputs is strongly influenced by its diversity and its size. Syntax coverage has become a standard proxy for diversity. Although more sophisticated measures exist, such as proximity of a sample to a uniform distribution, methods to use them tend to be type dependent. We use r-wise hash functions to create a novel, semantics preserving, testability transformation for C programs that we call HashFuzz. Use of HashFuzz improves the diversity of test sets produced by instrumentation-based fuzzers. We evaluate the effect of the HashFuzz transformation on eight programs from the Google Fuzzer Test Suite using four state-of-the-art fuzzers that have been widely used in previous research. We demonstrate pronounced improvements in the performance of the test sets for the transformed programs across all the fuzzers that we used. These include strong improvements in diversity in every case, maintenance or small improvement in branch coverage – up to 4.8% improvement in the best case, and significant improvement in unique crash detection numbers – between 28% to 97% increases compared to test sets for untransformed programs.

Index Terms—System Testing, Fuzz Testing, HashFuzz, Universal Hashing

1 Introduction

Contemporary software, once it reaches a certain scale, can be too large and too heterogeneous to be amenable to formal verification and validation. Assuring correctness, security, safety, robustness, or any other property of the code in the absence of formal techniques relies on testing, that is, on a finite set of executions that the tester hopes is representative of all executions. All executions may effectively be finite but this does not help as the number is large and there is insufficient time. This is the central problem of software testing: how best to represent a population with a sample. Statistical theory has answers but it can be difficult to apply them to software. What is the probability distribution of the population? It may be very imperfectly known. Statistics says to employ the Maximum Entropy Principle [1]. Choose the probability distribution, consistent with what you know, that has the maximum entropy. In the common testing situation of knowing nothing, this will be a uniform distribution. Entropy is a measure of diversity in the distribution and this fits with the tester's intuition that the test set must be as diverse as possible – sampling from a maximal entropy distribution produces a more diverse, more widely spread sample that better represents the whole population.

However, statistical sampling from some combinations of program types is challenging. Accurately sampling from string distributions in particular is challenging, leading to the promulgation of alternative techniques such as ones based on algorithmic information theory [2]. The most widely adopted alternative to statistical methods is the best known method, the industry mandated method, and the one based on the software under test itself: coverage criteria. Semantic behaviours of the code are represented in the code syntax. Cover the syntax according to some criterion and you have some guarantee of diversity.

Software testing strategies are commonly based on coverage. This influences contemporary methods for system testing, such as fuzzers. However, there is evidence that coverage, in terms of branch or line coverage, is not enough by itself to expose faults in software [3]. Other lines of research point to the usefulness of sampling from uniform distributions by way of random-based test case generators, particularly in improving fault detection abilities [4]. While the limitations of coverage are related to lack of diversity [3], the limitations of random sampling based methods are related to their inability to cover specific paths [5].

A combination of diverse sampling and syntax coverage may produce test sets that improve the representation power, and hence diversity, of those produced by either method. This idea has been explored in the context of program testing via symbolic execution. A relevant example here is from the work of Chakraborty et al. [6], [7]. Their chief aim was to apply universal hashing, i.e. the use of independent hash functions to partition a domain, to achieve as much sampling diversity as possible, with a close to uniform distribution sample as a target (Section 2.2). The obvious limitation for this methodology is induced by the use of program constraints. As Plazar et al. demonstrated, this limitation makes them inapplicable to large programs [8]. The scalability of these systems is poor and they can hardly be applied to system testing of software [9].

We seek to improve *system testing* on larger scale programs by producing test sets that combine good coverage with input samples from a near uniform distribution. We show how universal hashing can be the basis of a testability transformation [10] for programs in the context of fuzzing.

- *H. Menendez is with the Computer Science Department of Middlesex University London.*
  E-mail: h.menendez@mdx.ac.uk
- *David Clark is with the Computer Science Department of University College London.*
  E-mail: david.clark@ucl.ac.uk

Manuscript received April 19, 2005; revised August 26, 2015.

The key to achieving near uniform test sets is to transform the program by adding new branches immediately after the input. These branches force each input to be chosen from one part of a partition on the input space induced by r-wise independent hash functions (Section 2.2), embedding the hash function partition into the behaviour of the program. This is a semantics preserving transformation that, to successfully produce near uniform test sets, relies on the fuzzer being both mutational on inputs and instrumented to seek branch coverage via a feedback loop (Section 3). This idea applies to any programming language that can leverage feedback-based fuzzers for system testing.

To this end we introduce HashFuzz, a novel testability transformation on programs that improves the test sets resulting from fuzzing them. As a testability transformation, it is agnostic with regard to the instrumentation-based mutational fuzzer employed.

To demonstrate the effectiveness of HashFuzz, we performed system testing experiments on a set of 8 known, open source projects from Google's Fuzzer Test Suite [11]. These have an average of 200,000 lines of code each. In the experimentation we used 4 state-of-the-art fuzzers: AFL, FastAFL, FairFuzz, and LibFuzzer. The before and after results demonstrate huge improvements in diversity: the test suites generated by the fuzzers on the pre transformation programs are highly non-diverse according to the L2-test, a statistical test for discrete uniform distributions [12]. After transformation, diversity is so high that each test set is close to being "sampled from a uniform distribution". This is especially true in the case of post transformation test sets produced by LibFuzzer.

The post transformation test sets found by the four fuzzers improve coverage for up to 37.5% of the programs. Particularly impressive is how the HashFuzz transformation improves the ability of the fuzzers to detect more unique crashes than they can on the untransformed programs. For 62.5% of the post transformation programs there was an up to 97% improvement in the number of unique crashes found. Each experiment on a triple of program, transformed program, and fuzzer, was performed twenty times and the numbers reported above are median values over the twenty repetitions.

The main contributions of this work are:

- We introduce HashFuzz, a novel, semantics preserving, testability transformation that improves the diversity and coverage of test sets produced by instrumentation-based mutational fuzzers (Section 3). Its implementation is also publicly available [1].
- We evaluate the effect of the HashFuzz transformation on eight programs from the Google Fuzzer Test Suite using four state-of-the-art fuzzers that have been widely used in previous research (Section 4).
- Our results show various improvements in the test sets for the transformed programs across all the fuzzers that we evaluated. These improvements are: a strong improvement in their test input diversity in every case, maintenance or small improvement in their branch coverage – of up to 4.8%, and maintenance or significant improvement in their unique crash detection numbers, of up to 97% more unique crashes than on the untransformed program (Section 5).

The HashFuzz testability transformation could be adapted to other search-based test suite generation methods, insofar as they use coverage information from the software under test.

1. The current version of the repository can be found here: https://github.com/hdg7/hashfuzz

2 Background

HashFuzz aims to diversify the behaviour of fuzzers by applying the theory of universal hashing. More specifically, our work applies the part of the theory that relates to XOR constraints. This section explains how our methodology benefits fuzzing and how universal hashing and the XOR constraints work.

Fuzzers as system testing tools have begun to dominate the state of the art of automatic test generation for security as they have discovered multiple bugs in several systems that are daily used by millions of users per hour [13], for instance, the Heartbleed bug in OpenSSL [14]. Our goal is to combine fuzzers with a program transformation that causes a fuzzer's test suite generator to become closer to a uniform generator over the whole input domain. This kind of transformation, based on uniform hashing, has shown significant improvements in bug detection in electric circuits at the level of unit testing, when they are combined with SAT or SMT solvers [6] but, until now, they have never scaled up to system testing.

2.1 Mutational Fuzzers

Mutational fuzzers work mainly on the input space [15], [16], [17], [18], [19], [20]. They aim to create diverse inputs through repeated mutations of known inputs and so identify previously unexplored branches or paths on programs. These previously unexplored branches or paths can lead to potential bugs or previously unseen crashes. Some current fuzzers, such as AFL [17] or LibFuzzer [20], combine mutation with feedback from an instrumentation process to assist them in prioritising known inputs to be fuzzed. This type of fuzzer is called an instrumentation-based fuzzer and this type is the target of our research. Other fuzzers, such as zzuf [15] or Radamsa [16], manipulate inputs based on heuristics but, by default, they do not use information extracted from the program. They are out scope for this work.

