Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Enhanced Input Diversity by Hashing in Generator-based Fuzzing

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Johannes Brosz
geboren am:	13.08.1993
geboren in:	Magdeburg
Gutachter/innen:	Prof. Dr. Lars Grunske Dr. Yannic Noller

eingereicht am: .... verteidigt am: ....

Abstract

Automated testing is most important to detect bugs before an application is deployed. Fixing errors after deployment requires lots of resources and time. Fuzzing routines have become an essential part of automated testing.
The *JQF* Platform contains both fuzzers for different utilities and a framework to build a customised fuzzer. The Zest algorithm is a structured coverage-guided fuzzer delivered by JQF. As all coverage guided fuzzer, Zest benefits from creating and mutating divers inputs. To improve the input diversity on Zest, we augment the algorithm by a technique called *HashFuzz*. HashFuzz divides the input space into different partitions to initiate a more divers set of parent inputs. We implement a novel technique, *HashedZest*, building the idea of HashFuzz on top of Zest. This leads to new saved valid inputs Zest would not save by default and therefore and improved ability to detect bugs and coverage. Within the scope of the thesis, three versions of HashedZest are introduced, implemented and tested. We evaluate four fuzzers on five target programs and compare them among their detected bugs, branch coverage, valid to invalid input share and the execution speed. Although we cannot claim significant improvements in terms of bug detection and coverage, the thesis carries out a novel technique with an increased share of valid to total inputs and more promising findings for future work.

Contents

1 Introduction 4

2 Preliminary Work 6

2.1 Fuzzing 6
2.2 Mutation-based Fuzzing 7
2.3 Generator-based Fuzzing and Property-based Testing 8
2.4 Coverage-Guided Fuzzing 8
2.5 JQF 9
2.6 Fuzzing with Zest 11
2.7 Hashfuzz 13

3 Implementation 15

3.1 Workflow 15
3.1.1 Preprocessing 16
3.1.2 GetInput 19
3.1.3 ObserveGeneratedArgs 20
3.1.4 HandleResult 21
3.1.5 CheckSavingCriteria 21
3.1.6 SaveCurrentInput 22
3.2 Additional Concept: HashedZest+ 22
3.3 Additional Concept: Round Robin 22

4 Evaluation 25

4.1 Introduce Testing Setup 26
4.2 Mann-Whitney-U-Test 27
4.3 Effect Size Computation by Vargha and Delaney 27
4.4 Results 28
4.4.1 RQ1: Coverage 28
4.4.2 RQ2: Bugs 30
4.4.3 RQ3: Ratio valid to total inputs 33
4.4.4 RQ4: 37
4.5 Threats to Validity 39
4.5.1 Internal Validity 39
4.5.2 External Validity 39
4.5.3 Construct Validity 39

5 Conclusion 40

1 Introduction

In the course of digitisation, software inherits many tasks done manually before. Thus the amount and complexity of software increases as well as its maintenance costs [29]. Once software is deployed, fixing bugs and dealing with crashes require plentiful resources. To assure a certain quality standard, static code analysis tools like *Sonarlint* are available for almost every integrated development environment [28]. Sonarlint highlights bugs and code vulnerabilities and makes suggestions to reduce cognitive complexity in terms of clean code. Tools like Sonarlint reduce the number of bugs by avoidable coding mistakes.

Complementary to static code analysis, dynamic analysis reasons about behaviour of an application during execution [1]. Fuzzers feeding a program with inputs and either terminate when a crash is detected or continue recording the exposed errors until a time limit is reached. The routine can also be interrupted manually. As fuzzers report very few false-positive crashes and being versatile in terms of usability, they gain in importance. *Random fuzzers* apply many inputs to their target program by generating input files randomly [13]. They lack in producing valid test cases an application would actually work with and hence detecting bugs. *Feedback-Based fuzzers*, or coverage-guided fuzzer, collect coverage information during each execution and use them to generate valid inputs [30]. There are many more types of fuzzers, but the last category we would like to introduce are *Mutation-Based fuzzers.* Mutation-Based fuzzers require valid inputs to mutate bits or bit-sequences. Valid inputs can be provided either due to initial seed inputs or by generating them. To produce valid inputs on its own, the fuzzer needs information provided by feedback-based fuzzing techniques.