Fuzzers commonly start with a set of seeds, or inputs, together with the program under test. Instrumentation-based fuzzers also add instrumentation to the program, normally in the compilation and linking phase, to guarantee that they can receive feedback during the testing process. This feedback is usually the coverage information achieved by the generated test suite. Instrumentation-based fuzzers perform the fuzzing process in three steps: 1) they add the set of seeds into a queue, 2) each input in the queue is mutated via some heuristic, and 3) when a new input finds an uncovered path, this new input is also added to the queue, as after mutation it can yield new inputs that potentially find further new branches. A round of fuzzing

is normally terminated by the user, however, some fuzzers, such as LibFuzzer, terminate the round when they find an input that crashes.

The performance of a mutational fuzzer is affected by the selection of the initial, seed inputs for any instance of its application and, more generally, by the heuristics used by the fuzzer to achieve fresh mutations of the inputs.The heuristics might not be able to generate well formed inputs for specific paths because these are guarded by conditionals that have a required structure [5]. Diversity in the seeds can help overcome limitations in the mutational heuristics. Mostly, these heuristics mutate the inputs as binary strings. However, some fuzzers incorporate specific structure information into their heuristic. This approach has led to grammar-based fuzzers [21], [22] and dictionary-based strategies [17]. Improving the diversity of grammar-based fuzzers would be more complex than our approach as the diversity needs to be generated at the grammar level. Therefore, it is out of scope for this paper.

2.2 Universal Hashing and XORSample'

Universal hashing of test suites focuses on diversifying the automatic test generation process. This diversification is based on the uniform distribution. Assuming specific constraints for a program, the aim is to generate inputs for specific parts of the program where each possible input has the same probability of being generated, i.e. a generator following a uniform distribution. This aim is formalised as follows: considering G as a probabilistic generator for a program P, and $i \in I$ an input for P, the aim of a uniform generator G^u must satisfy:

$$p(G^u(P) = i) = 1/|I|.$$

The algorithms XORSample' [23] and UniGen [6] were created to improve the diversity of deterministic solver-based generators. They were originally applied to test electrical circuits. Their target for diversity was achieving sample sets close to the uniform distribution, the closer the better. Their limitations only allow them to create near-uniform generators. This type of generator is defined similarly to a uniform generator, but the individual probabilities are biased by a constant factor $c \in (0, 1]$, therefore,

$$p(G^{nu}(P) = i) = c(1/|I|).$$

To create the near-uniform generators, these two algorithms perform three main steps: 1) partition the original space into parts, generated by r-wise independent hash functions, 2) select one part uniformly at random and, 3) select an element inside the part uniformly at random.

The input space depends on the input variables and the alphabet of these. Let $\Sigma = \{0, 1\}$ be an alphabet, let $X = \{x_1, \ldots, x_n\}$ be variables whose values correspond to the alphabet, i.e. they have binary values, and let F_X be a function on these variables. For instance, $F_X(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$ would be an XOR operation on the variables of the alphabet. Suppose that you need to generate inputs for F_X, satisfying $F_X = true$ (or 1). An obvious input for the example would be $(x_1, x_2, x_3) = (1, 0, 0)$. Sending F_X to a SAT solver would produce this result or a similar one. Moreover, if no extra constraints are included in the solver, it might constantly provide this input or a similar one, as solvers are based on heuristics.

In general, submitting a function repeatedly to a solver produces results that are very similar over all queries [23]. This reduces diversity when generating inputs from constraints, as some valid inputs will have a higher probability than others. To deal with this problem, XORSample' and UniGen add extra constraints that force the solver to look for inputs in different sections of the input space.

This idea is based on the properties of r-wise independent hash functions. An r-wise independent hash function h, selected uniformly at random from a family of r-wise independent hash functions H, satisfies:

$$p\left[h(k_1) = v_1, \ldots, h(k_r) = v_r\right] = \frac{1}{|V|^r}, \tag{1}$$

where $k_i \in K$ are potential keys and $v_j \in V$ potential values. This means that if we apply the hash function to the variables of a function, every possible combination of values would have the same probability. Moreover, the hash is a projection to a lower dimensional space. This projection defines a partition in the original input space. One part of this partition will be selected before we create an input, and the input will belong to this part. The part is selected uniformly at random. This approach spreads the input generation process through the input space.

Our work focuses on the $\mathcal{H}_{XOR}$ family. This family is formed by any hash function that performs XOR operations on the variables. The previous example, F_X, is part of this family. Every hash function (h_i) of a family H is defined from a set of keys K to a set of values V, $h_i : K \to V$. In the case of F_X, the keys can be thought of as any binary number from 0 (000) to 7 (111) and the values, according to the function, are their parities (1 or 0). $\mathcal{H}_{XOR}$ is a family of 3-wise independent hash functions, as Gomes et al. proved in [23]. Therefore it satisfies that for any $h \in \mathcal{H}_{XOR}$, selected uniformly at random:

$$p\left[h(k_1) = v_1, h(k_2) = v_2, h(k_3) = v_3\right] = \frac{1}{|V|^3}. \tag{2}$$

In the case of $\mathcal{H}_{XOR}$, two elements can describe every function inside the family: a vector of coefficients a that multiplies each variable, and an independent element b. Therefore, the definition of every $h_i(X)$ is:

$$h_i(X) = b_i \oplus (\bigoplus_{j=1}^{n} a_i^j \cdot x_j) \tag{3}$$

If a and b are selected uniformly at random, h_i is selected uniformly at random from $\mathcal{H}_{XOR}$ [23]. Partitioning the input space requires selecting more than a single hash function, as we want to control the granularity of the partition. The higher the number of hashes we select, the smaller each part of the partition is.

Let $\mathcal{H}_{XOR}(X, q)$ be a hash function generator on the set of variables X where each variable is selected with probability q every time that we generate a hash function. If $q = 1/2$, then, every XOR constraint is chosen with probability $2^{-(n+1)}$, where n is the number of variables. As the total number of possible hash functions in $\mathcal{H}_{XOR}(X, q)$

Algorithm 1 XorSample' algorithm

Require: A formula $f_P(\vec{x})$, representing a deterministic circuit P; a natural number $s \leq n$

1: $Q_s = \{s$ constraints randomly sample from $\mathcal{H}_{XOR}(X, 1/2)\}$
2: $f_P^s(\vec{x}) = f_P(\vec{x}) \cup Q_s$
3: mc =SATModelCount(F_P^s)
4: **if** $mc \neq 0$ **then**
5: $i = sample(1, mc)$
6: Extract i elements of F_P^s with the solver
7: Select the i-th element

is $2^{(n+1)}$, by definition, their selection is uniform, as we require.

This hash function generator satisfies the first step of the input diversification process, as it partitions the original space into parts. Moreover, the uniform selection of the different h_i equations from $\mathcal{H}_{XOR}(X, q)$ corresponds to the selection of each part. Once we have the part, a SAT solver can provide every input inside it and we can select one uniformly at random. Controlling the granularity of the parts is extremely relevant to guaranteeing termination in a timely way. This whole process generates the XORSample' algorithm [23]. This algorithm (Algorithm 1) starts with the formula representing a logic circuit f_P and the value s that controls how many hash functions are selected, i.e. the granularity of the parts. Then, it selects s hash functions uniformly at random from the $\mathcal{H}_{XOR}$ family, by simply choosing the a and b coefficients of equation 3 (line 1). The hash functions are included as new constraints for f_P and submitted to a SAT solver to count the number of possible solutions that satisfy them (lines 2 and 3). Then, the index for a solution is chosen uniformly at random (i at line 5) and the solver is asked to generate solutions until this index is reached (line 6). This paper leverages this idea to embed XOR hashes in programs and generate inputs from their constraint without solvers, just by using the existing machinery of the fuzzer and a semantics preserving testability transformation on the target program (Section 3.1).

3 Universal Hashing and Fuzzers

Universal hashing improves unit testing, especially for electrical circuits [23]. Nevertheless, the necessity of transforming the programs into a set of constraints and submitting them to an SMT or SAT solver limits its ability to be extended to system testing [8]. Our methodology, called HashFuzz, extends universal hashing to system testing by replacing the solver with a fuzzer and leveraging a testability transformation to improve diversity in the fuzzer's queue.

HashFuzz is focused on instrumentation-based fuzzers, as it needs information from the program to increment measures of coverage. As the fuzzer generates test suites that cover all possible branches in the program, we divide the input space in a way that every part in the partition corresponds to a first branch of a unique path.

```
1  int main(int argc, char *argv[]){
2    int a,b,c;
3    scanf("%d%d",&a,&b);
4    if(a > 0) {
5      a=a%100;
6      c=b/a; //Potential division by zero
7    }
8    print("%d",c);
9    return(0);
10 }
```

```
1  int main(int argc, char *argv[]){
2    int a,b,c;
3    scanf("%d%d",&a,&b);
4    if(a % 100)
5      //...
6    if(a > 0) {
7      a=a%100;
8      c=b/a; //Potential division by zero
9    }
10   print("%d",c);
11   return(0);
12 }
```

Fig. 1. Example of a buggy program with a division by zero in Line 6 (top) and the same program with an extra path that alters the behavior of an instrumentation-based fuzzer (bottom).

3.1 Testability Transformation

When an instrumentation-based fuzzer creates the queue of inputs, it starts with the seeds provided by the analyst. Afterwards, every input that discovers a new branch is included in the queue (Section 2.1). For instance, Figure 1 (top) shows a program with a potential division by zero error in the sixth line. This division by zero produces a crash when $a\%100$ is zero. Suppose that the input (2,1) is a seed to this program. This input does not activate the crash and it produces the correct output. This input will be part of the queue and mutated until a new input covers the path $a \leq 0$. The final test suite will have maximum coverage on the program, but as (2,1) is the only input traversing the bug, it will neither activate the bug nor a potential observation of it.

Adding a constraint to the previous program (Figure 1, bottom) produces two new branches that divide the input space between the area where $a\%100$ is 0 and the area where it is not. The original program has two branches while the transformed one adds two more. During the fuzzing process, the fuzzer generates another input different to (2,1) that traverses the bug, as a must satisfy the new constraint. This activates it. We extend this idea to the whole input space intending to generate diverse test suites.

Our testability transformation leverages universal hashing [23] by adding branches controlled with XOR constraints at the entry point of the program. The transformation applies these constraints to the program input. Figure 2 shows an example of the testability transformation and Figure 3 shows an example of how the hashes are applied to the input. Equation 3 generates the XOR hash constraints by choosing its coefficients uniformly at random (i.e. a and b in the equation). To maintain semantics, the basic blocks associated with the branches (`if/else`) do not change any variable or path on the program, but add an extra step that

Algorithm 2 HashFuzz algorithm

Require: A program P

1: Select the coefficients a and b uniformly at random.
2: Create the 3 constraints (Q_s) using the XOR expressions of equation 3.
3: Add the Q_s at the entry point of the program as conditions for branches.
4: Apply the fuzzer.

a fuzzer understands as a new branch to cover (Section 3.3). This process partitions the input space into parts whose boundaries are set uniformly at random, similarly to XORSample' (Section 2.2). The main difference is that we embed the partition constraints directly into the program, via the testability transformation, and this requires the fuzzer's mutation heuristic to choose inputs for each of these parts. This eliminates the dependency on solvers of previous work but keeps the near uniform capability of the original solver approach.

HashFuzz performs four steps (summarized in Algorithm 2):

1) Choose the form of input for the program, normally the standard input.
2) Select the hash by choosing random values for the coefficients a and b in equation 3. This partitions the whole input space into disjoint parts.
3) Set the hash at the entry point of the program, so not to affect program execution. This creates $2n$ new branches in the program, where n is the number of hash equations. We select n as 3 as the XOR hash functions are 3-wise independent [23]. This creates the smallest possible parts that keep the 3-wise independence property but does not overload the program significantly.
4) Submit the transformed program to a fuzzer. The modification on the branches causes the fuzzer to tend to look for inputs in different parts when covering branches. The queue conserves these inputs as each part is associated with a new branch. The queue's inputs generate new inputs covering different regions of the input space.

It is a common practice to repeat the fuzzing process several times, as in Klees et al. [13]. Our methodology benefits from this practice when the hashes are reset in every repetition. This changes the parts and, as a consequence, the fuzzer finds other inputs in different areas of the space. Therefore, different executions of HashFuzz generate different test suites, reducing collisions among them, and compensating for the diversity bias introduced by the mutational heuristics.

It is important to remark that HashFuzz can create more than three branches, but it needs another r-wise independent hash function satisfying that the r value is greater than 3 in order to meet the theoretical requirements.

3.2 Implementation

Although HashFuzz is program and language agnostic, for experimentation we used programs in C and C++. Different fuzzers leverage different methodologies to introduce inputs to C-family programs, where the most common is the standard input (`stdin`).

```
1  int main(int argc, char *argv[]){
2    int a,b,c;
3    if(XORhash1(stdin))
4      //...
5    restablish(stdin)
6    if(XORhash2(stdin))
7      //...
8    restablish(stdin)
9    if(XORhash3(stdin))
10     //...
11   restablish(stdin)
12   scanf("%d%d",&a,&b);
13   if(a > 0) {
14     a=a%100;
15     c=b/a; //Potential division by zero
16   }
17   print("%d",c);
18   return(0);
19 }
```

Fig. 2. Example of instrumentation for the program in Figure 1. Every `if` also has an `else` associated. Both generate two new basic blocks.

We consider the input provided by the fuzzer as the hash target. For AFL-based fuzzers, this is the `stdin` and, for LibFuzzer, it is the `data` input of the `LLVMFuzzerTestOneInput` function. We modify each program by adding three XOR hashes created by equation 3 (see Section 2.2). We can not control the input size, as the fuzzer generates different inputs of different sizes, therefore our hashing selects coefficients for one byte and applies them to an input byte by byte along its length (Figure 3). We consider the `main` function as the entry point and we set the hashing immediately after the variable declaration. Figure 2 shows an example of Figure 1's program instrumented.

In this example, we can see that every `if` creates two branches. On the first one, the hash is different to zero, while, on the second (`else`), it is 0. Following equation 3, it is easy to see that the only possible values of the XOR equations are 0 and 1. As a simple optimisation, the assignment of the coefficients to byte-level can be replaced by a combination of XOR operations on the AND between the coefficients and each byte of the input. For instance, if we consider a hexadecimal input such as $0xFFEECC$ and a hash with coefficients $0xAA$, the hash operation according to equation 3 is equivalent to:

$$XOR(FF\&AA) \oplus XOR(EE\&AA) \oplus XOR(CC\&AA),$$

where the XOR function applies the operation to the whole byte at bit-level. This reduces the workload of the operations as the hash performs simple byte operations. There are three hash functions applying these operations. If the result of one of these operations is 0, the input will be in a part of the input space, while, if it is 1, it will be in a different part. This allows us to partition the input space into 8 parts, depending on the results of these hashes (Figure 4, top). The branches will force the fuzzer to keep up to one input for each part into the queue because they are exposing new transitions between branches (Section 3.1).

During the process, we use the `stdin` as a file, setting it to its beginning every time we reach the `EOF` symbol. In this way, we do not need information about the size of the

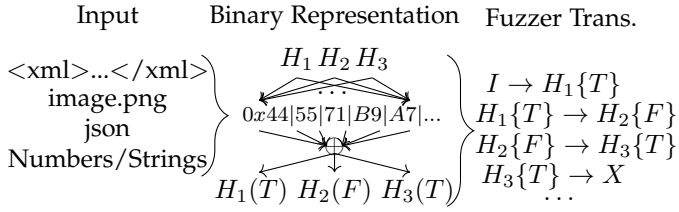


Fig. 3. Example of the application of the hash functions to an input. The hashes consider the binary representation of the input, therefore they can be applied to any kind of input. Once the hashes have been evaluated on a specific input, byte by byte, the branches associated with the hashes provide new transitions for the fuzzer to cover.

input, and we can hash it while we are reading it. The code inside the `if/else` statements needs to be simple enough to not produce an overload of the execution, but complex enough to guarantee that the compiler does not optimise it. To address this, our implementation uses a static counter, which is an atomic operation that is not optimised by the compiler, and it does not produce an overload. After the program modifications finish, the program is submitted to the fuzzer, which generates a test suite for it.