It is obvious that coverage-guided fuzzers would benefit from enhanced input diversity because it leads to unexplored branches. Furthermore new identified branches result in a higher overall code coverage and therefore more spotted bugs. However, some branches are guarded with conditions that are hard to meet. To increase the chance of hitting those branches we need to enhance the input diversity. Mutational fuzzers struggle to stick to a specific structure, those branches need. They rely on mutating binary strings without considering the basic input structure. Grammar-based fuzzers perform better as they generate structured test inputs. *Zest*, for instance, produces structured inputs as only certain bit sequences mutate while the overall input structure remains.

In the following *HashedZest*, a routine based on the coverage-guided fuzzer Zest , will be introduced. It is augmented by *HashFuzz*, a technique to enhance input diversity [24]. Zest combines coverage-guided fuzzing and mutation-based fuzzing. As most fuzzers, the performance of Zest highly relies on its input diversity [15]. The essential idea of HashFuzz is to subdivide the input space into equal sized partitions. This is done by performing an XOR-calculation on the binary input structure in order to compute a hash key to access each partition. Zest is a fuzzer based on the *JQF* framework. JQF provides a platform to use fuzzers for testing reasons or build a customised fuzzer in scope of research. JQF enables us to modify the Zest guidance in order to implement HashFuzz on top of Zest. Therefore we will use HashFuzz to extend Zest by adding an input saving criterion [34]. If the input is valid and the inputs' key is not already represented in the queue, it is saved. Therefore each partition of the input space is represented in the fuzzers queue.

During the course of this thesis, three approaches are developed in order to implement HashFuzz on top of Zest. It is assumed that HashedZest does exactly what HashFuzz is supposed to do. An extra saving criterion guarantees that any partition of the input space is represented in the queue. The subversion *HashedZest+* resets the criterion after a certain number of saved inputs. The idea is to save more valid inputs, Zest otherwise would not add to the queue by default. The last approach, *RoundRobin*, splits the entire queue into eight parts, one for each partition. The next input for fuzzing is picked sequentially from another part of the input space. This approach is intended to provide the highest diversification. On one hand, it is hard to predict how RoundRobin performs as it used to beat Zest, HashedZest and HashedZest+ in terms of diversification. On the other hand, subdividing the queue, keeping track of queue's indices and manage their responsibilities, demand many extra resources.

To answer the question, whether Zest benefits from the idea of HashFuzz, all implemented techniques are tested against Zest. Menendez and Clark (2022) demonstrated that Fuzzers such as *AFL* and *LibFuzz* profit from HashFuzz [24]. To evaluate this hypothesis three benchmarks: coverage, detected bugs and ratio of valid to total inputs, are used. In terms of coverage we distinguish between coverage only achieved by valid inputs and overall coverage. Zest, HashedZest, HashedZest+ and RoundRobin compete on the target programs ant, bcel, closure, maven and rhino. Each algorithm was applied twenty times for three hours on each target.

The following chapter introduces underlying fuzzing techniques of Zest as well as the JQF framework. HashFuzz and Zest are explained in-depth also. Chapter three outlines the fuzzers workflow within the augmented Zest guidance. Emphasis is put on modifications made in order to implement HashedZest and its' subversions. In chapter four research questions, the experimental setup, the results and threats to validity will be presented and discussed. Chapter five contains the conclusion and recommendations for future work.

2 Preliminary Work

2.1 Fuzzing

One of the earliest papers dealing with fuzzing was Miller et. al. in 1990 [26]. A fuzz generator was introduced as a random character generator. Today, fuzzing is a dynamic code analysis technique. Depending on the context, fuzzers come in various shapes with different strengths and weaknesses. Nevertheless, they all perform the same essential steps.