3.3 Fuzzer's Behavior

After the testability transformation, the fuzzers create inputs to cover the new branches. The number of inputs that they include in the queue is higher than those included in the original program queue. For instance, the example of Figure 2 includes the seed, that we assume is (2,1), and up to 8 more inputs for a branch-based fuzzer, while the original program in Figure 1 includes up to 2 inputs including the seed.

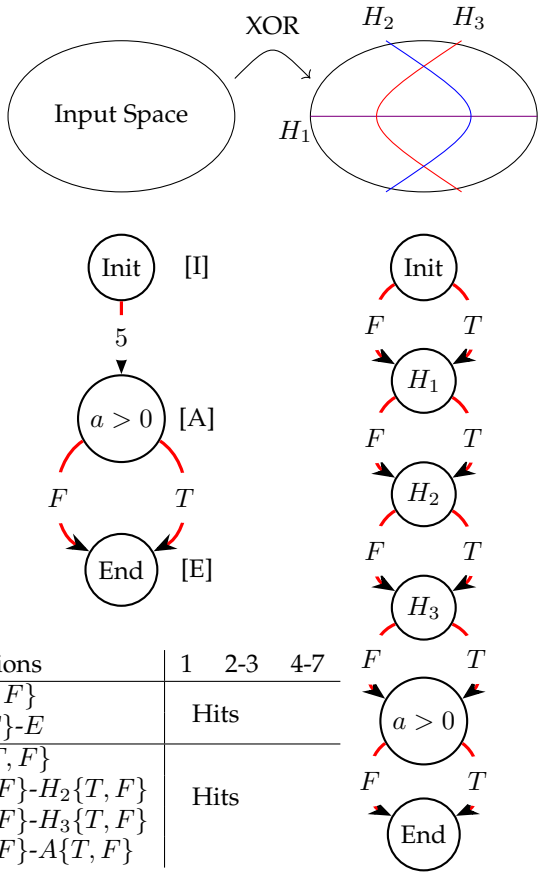
To make sure that the diversification is not producing a bottleneck in a specific area of the program, we have set it at the beginning. Considering that the operations do not add a lot of computation to the execution, as they are fast, the main problem is in the exploitation of these solutions.

The exploitation depends on the fuzzer's logic, but fuzzers commonly increment the number of branches visited. Inputs traversing the new branches set at the beginning remain in the queue if they visit these branches for the first time. They also belong to different parts of the input space. Although not all the parts may be visited, reapplying HashFuzz before the next fuzzing iteration creates new parts, improving diversity each time the fuzzer is executed.

For AFL-based fuzzers [17], [18], [19], for instance, every input that adds a new transition stays in the queue. These transitions work as follows. Imagine a trace of Figure 1's example, whose control flow graph is represented in Figure 4 (left). The two possible sequences of blocks and their associated transitions are:

$$I->AT->E \quad [I-AT, AT-E]$$
$$I->AF->E \quad [I-AF, AF-E]$$

Once we introduce the hash functions, as in Figure 2, there are new transitions on the transition table (Figure 4). These transitions correspond to parts of the partition induced by the hashes. The parts are disjoint sets. When an input traverses, for instance, H1T → H2F → H3T, it is in a different



Transitions	1	2-3	4-7
I-$A\{T,F\}$ $A\{T,F\}$-E	Hits		
I-$H_1\{T,F\}$ $H_1\{T,F\}$-$H_2\{T,F\}$ $H_2\{T,F\}$-$H_3\{T,F\}$ $H_3\{T,F\}$-$A\{T,F\}$	Hits		

Fig. 4. Example of the branch modification produced by the program transformation applied to Figure 1. The control flow graph of the original program is on the left while the equivalent on the transformed program is on the right. The table represents the transition hits of an AFL-based fuzzer. The transitions are summarised. $I-A\{F,T\}$ summarises transitions $I-AT$ and $I-AF$.

part to another input traversing H1T → H2T → H3T. AFL includes the transitions on its table, and their related inputs on its queue. The inputs related to these transitions diversify future inputs of the program as they are in different parts of the input space.

The fuzzers prioritise inputs by discovered branches using different strategies. The most common strategy, used by AFL [17] and LibFuzzer [20], is direct exploitation, where the queue is consulted in order. Nevertheless, AFLFast [18] and FairFuzz [19] use different strategies to this. Both tools aim to maximise rare paths to identify more crashes. As our diversification is set at the beginning, this selection makes sure that the fuzzer follows as many parts as possible initially, guaranteeing stronger diversity in the test suite from the very beginning. This leads to a better exploration of the input space.

4 Experimental Setup

HashFuzz improves diversity during the fuzzing process, by forcing exploration of different regions of the input space. This process improves crash detection and has a low negative impact on coverage. Universal hashing has to date only been shown to be effective on solvers and electric circuits [6]. Our research focuses on the implementation and

evaluation of this technique on fuzzers and significantly large programs. Our main research questions are:

RQ1: Does HashFuzz produce diverse test suites whose distribution is close to a uniform distribution? The original work in this area shows that algorithms such as XorSample' or UniGen are near-uniform [6], [23]. We evaluate whether our testability transformation affects the diversity of the test suite generated by the fuzzer. We measure whether the inputs generated are following a distribution close to the uniform distribution. We apply the L2-test for uniformity which estimates the distance between a sample set and a discrete uniform distribution based on number of collisions [12].

RQ2: Does HashFuzz affect coverage? Using the llvm-cov tool, we compare the coverage achieved by each pair of generated test sets produced by a given fuzzer on each original program / transformed program pair. This is done for each of the four chosen fuzzers run on each of the 16 programs in the eight pairs with each fuzzer run on a program for 24 hours.

RQ3: Does HashFuzz detect new unique crashes on the programs? We execute the same scheme as in RQ2, but this time we focus on the unique crashes of the program. As different fuzzers and the HashFuzz process can modify the uniqueness of the crashes we only consider crashes of the original program before transformation and instrumentation. Therefore, we take the set of unique crashes generated by each fuzzer and **reduce them based on the original program**, using the minimisation process of `afl-cmin`. This minimisation forces the removal of every crash that does not introduce any unique edge in the original program, which is the maximal minimisation of the tool. This also removes every false alarm that a fuzzer may generate. Based on this, we can check whether our process finds new unique crashes that have not been found before in the original program.

4.1 Dataset and Fuzzing tools

The evaluation of HashFuzz has been performed on programs from the Fuzzer Test Suite of Google [11]. This dataset contains 24 open-source programs some of which have known crashes. We selected eight programs randomly, the number of programs and fuzzers chosen as a sweet spot between generality and computation effort. Experiment effort for a single CPU over all experiments was 1,280 computation days, corresponding to 24 hours per iteration, 20 repetitions, four fuzzers and eight program pairs. Fortunately, we were able to execute eight fuzzer-program pairs in parallel at a time, with the number of elapsed days needed being 160. Table 1 shows the size of the programs in terms of lines of code. As several of these programs contain multiple languages, we specifically show the number of lines of code in C and C++, which are the ones instrumented by the fuzzers. The average number of lines of code is 201,433.

We apply the testability transformation at the entry point of each program. After, we compile both the original and the transformed program and execute each fuzzer on both. The selected fuzzers for the evaluation are:

- **AFL** [17]: American Fuzzy Lop. It instruments the program under test to measure branch coverage statistics during the test generation process. It mutates inputs using several different heuristics to maximise branch coverage and to detect unique crashes and hangs.
- **LibFuzzer** [20]: this fuzzing library is embedded in the program via the SanitizerCoverage. It uses a specific target function called `LLVMFuzzerTestOneInput` and produces inputs for this function. During the generation process, it mutates these inputs, similarly to AFL.
- **AFLFast** [18]: AFLFast extends AFL to activate low-frequency paths, so detecting more behaviours of the program. This tool improves the selection of seeds to be mutated and includes a power schedule to control the number of times a seed is fuzzed.
- **FairFuzz** [19]: This modification of AFL focuses on guiding the creation of inputs so as to exercise rare branches during the generation process. It performs both seed prioritisation and heuristic selection to guide its input mutations.

Project Name	Total LoC	Total LoC in C/C++
C-ares	137,064	118,369
LibPng	75,928	28,241
LibXml2	452,203	230,232
OpenSSL101f	377,328	278,351
OpenSSL102d	442,864	304,904
Pcre2	89,932	70,161
RE2	32,621	28,267
Woff	3,525	3,434

TABLE 1
The number of lines and the number of instrumented lines for the subject projects.

For a fair evaluation of the fuzzers, we have provided them with the same set of initial seeds and the same amount of time, following the advice of Klees et al. [13]. All of them use the default configuration. In the case of LibFuzzer, which stops when it finds a crash, we set a loop that restarts the fuzzing until it reaches the time limit. In every iteration of this loop, the test suite found remains as seeds for the repetition. Each fuzzer ran on 3 cores of a 24 core machine, with no memory limitation apart from the physical machine limitation (128Gb of RAM). Due to the stochastic nature of this process, we have repeated each experiment 20 times, and we report the median values for coverage and unique crashes.

5 Evaluation

HashFuzz improves diversity of the test suites generated by fuzzers by applying a semantics preserving testability transformation. One of our evaluations of HashFuzz measures the improvement in diversity as proximity of the fuzzers' test sets to a uniform distribution (Section 5.1). Another measures the effect of the transformation on branch coverage (Section 5.2) and the third evaluation measures how it increases the detection of unique crashes (Section 5.3).

Methodology	Espsilon Values		
	0.1	0.05	0.01
AFL	0.00%	0.00%	0.00%
AFLFast	0.00%	0.00%	0.00%
FairFuzz	0.00%	0.00%	0.00%
LibFuzzer	0.00%	0.00%	0.00%
Hash + AFL	100%	87.5%	75.0%
Hash + AFLFast	100%	87.5%	87.5%
Hash + FairFuzz	100%	75.0%	75.0%
Hash + LibFuzzer	100%	100%	100%

TABLE 2
Percentage of programs that pass the L2-test for uniformity before and after applying the hashing process. The epsilon values measure the different levels of proximity to the uniform distribution.

5.1 Uniformity of the inputs

As we explain in Section 2.2, the notion of diversity that underpins and motivates the invention of the HashFuzz program transformation is the proximity of a sample set, in this case a fuzzer's test suite, to a discrete uniform distribution. We perform this measure of proximity by applying the L2-test [12]. The L2-test is a collision-based statistical test that compares the self-collisions (repetitions) occurring in a sampling *process* with repetition to determine, based on a mathematically calculated threshold, how far a distribution is from a uniform distribution. The sampling process requires a minimum number of instances to test a chosen threshold distance from uniform with a given confidence level (95% in our case). In the literature, the minimum distances are usually in the 0.5 - 0.1 range, which require up to 6,000 samples to test. In our case, as the fuzzers provide a large number of inputs in the queue, we can test for distances up to 0.01-far from uniform. For these measurements, we need to generate 12,000 inputs for 0.05 and 18,000 inputs for 0.01, according to Lemma 5 of [12], which bounds the number of samples for specific values of epsilon, the distance parameter..

Table 2 shows the results of the L2-test on test suites generated by the fuzzers before (top) and after (bottom) HashFuzz. Each row measures the percentage of programs whose test suites pass the test for 0.1, 0.05 and 0.01 distance from uniform. As a measure of diversity, the uniformity is showing that, after hashing, every program is, at least 0.1-far from the uniform distribution, which is the limit normally analysed in the literature [12]. For the fuzzers based on AFL, we obtain a good amount of programs that are, at least, 0.05-far and 0.01-far (between 75 and 87.5%). In the case of LibFuzzer, the amount is maximum for every distance threshold.

RQ1: *HashFuzz always produces test suites at least 0.1-far from the uniform distribution, according to the L2-test. Combined with LibFuzzer, the level of uniformity is a maximal up to 0.01-far from uniform.*

5.2 Coverage of HashFuzz

We want to measure the coverage reached by the original fuzzers and their coverage after the program transformation. Although coverage is not a guarantee to detect bugs, as several authors have discussed in recent work [3], it is relevant for us in order to measure the trade-off between the potential coverage lost of stressing the same branches several times, and the diversity improvement produced by the universal hashing process (Section 3).

Table 3 shows the branch coverage achieved by the different fuzzers before and after the application of HashFuzz. The table shows that the different fuzzing strategies are reaching similar coverage results in almost all cases. LibFuzzer is the main outlier. After the transformation, there are a few cases where the coverage is higher, especially for the tools based on AFL, and fewer cases where it is significantly lower, according to the Wilcoxon test. The global overall improvement on coverage is small: 4.8% for AFL, 2.3% for AFL-Fast, 1.9% for FairFuzz and 3.5% for LibFuzzer. In order to evaluate whether HashFuzz would improve the coverage for a specific technique, we applied the Vargha-Delaney effect size [24]. This measures in which percentage HashFuzz improves the normal fuzzing campaign. If the value is smaller than 0.5, HashFuzz improves the campaign, otherwise, the campaign is better without it. As an average for all the case studies, we can see that HashFuzz is improving the coverage during the campaign, although the improvement is small: 0.484 for AFL, 0.492 for AFL-Fast, 0.492 for FairFuzz and 0.484 for LibFuzzer. In general the measure is 0.483 for coverage. These improvements in coverage show that our diversity method is not producing a trade-off with coverage, as we might have expected.

RQ2: *HashFuzz has no negative effects on branch coverage. On the contrary, HashFuzz improves branch coverage by up to 4.8%.*

5.3 Crashes Exposed by HashFuzz

This final experiment aims to measure whether HashFuzz is able to identify new unique crashes in programs. Table 4 shows the number of unique crashes that the fuzzing tools identify both before the program transformation (O) and after (H). Each fuzzer runs for 24 hours using the same resources of memory and CPUs as the others (Section 4.1). The results show that the transformation always obtains the same or better results than the original program. When the original number of crashes is low or none, as is the case for the programs c-ares, libpng and openssl101f, the transformation makes no improvements. In the case of libpng, the bug reported in the Google database is a timeout, instead of a crash. This timeout has an effect on the multiple executions of the fuzzers, and it also produces low coverage (Section 5.2). The crashes of c-ares and openssl101f are found in less than 10 minutes. In the case of c-ares, the fuzzer can not explore the program more deeply as its execution is stopped by the crash.

When the fuzzer can explore more deeply into the program, as is the case of libxml2, openssl102d, pcre2, re2 and woff, the number of crashes found is affected by the fuzzers strategy. AFL and LibFuzzer, for instance, are the fuzzers that generally discover fewer crashes in these programs except in the case of pcre2 for AFL and re2 for LibFuzzer. FairFuzz obtains really good results on LibXML2 and better than the others on Woof2 as a consequence of its ability to interact with structured data [20]. Also, these two techniques obtain better results than LibFuzzer and AFL on openssl102d.