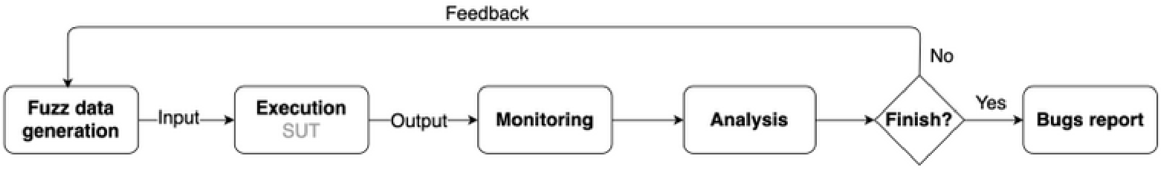


Figure 1: From [41]: The fundamental workflow of a fuzzer

Figure 1 shows every fuzzer begins with choosing an input to start with. This can be either generated by itself or is provided by the user as *seed*. The system under test (SUT), also known as *target*, is fed with the input. During the trial, information of the programs behaviour is recorded. The collected data is analysed and used for generating the next input. At the end, the detected bugs such as the inputs that triggered them are carried out by the fuzzer as text file into an output directory. As diving into all different kinds of fuzzers would go beyond the scope of this thesis, only a view types of fuzzers are explained consecutively. To get an overview of the current state of fuzzing, we refer to Manes et. al. [23]. To provide a brief overview on kinds of fuzzers and their various applications, a few fuzzing techniques are presented briefly.

LangFuzz is a mixture of a generative and mutative fuzzer [17]. The algorithm modifies existing input by randomly mutating it in two phases: first the fuzzer learns from input files and divides them into fragments. Second random fragments of existing files are replaced by the gathered fragments. This technique aims to generate input that is common enough to pass the interpreter but also triggers exceptional behaviour. However, it is a time-consuming and complex approach hash-based fuzzing could ease.

LibFuzzer is a coverage-guided fuzzer to test libraries [38]. The technique provides fuzzed inputs for the library under test and gathers coverage information due to *SanitizerCoverage* instrumentation [39]. Because LibFuzzer mutates inputs randomly, the routine relies on seed inputs to meet syntactical complex requirements.

The last technique introduced in this chapter, *NeuFuzz*, relies on a neural network to spot vulnerable paths rather than aim for a high overall code coverage [46]. Firstly, the model learns from a large amount of training data to distinguish clean and vulnerable paths. This enables the fuzzer to prioritize inputs by there ability to cover vulnerable paths. Subsequently, the fuzzer selects inputs that are more likely to be vulnerable.

2.2 Mutation-based Fuzzing

A random testing tool like QuickCheck generates inputs randomly [5]. Input produced by random fuzzers often misses the syntactical requirements a program demands from its inputs. To tackle this problem, mutation-based fuzzing relies on seed input, also known as *corpus* [16]. The corpus consists of initial inputs provided by the user. The fuzzer adds the seed to a set of *interesting inputs*. Every input within the set of *interesting inputs* can be regarded as template for mutations, as *parent* input. A mutation-based fuzzer now flips bits or bytes of the given initial input in order to mutate the parent and generate so called *children* inputs or *childs*. Further, a set to keep track of the coverage achieved so far, is initialized. If an input manages to increase the coverage, it joins the set of interesting inputs. The *seed scheduling* regulates which interesting input is used next. One way to perform seed scheduling is a graph centrality analysis to count the paths reached by a certain seed input [40]. The fuzzing procedure terminates either if a bug is detected or a certain time budget, determined by user, is consumed.

A well known mutation-based fuzzer is the *american fuzzing loop* (AFL) [47]. AFL is a brute-force approach using genetic algorithms for mutations. There are three mutations the algorithm can perform: overwrite, insert or delete bits. AFL can either flip single bits or mutate entire sequences of bytes. The new generated input is stashed in the queue and the fuzzer continues. The algorithm also collects information for each tested input file. The remaining issus is, that some paths might be uncovered as they require a very specific input. A genetic algorithm may takes too long to generate a fitting test case. Further AFL fails to generate highly structured input. Often inputs do not pass the syntactical stage of target programs. What is more, test cases generated by AFL lack in complexity.

Fairfuzz is an extension of AFL and aims to cover branches AFL can barely hit [22]. In the first step, FairFuzz runs a few AFL-produced inputs to identify rare reached branches. In the second step the algorithm produces test inputs targeting those rare branches. Over time, FairFuzz develops a mutation strategy by figuring out which parts of the input cannot be mutated in order to hit the target branch. This increases the probability to generate an input meeting the condition of the target branch and, therefore, discover new areas of the code.