Dataset	AFL (O)	AFL (H)	AFL-Fast (O)	AFL-Fast (H)	FairFuzz (O)	FairFuzz (H)	LibFuzzer (O)	LibFuzzer (H)
C-ares	139	139	139	139	139	139	139	139
LibPng	1192	1192	1192	1192	1194	1192	621	†▲1566
LibXml2	9855	†▲10417	10062	‡▼9527	9718	†▲9899	9527	9527
OpenSSL101f	6342	†▲7022	6440	†▲7022	7048	▼7022	6445	6445
OpenSSL102d	1446	1446	1446	1446	1446	1446	1197	†▲1446
Pcre2	8044	▼8015	7675	†▲8029	8003	▲8103	7809	7805
RE2	4407	†▲4790	4407	†▲4790	4407	†▲4796	4837	4837
Woff	2007	2007	2009	2007	2007	2007	2185	▼2160
Improvement		4.8%		2.3%		1.9%		3.5%
Vargha-Delaney		0.484		0.492		0.492		0.484

TABLE 3
Median branch coverage achieved during the fuzzing process divided by different fuzzing techniques and programs. The ▲symbol shows that there is a statistically significant improvement between the original execution of the fuzzer and its execution after applying the program transformation, according to the Wilcoxon test with a p-value smaller than 0.05. The ▼symbol shows the opposite. The † symbol indicates when the Vargha-Delaney value favours HashFuzz (VD value lesser than 0.45), while the ‡symbol shows the opposite (VD value greater than 0.55).

Once the testability transformation is applied to the programs and the fuzzing process is executed, the resulting test suites show improvements, especially for libXML2, openssl102d and pcre2. Also, when there are no improvements, the results are the same as for the test suites obtained when fuzzing the original program. The improvements on AFL after the crash minimisation process are: 8 times more unique crashes for libXML2, 50% more for openssl102d and 54% more for pcre2. Counting the total number of unique crashes, the improvement for AFL is 55.4% higher. In the case of AFL-Fast, we notice that it produces improvements in more programs, as this one also improves test suites for c-ares and woff2 after transformation, but, percentage-wise, they are less significant (100% for c-ares, 43% for libXML2, 20% for openssl102d, 55% for pcre2 and 50% for Woff). The total improvement for AFL-Fast is 52.9%. The application of HashFuzz on programs for FairFuzz improves test suites for the same programs as on AFL but these improvements find the maximum number of crashes found in the available time for LibXML and openssl102d. The improvements are 30.8% for LibXML2, 36.4% for openssl102d and 106% on pcre2. The total improvements for FairFuzz are 96.7%. Finally, for LibFuzzer the improvements are focused on openssl102d (33%), pcre2 (8%), re2 (67%) and woff (200%). This fuzzer is the only one that detects unique crashes in re2. The total improvement, in this case, is the lowest (27.9%).

We also applied the Vargha-Delaney measure to the fault detection process. As an average among all the case studies, we were able to see that the results are better for fault detection than for coverage. For AFL the measure is 0.438, for AFLFast is 0.438, for FairFuzz 0.461 and for LibFuzzer is 0.414. The general improvement is 0.457, which shows that HashFuzz is either the same or better in the 54.3% of the cases.

RQ3: *HashFuzz either improves the number of unique crashes found by the test suites or maintains it, depending on the depth achieved in the fuzzing process. These improvements on the number of unique crashes range from 28% up to 97%.*

6 Discussion and Limitations

HashFuzz improves test suite diversity for fuzzers by applying a semantic preserving testability transformation. This approach follows the idea of Chakraborty et al. [6], when they were facing the adversarial behaviour of solvers during the input generation process. Our results show that the fuzzer skills also improve in terms of crash detection and coverage for several cases when diversity is embedded within the generation process. This is similar to the results of Böhme et al. [25] when they leverage entropy to increase the diversity of behaviours during fuzzing campaigns. Therefore, r-wise independent hash functions compensate the adversarial behaviour of fuzzers and solvers during the input generation process by improving the fuzzers diversity.

In terms of scalability, HashFuzz depends on the based fuzzer. It only includes more diversity by selecting proper seeds at the beginning, but it has no negative influence in coverage or fault detection (Section 5).

The fuzzer's logic affects our testability transformation in terms of uniformity (Section 5.1). LibFuzzer follows a path-based strategy to measure coverage while AFL follows a transition-based one. We can see that diversity is more effective in the first case than in the second. Also, those fuzzers that are targeting specific kinds of paths (such as rare paths) are more resistant to diversity that those that do not focus on specific kinds of targets. This is the case of FairFuzz.

To validate HashFuzz, we applied the Vargha-Delaney effect size measure [24] to compare the effectiveness of the transformation under different projects of different sizes in the same amount of time and resources.

In our experiments, the Vargha-Delaney measure is 0.483 for coverage (Table 3) and 0.457 for fault detection (Table 4), indicating that the normal fuzzers performed worse than HashFuzz for both, because 0.48 and 0.457 are less than 0.5. We can also interpret the fault detection value as: HashFuzz will work better by detecting faults 55% of the time, while in the other 45% it will behave in the same way as a campaign without the testability transformation.

In terms of analysing how the coverage and fault detection are affected by uniformity, we also compare it with the Vargha-Delaney measure. This shows that, when the uniformity is high (Table 2) the Vargha-Delaney will be more significant, as it is in the case of LibFuzzer for both coverage and fault detection. Even if the improvement in general numbers is not higher, HashFuzz will improve more projects than the normal campaign.

In terms of coverage, there are no significant penalizations after applying HashFuzz. On the contrary, it improves

Dataset	AFL (O)	AFL (H)	AFL-Fast (O)	AFL-Fast (H)	FairFuzz (O)	FairFuzz (H)	LibFuzzer (O)	LibFuzzer (H)
C-ares	1	1	1	▲2 (100%)	1	1	1	1
LibPng	0	0	0	0	0	0	0	0
LibXml2	1	†▲8 (700%)	7	†▲10 (43%)	13	†▲17 (31%)	5	5
OpenSSL101f	1	1	1	1	1	1	1	1
OpenSSL102d	8	†▲12 (50%)	10	▲12 (20%)	11	†▲15 (36%)	6	▲8 (33%)
Pcre2	310	†▲478 (54%)	253	†▲391 (55%)	213	†▲439 (106%)	24	▲26 (8%)
RE2	0	0	0	0	0	0	3	†▲5 (67%)
Woff	2	2	2	†▲3 (50%)	3	3	3	†▲9 (200%)
Improvement		55.4%		52.9%		96.7%		27.9%
Vargha-Delaney		0.438		0.438		0.461		0.414

TABLE 4
Median unique crashes detected during the fuzzing process divided into different fuzzing techniques and programs. The ▲symbol shows that there is a statistically significant improvement between the original execution of the fuzzer and its execution after applying the program transformation, according to the Wilcoxon test with a p-value smaller than 0.05. For those cases where there is a significant improvement, the results also provide the improvement percentage. The † symbol shows when the Vargha-Delaney value favours HashFuzz (VD value smaller than 0.45).

coverage in the same amount of time. Therefore, adding diversity will help to discover new branches within programs faster (Section 5.2) and also new crashes (Section 5.3). In some specific cases the number of branches discovered are reduced (normally one per fuzzing campaign), but the general trend improves the covered branches. The most interesting case is the combination of HashFuzz and AFLFast when applied to LibXML2. In this case, our combination exposes on average 500 fewer branches than AFLFast alone. However, in terms of reported unique crashes, the campaign detects 3 more unique crashes. It is possible that the exploration/exploitation tradeoff had been altered by using HashFuzz together with AFLFast thus finding new branches that are more interesting in terms of exposing bugs than the ones that AFLFast discovered without the testability transformation. On further investigating the LibXML2 and AFLFast result, we first noted that LibXML2 coverage could have been significantly improved with a longer campaign. This was also apparent in the experimental work of Böhme in his paper on the connections between biological species coverage and fuzzing campaigns [26]. Böhme included LibXML2 as a SUT in his experimental effort and, using longer campaigns than we did, discovered a large jump in coverage for LibXML2 once the number of discovered branches passes around 6,000. Also, he witnessed coverage growth showing strong variation in each iteration. We concluded that the 500 branch difference is likely to be statistical noise in the median values of the 20 repetitions with and without HashFuzz. This left the question as to why the combination with HashFuzz discovered three faults that AFLFast did not. We noted that we used the dictionary option with all the fuzzers in every campaign as keeping some extra records at the beginning, as HashFuzz does, improves diversity during the construction of complex XML inputs. Nonetheless, these inputs do not necessarily activate more branches but further explore the ones already discovered. The generation of these extra, exploratory inputs on paths has in this case found three memory access faults. Of course, these faults could also be discovered by the fuzzers anyway, given a longer campaign, but HashFuzz induced diversity allows them to be discovered more efficiently.

Finally, one interesting outcome for HashFuzz is related to the uniqueness of the crashes. In the case of c-ares, the second unique crash can only be discovered if HashFuzz is applied; for LibXML and OpenSSL102d, 4 crashes can only be exposed by HashFuzz; for pcre2, around 100 unique crashes are only discovered when HashFuzz is applied and for woff 6 crashes are only exposed when we apply HashFuzz. These crashes are all memory violations that the inputs activate. They are not associated with any specific property of HashFuzz. We assume that adding diversity or giving more time or iterations to the fuzzers, they would also be able to discover them at some point. HashFuzz only tends to accelerates this process.

6.1 Threats to Validity

Our study shows the influence of augmenting diversity during the process of fuzzing. Although our testability transformation applies diversity at a binary level, which makes it applicable to any kind of program, it lacks information about the input's structure. This information will help to include other kinds of diversity which will help to discover new bugs in programs. Another internal threat to our study is the generalisation. We have been limited by our resources and performed fuzzing campaigns on 8 projects during 20 runs for 24 hours, covering 1,280 computational days. Although more projects could have been evaluated, the selected projects cover different aspects in terms of lines of code (Table 1), branches (Table 3) and crashes (Table 4) giving us enough information to evaluate the effects of diversity in fuzzing. In terms of experimental replication, we have provided the code for the experiments and the data has already been published. Since our comparison is based on performance, using different machines will affect the final results. The results might also be affected if the fuzzers add a new diversity strategy, as was recently the case for LibFuzzer, that now includes the "Entropic" extension, improving diversity during the queue selection process [25].

The external threats of validity depend on the fuzzers and their limitations. During our experiments, we can see that the fuzzer selection influences significantly in the campaign's performance. Some fuzzers are more effective than others. Our testability transformation adds diversity to the fuzzer generation process, therefore if this is already implemented, we assume that the transformation will have no influence on the performance, although we have not found any fuzzer with this attribute during our experimentation.

7 Related Work

In what follows we discuss the influence of diversity in software testing (Section 7.1) and the relevance of current fuzz testing methodologies (Section 7.2).

7.1 Diversity and Uniformity in Software Testing

Our work uses the uniform distribution as our target for diversity. This distribution is, by definition, the distribution that provides the same probability to each possible event [1]. It connects with the concept of entropy, from an Information Theory perspective, as the maximum entropy of a random variable is only reached when its probability distribution is uniform (Theorem 2.6.4 of [1]).

Diversity can complement other criteria for testing, especially coverage [4], [27], [28], when the target is fault detection. There are several works applying diversity to test suite generation [4], [6], [7], [29], [30], [31], [32], [33] or selection [2]. Diversity complements the lack of exploration that specific tools suffer, such as solvers [6]. One of the first and most significant works on diversity was Adaptive Random Testing (ART) [4]. ART was a diversification methodology on random testing, intending to spread the inputs all around the input space. To reach this goal, Chen et al. applied a uniform sampling of the space, instead of using pseudo-random generators, and reported a significant improvement in the results. Our work combines diversity with coverage and extends the that of Chakraborty et al. [6] on universal hashing applied to electronic circuits and SAT solvers. Our main aim is to achieve good scalability on software. Solvers do not scale to large programs, as papers on symbolic execution have demonstrated [9]. Moreover, as Plazar et al. showed recently [8] universal hashing algorithms fail to scale to large systems and "more work is required" to deal with the trade-off between uniformity and scalability. Nevertheless, as we explained in Section 3, the idea of universal hashing can be adapted to a search-based scenario.

There are also other diversity-based approaches focused on different aspects of the programs that have demonstrated potential for detecting different kinds of bugs. Good examples are those focused on output diversity. For instance, the work of Alshahwan and Harman [34], [35] introduced an adequacy criterion based on the uniqueness of outputs to generate test suites whose outputs would be as diverse as possible. Nevertheless, they did not focus on the same concept of diversity as our current work, as their diversity criterion is the total count instead of spread and uniformity of outputs. Another good example is the work of Matinnejad et al. who introduced similarities between output signals for Simulink models, to diversify these signal via search [36].

The definition of diversity significantly affects the quality of the work, for instance, in Matinnejad et al.'s work they define a similarity measure [36]. The definition of diversity is not clear in the literature, therefore, our work has focused on finding a consistent definition inherited from information theory. In information theory, diversity is usually connected either with the Kolmogorov complexity or the entropy, which are themselves related. From a Kolmogorov complexity perspective, diversity is practically measured via the normalised information distance (NID) [32], or its computable version: the normalised compression distance (NCD). An effective, practical measure of test suite diversity is diameter of a test suite, the equivalent of an average NCD distance between the tests [2]. Maximising this diameter is equivalent to maximising the diversity in terms of Kolmogorov complexity. On the other hand, entropy directly connects with the uniform distribution, and some papers aim to maximise the diversity of test suites via entropy maximisation [33] or uniformity [6]. We follow this last definition, as we consider a test suite generator diverse when the entropy of its test suites is maximum or its generation probability distribution is uniform.

Our main concern once the diversity measure was chosen was to select a proper method to estimate it. Our methodology for creating entropic generators descends from Chakraborty et al.'s [6], as we include universal hashing, but the evaluation of these generators leverages uniformity tests (Section 5.1). This kind of evaluation is novel with respect to other state-of-the-art methodologies in uniformity which use visual evaluation [6], [7], [23], [31]. We selected the L2-test, a collision-based test that can deal with discreet and continuous spaces in tandem, which is the normal scenario for program inputs [12]. Although there are other different statistical tests for uniformity as Marhuenda et al. collected in [37], these are not suitable for our problem and they can either measure continuous or discrete spaces, but not both together. The main problem of collision-based tests is that they require a significant number of samples for performing the test [12], nevertheless, fuzzers, as we have shown, were able to generate enough inputs in the time provided to pass or discard the test (Section 5.1).

Our methodology can be considered as "diverse by construction", as our testability transformation forces the diversity during the test suite generation process. A similar idea is GödelTesting. Poulding and Feldt [29] introduced this technique as a methodology to manually create test suite generators that are parametrised and then select suitable parameters to create diversity. In contrast with this technique, we do not need a manual generation process as the introduction of the uniform hashing constraints together with the input mutation is automatic (Section 3). This idea extends our previous work on output diversity [38] and focused testing [39], where we used a solver to generate the inputs. In this case, the solver does not limit our generation process due to we are replacing it with fuzzers.

7.2 Automated Test Generation and Fuzzers

Automatic unit test case generation has classically leveraged different strategies such as symbolic execution, model-based testing, adaptive random testing, search-based testing and combinatorial testing [40]. The common goal of these techniques is finding bugs and, normally, the common strategy is coverage. Nevertheless, some of these strategies, such as symbolic execution, lack scalability, while others, such as search, require major levels of sophistication to exploit their potential.

The work of Shamshiri et al. [41] has shown these limitations by comparing the fault detection abilities of three different unit test case generation tools (EvoSuite [42], [43], Randoop [44] and Agitar [45]). Shamshiri et al. showed how the tools struggle to find more than 40.1% of the bugs

when they work independently and more than 55.7% when they work together. This work is a good example of how coverage is not enough. Even when a bug is covered the error might fail to propagate to the observation point [46]. This phenomenon, known as failed error propagation, is a consequence of coincidental correctness of the program observation, and it is covered in several papers in the literature [46], [47], [48], [49], [50].

From a system testing perspective, fuzzing is becoming a predominant area because of the fuzzers' abilities to discover new bugs and crashes in systems in a timely way. There are several kinds of fuzzers, where the predominant are either purely black box [15], modifying a set of inputs called seeds to generate new ones, or greybox [17], which also instrument the program to select the next input from those that have discovered new paths or properties during their execution. Although fuzzers are powerful as testing tools, they need to deal with particular problems such as the selection of initial seeds, the search process that is guiding them and the mutation operations applied to the inputs [13].