2.3 Generator-based Fuzzing and Property-based Testing

Generator-based fuzzing and Property-based testing are used as synonyms by Padhye et al. [34] to describe the Zest fuzzing routine. Property-based testing was first mentioned by Claessen and Hughes [5]. Before property-based testing was widely known, most fuzzing techniques tried to identify bugs which force an application to crash. Property-based testing asserts that the target program is doing the right thing. Given a certain input to an application, a specific outcome is expected.

Imagine a method adding two integers. Assuming two integer values are provided, it is highly unlikely this method causes a crash except its result grows so big, positive or negative, it cannot be displayed or processed anymore. However, the method under test is always adding one to its result and computes wrong values. Most fuzzers would not report a bug because this behaviour will not cause a crash. By using property-based fuzzing, we can detect this bug by providing a *unit test*. The unit test determines when the test can be considered successful. An assertion, the output of the method has to equal the sum of both provided integers, would be violated and therefore, reported as bug.

In order to assure a valid input syntactically, *generator-based fuzzing* uses standard libraries to generate input of a certain type. In Java, the *DOM*-library produce syntactical valid XML for example. This allows users to put an emphasis on semantic testing.

2.4 Coverage-Guided Fuzzing

Coverage guided fuzzing provides information, so called *Trace Events*, regarding the coverage of each given input on a fuzzing target [33]. For instance, a *BranchEvent* records a conditional branch, a *CallEvent* marks a methods invocation and an *AllocEvent* reports a creation of a new object. During the fuzzing process, the tool creates a pool of testing files and induces new files by mutation.

As shown in Figure 2, coverage-guided fuzzers carry out two sets. The set S is initialized in Line 1 with a corpus. S contains test inputs that fit syntactical constraints. In Line 2, a set F is initialized to collect invalid inputs. Before the fuzzing loop starts, the *totalCoverage* is introduced in Line 3. Now, for each input in S (Line 5), a certain number of mutations will be generated and evaluated. Firstly, a candidate is produced in Line 7 by mutating the current input. Secondly, the trials result is captured in Line 8. If the result was a failure, the candidate is added to F (Line 10). Otherwise the candidate joins S in Line 12 and contributes its coverage to the overall coverage achieved (Line 13). In the following, a coverage-guided fuzzer is introduced as example those fuzzer can be implemented.

BonsaiFuzz is a grammar-based fuzzing technique to generate compact test inputs [45]. The key idea is to grow a corpus of concise test inputs bottom-up instead of reducing a generated test case. In order to so, BonsaiFuzz constructs small inputs based on coverage-guided grammar fuzzing rather than fuzz test inputs and reduce them afterwards.

```
Algorithm 1 Coverage-guided fuzzing.
Input: program $p$, set of initial inputs $I$
Output: a set of test inputs and failing inputs
$\mathcal{S} \leftarrow I$
$\mathcal{F} \leftarrow \emptyset$
$totalCoverage \leftarrow \emptyset$
repeat
    for $input$ in $\mathcal{S}$ do
        for $1 \leq i \leq$ NUMCANDIDATES($input$) do
            $candidate \leftarrow$ MUTATE($input, \mathcal{S}$)
            $coverage, result \leftarrow$ RUN($p, candidate$)
            if $result =$ FAILURE then
                $\mathcal{F} \leftarrow \mathcal{F} \cup candidate$
            else if $coverage \not\subseteq totalCoverage$ then
                $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
                $totalCoverage \leftarrow totalCoverage \cup coverage$
until given time budget expires
return $\mathcal{S}, \mathcal{F}$
```

Figure 2: From [34]: Steps of Coverage-guided fuzzing

2.5 JQF

JQF is a platform for coverage-guided fuzzing with property-based testing in Java [33]. The audience are researchers and practitioners. For researches, JQF provides the guidance interface as extension point. This allows researchers to implement standalone coverage-guided fuzzer. To do so, researchers can extend an existing guidance or implement a new one. Figure 3 show the JQF fuzzing loop. By adapting the guidance, the fuzzers behaviour can be modified.

For instance, the *guidance.getInput* method in Line 5 determines, how the next input is chosen. By implementing the *getInput* method from the JQF *guidance* interface, the researcher can introduce a new procedure to fill the fuzzers queue. Furthermore, the guidance set the configuration of a fuzzer like conditions, when a fuzzing routine will stop. By default, the algorithm runs as long as it is interrupted by the user. The fuzzer can stop when a test failure comes across or a certain timeout or number of trials is reached. However, the saving criteria for a interesting input, the benchmarks recorded during the run and the way, how a new input is selected for fuzzing and much more can be adjusted in the guidance. In addition, new functionalities can be implement here, such as the calculation of hash values based on the inputs' binary structure for instance.

Developers can use JQF with shipped guidances such as AFL or Zest. By default, JQF uses the Zest algorithm to generate test inputs. A brief description, how this works, will follow in chapter 2.4. To fuzz a method within a class, the practitioner can annotate a class with *Run with(JQF.class)*. The method the developer wants to test is annotated with *Fuzz*.

```
 1 TestMethod test    = ...; // @Fuzz test driver
 2 Guidance guidance = ...; // Fuzzing algorithm
 3 while (guidance.hasInput()) {
 4     // Generate args for test method
 5     Object[] args = JQF.gen(test, guidance.getInput());
 6     try {
 7         JUnit.run(test, args); // fires TraceEvent(s)
 8         guidance.handleResult(SUCCESS, null);
 9     } catch (AssumptionViolatedException e) {
10         guidance.handleResult(INVALID, e);
11     } catch (Throwable t) {
12         guidance.handleResult(FAILURE, e);
13     }
14 }
```

Figure 3: From [41]: The JQF fuzzing loop

Figure 3 illustrates the JQF fuzzing loop. In Line 1, the method to fuzz is set. JQF provides many different fuzzers a tester picks from. Then, as long as the guidance as inputs, the fuzzing loop runs. Because the guidance creates new structured inputs in case the queue is empty, the guidance always has input. The routine continues by generating the next input to fuzz in Line 5. To do so, three steps are executed:

1. Get an input from the guidance. If Zest is launched without seed input, the guidance will return a new generated test case for the first iteration. Otherwise, the next input from the queue is returned. At this point, the input is provided as byte stream.

2. The *SourceOfRandomness* takes a arbitrary amount of bytes and change their values. As the *SourceOfRandomness* is non-deterministic, the amount of bytes taken at once such as their processing varies. For example the bytes read from the input

stream could be converted to a character, an integer or something different. The decision is pseudo-random. In section 2.6 a more detailed explanation is provided.

3. Finally, the generator maps the sequence returned by the *SourceOfRandomness* to an array *args*

Now the fuzzer applies the input on the tested method in Line 7. The trial can be considered as success, invalid or failure. The different possible outcomes are explained in section 3.1.4. Before we continue to Zest itself, two fuzzers based on JQF are introduced to give an idea of the various applications JQF can be used for.

PerfFuzz puts an emphasis on revealing performance bottlenecks on target programs [21]. It extends AFL with performance feedback by mapping a value v with a key k on the coverage map. Each key refers an edge an the code and a value v counting loop executions or the amount of allocated memory for instance. If an input reveals new coverage or provides a higher v for a certain k, the input is saved. PerfFuzz extends JQF by adjusting the parameters to assess the result of a fuzzing trail. In particular, the feedback of the by PerfFuzz implemented performance map is considered to evaluate fuzzing trial outcomes.

BigFuzz for data-intensive scalable computing (DISC) applications [48]. Due to the high latency of DISC-programs, BigFuzz abstracts the framework using specifications. To do so, two steps are performed:

1. Build an acyclic graph of the data flow and search for method invocations corresponding to data flow operators

2. Rewrite the application as simplified yet semantically-equivalent program

It will be easier to generate suiting tests for the resulting application. BigFuzz extends the JQF functionality by adding trace events to keep track on data flow events.

2.6 Fuzzing with Zest

Zest is based on the feedback-directed testing framework JQF [34]. The fuzzer augments the approach of coverage-guided fuzzing by two essential principles:

1. Zest distinguish between the *totalCoverage* and *validCoverage*. *TotalCoverage* keeps track on the overall gathered coverage. *ValidCoverage* is an extra coverage counter to keep track on all branches covered by valid inputs. The distinction allows to put an emphasis on valid inputs in order to dig deeper into an applications semantic stage by fuzzing [35].