There are several fuzzers currently available. From the pure black testing side, the most famous are zzuf [15] and Radamsa [16]. Some of the instrumentation-based fuzzers –or greybox ones– that are famous are AFL [17], AFLFast [18], FairFuzz [19], LibFuzzer [20] and VUzzer [51]. Fuzzers such as AFLFast and FairFuzz are based on AFL and they try to improve the fuzzer quality solving problems such as seed prioritization in the case of AFLFast and heuristic selection in the case of FairFuzz. New fuzzers like QSYM [5] or Eclipser [52] apply concolic execution to detect inputs that can traverse difficult branches. Fuzzers like SLF [53] aim to solve the problem of invalid seeds, while others, like directed greybox fuzz [14], focuses the fuzzing process on specific areas of the program.

Modern approaches to fuzzing aim to perform specific tasks that are challenging for automatic test generation methods. Some examples are the work of Liang et al. [54] applying directed greybox fuzz to cover different sequences of programs, Nilizadeh et al. who apply differential fuzzing for side-channel analysis [55], Cerebro [56] that fuzzes the program's context to detect vulnerabilities, ContractFuzzer [57] that detects vulnerabilities on smart contracts or DeepHunter [58] and SeqFuzzer [59] that fuzz deep learning algorithms.

Fuzzing evaluation and comparison requires a proper setup. To unify the criteria of fuzzing evaluation, Klees et al. [13] provided some suggestions for a fair experimental comparison methodology. We used these suggestions to prepare our experimental set up as described in Section 4. To the authors' knowledge, our work is the first approach that combines universal hashing and fuzzers.

8 Conclusions

This work presents HashFuzz a novel, semantics invariant, testability transformation on programs based on r-wise independent hash functions. HashFuzz improves the diversity of test sets produced by mutational, instrumented fuzzers that use branch coverage feedback. The transformation uses the XOR hash family to constrain the fuzzer to find inputs from different regions of the input space. These regions are defined uniformly at random by the hashing selection process which is embedded after the program's input by the transformation.

We demonstrate experimentally that transforming programs with HashFuzz strongly improves the diversity of the test sets that fuzzers produce when run on them. The coverage scores of the test sets are maintained or improved upon for the transformed programs. Although the effect is small on average, 4.8%, it is significant as large gains in coverage are difficult to achieve.

More significant is the large improvement for some programs in detection of unique crashes. Improvements are between 28% to 97% after the transformation, providing strong evidence for the utility of combining input sampling diversity with syntax coverage based search when constructing test sets. An interesting question is whether HashFuzz could easily be extendable to other coverage based automated test set generation tools and methodologies.

Future work will centre on using the testability transformation to provide other forms of diversity, e.g. output diversity, following the work of Alshahwan and Harman [35] on output uniqueness, and improving grammar-based fuzzers such as BlendFuzz [60] where the grammar restructures provided inputs. Another possibility is to use the transformation to diversify a test set produced by a focused test generation process [14].

Acknowledgments

This work has been supported by the InfoTestSS EP/P005888/1 research project from EPSRC. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research. Many thanks to Robert Feldt for useful discussion on how to find a language to explain and present the novelty in this work.

References

[1] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[2] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 223–233.

[3] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 263–272.

[4] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[5] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.

[6] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.

[7] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform sat witness generation." in *TACAS*, 2015, pp. 304–319.

[8] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 240–251.

[9] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," in *2010 19th IEEE Asian Test Symposium*. IEEE, 2010, pp. 59–64.

[10] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, Jan. 2004.

[11] Google, "Google's fuzzer test suite," 2019. [Online]. Available: https://github.com/google/fuzzer-test-suite

[12] I. Diakonikolas, T. Gouleakis, J. Peebles, and E. Price, "Collision-based testers are optimal for uniformity and closeness," *arXiv preprint arXiv:1611.03579*, 2016.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[14] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[15] S. Hocevar, "zzuf—multi-purpose fuzzer," 2011.

[16] A. Helin, "Radamsa fuzzer," 2006.

[17] M. Zalewski, "American fuzzy lop," 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[18] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[19] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 475–485.

[20] K. Serebryany, "libfuzzer a library for coverage-guided fuzz testing," *LLVM project*, 2015.

[21] T. Guo, P. Zhang, X. Wang, and Q. Wei, "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *2013 Second International Conference on Informatics & Applications (ICIA)*. IEEE, 2013, pp. 212–215.

[22] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 724–735.

[23] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using xor constraints," in *Advances In Neural Information Processing Systems*, 2007, pp. 481–488.

[24] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[25] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 678–689.

[26] M. Böhme, "Stads: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 2, pp. 1–52, 2018.

[27] H. D. Menéndez, "Software testing or the bugs' nightmare," *Open Journal of Software Engineering*, vol. 1, no. 1, pp. 1–21, 2021.

[28] G. Gay, "The fitness function for the job: search-based generation of test suites that detect real faults," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 345–355.

[29] S. Poulding and R. Feldt, "Generating structured test data with specific properties using nested monte-carlo search," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 1279–1286.

[30] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb 2013.

[31] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi, "Approximate probabilistic inference via word-level counting." in *AAAI*, vol. 16, 2016, pp. 3218–3224.

[32] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, 2008, pp. 178–186. [Online]. Available: https://doi.org/10.1109/ICSTW.2008.36

[33] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu, "Measuring the diversity of a test set with distance entropy," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 19–27, March 2016.

[34] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1345–1348. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337414

[35] ——, "Coverage and fault detection of the output-uniqueness test selection criteria," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 181–192.

[36] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 595–606.

[37] Y. Marhuenda, D. Morales, and M. Pardo, "A comparison of uniformity tests," *Statistics*, vol. 39, no. 4, pp. 315–327, 2005.

[38] H. Menendez, M. Boreale, D. Gorla, and D. Clark, "Output sampling for output diversity in automatic unit test generation," *IEEE Transactions on Software Engineering*, 2020.

[39] H. D. Menéndez, G. Jahangirova, F. Sarro, P. Tonella, and D. Clark, "Diversifying focused testing for unit testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–24, 2021.

[40] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[41] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[42] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in 11^{th} *International Conference on Quality Software (QSIC)*, M. Núñez, R. M. Hierons, and M. G. Merayo, Eds. Madrid, Spain: IEEE Computer Society, July 2011, pp. 31–40.

[43] ——, "EvoSuite: automatic test suite generation for object-oriented software," in 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. ACM, September 5th - 9th 2011, pp. 416–419.

[44] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[45] "Agitar, [Online]. Available: https://www.agitar.com, [Accessed: 06-Mar-2018]."

[46] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 573–583. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568314

[47] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 2009, pp. 1–5.

[48] W. Masri and R. A. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 165–174.

[49] ——, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 8:1–8:28, 2014.

[50] X. Wang, S. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 45–55.

[51] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.
[52] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 736–747.
[53] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 712–723.
[54] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, "Sequence coverage directed greybox fuzzing," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 249–259.
[55] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "Diffuzz: differential fuzzing for side-channel analysis," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 176–187.
[56] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 533–544.
[57] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.
[58] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 146–157.
[59] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 59–67.
[60] D. Yang, Y. Zhang, and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 1070–1076.

Héctor Menéndez is a Lecturer at Middlesex University London, working on applications of information theory to software testing. Originally, he worked designing machine learning algorithms based on graph structures and search based optimization. He has applied these ideas to several different fields, where the most relevant are malware analysis, unmanned air vehicles and, currently, software testing.

David Clark is a Reader in Software Engineering at University College London. His research interests include Software testing, Application of Information Theory to software analysis, Program flow security, Slicing programs and software models, Malware detection and classification. David has published articles on a wide range of topics, including disrupting android malware triage by forcing misclassification, quantifying the diversity of sets of test cases, and test oracle assessment and improvement.