2. To create fresh input, Zest merges the strengths of generator-based fuzzing and mutation-based fuzzing. The algorithm implements a so called *parametric-generator*.

As displayed in section 2.3, generator-based fuzzing uses libraries to provide syntactical valid inputs. A generator produces a valid input file. Then, Zest identifies *parameter sequences* and maps them wherever the bytes are in use. As shown in Figure 4, the sequence *foo* is mapped for two locations. Although *foo* appears twice in the XML-file, it is represented once as parameter sequence for further mutations. Accordingly, a single bit flip on the sequence impacts both locations of the file. Zest collects those *parameter sequences* bytes and gather them as *parameter stream* to the mutation engine. The mutations only happen on the parameter stream with no harm the syntax. Furthermore, Zest modifies XML-tags. To stick to the example, Zest can also introduce new tags and therefore, increase the complexity of generated inputs. This allows Zest to test an application under circumstances closer to the applications natural environment than most other fuzzing techniques.

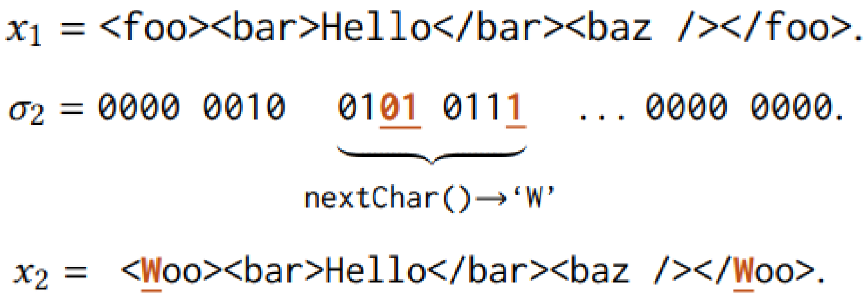

x1 = <foo><bar>Hello</bar><baz /></foo>.
σ2 = 0000 0010 0101 0111 ... 0000 0000.
nextChar()⟶'W'
x2 = <Woo><bar>Hello</bar><baz /></Woo>.

Figure 4: From [34]: How the parametric-generator mutates parameter sequences

To perform random mutations on the parameter sequences, Zest embed a technique invented for Hashkell applications. *QuickCheck* is a library for random testing [5]. The tool relies on target properties provided by the programmer in order to generate a huge amount of random test cases. QuickChecks' input generator is of interest for Zest to perform random mutations on the parameter sequences. Because QuickCheck is invented for Haskell programs, Zest relies on the *junit-quickcheck* library as adaptation for Java [18]. This library allows Zest to perform byte-level mutations as sketched in Figure 4. The mutation procedure consists of three main stages outlined in [34]:

1. Set a random number m as quantity of mutations applied on the parameter sequence
2. Set two more random numbers. The length of bytes to mutate l and an offset k to determine where to begin the mutations on the selected byte sequence
3. Replace the bytes from position k to k+l with l randomly generated bytes

The parameters l and k follow a geometric distribution. Padhye et al. set the mean to four because a four-byte integer is a frequently requested value.

2.7 Hashfuzz

HashFuzz tries to enhance a fuzzers input diversity by subdividing the input space due to universal hashing. In order to explain HashFuzz, we dive deeper into the concept of universal hashing. Afterwards, the opportunity for fuzzing by using universal hashing is demonstrated.

XORSample [11] and UniGen [4] were invented to enhance the diversity of solutions a deterministic solver would provide. Usually, the solver would return the first solution found. But there are other possible solutions the tester might be interested in. There, universal hashing is used to increase the diversity of solutions. Menendez and Clark sketch the idea by example: Imagine a boolean function $f_x(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$. The tester requires a solution satisfying f_x. The solver works deterministic and, therefore, flips x_1 first. The result already satisfies f_x and the solver returns $x_1 = 1, x_2 = 0, x_3 = 0$. If the determistic solver is asked to find a solution for f_x over and over again, the first solution presented would occur disproportionately often. That is why UniGen and XorSample force the solver to find different solutions for each partition of the input space. In order to do so, the partitions of the input space must be accessed by an r-wise independent hash function that satisfies:

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

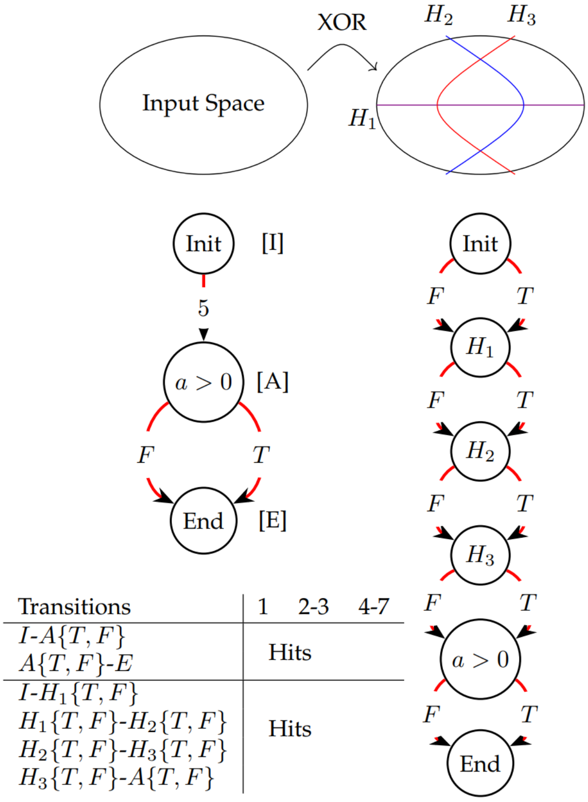
Figure 5: From [24]: the uniform distribution for r-wise independent hash functions

The is represented as $k_i \in K$ and $v_j \in V$ as value. With regard to the example, f_x is a boolean function. Thus, the values of $x_i \in \{0,1\}$ and $|V| = 2$. Gomes et al. [11] defined a H_{XOR} family. H_{XOR} consists of hash functions execute XOR operations on their values. Any function of H_{XOR} is three-wise independent also proved by Gomes et al. [11]. Accordingly, the input space is divided into $|V|^3 = 8$ partitions. Back to the example: the hash function h_i represents one variable of f_x. h_1 can either be zero or one such as x_1. If x_1 is set to zero, the solver must find a different solution. Consequently, the solver will also try to find a solution by flipping x_2 and x_3. To define a hash function, two parameter are require: An independent element b and a vector a. The vector provides coefficients for each variable. Both parameter determine a hash function as follows:

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

Figure 6: From [24]: b and a determine the hash value on each variable

As see in section 2.4, coverage guided fuzzer would benefit from enhanced input diversity because input diversity leads to unexplored branches. Furthermore, new identified branches result in a higher overall code coverage and therefore more spotted bugs. In addition, some branches are guarded with conditions that are hard to meet. To increase the chance of hitting those branches we need to increase the input diversity. In section 2.2 we explained that mutational fuzzers struggle to stick to the syntax while comply the semantic constraints those branches need. They rely on mutating binary strings without considering specific structures. Grammar based fuzzers perform better as they generate structured test inputs. Zest, for instance, produces structured input and mutates only certain parameter sequences while the overall input structure remains. Applying the idea of HashFuzz to a structured fuzzer like Zest might symbiotic effects.



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		

Figure 7: From [24]: Example of an modified program by adding artificial branches by *HashFuzz*. Furthermore we can see the divided input space by those artificial branches.

The input space would be divided as displayed in Figure 7. A hash function h consists of the three concatenated hash values of h_i. For example, we want to address the partition bordering the purple, blue and orange line in the upper half of the input space. We determine $h_1 = 1$ as the upper half of the input space and $h_1 = 0$ as the lower half. Further everything on the right of the blue respectively orange line is set to $h_2 = 1$ and $h_3 = 1$. To access desired partition, we need the hash function $h = [h_1 = 1, h_2 = 0, h_3 = 1]$ which equals $h = 101$.

3 Implementation

We implement a new concept called *HashedZest* and compare it along Zest. HashedZest describes the approach to implement *HashFuzz* on top of Zest. Regarding to section 2.7, *HashFuzz* suggest to divide the input space into eight partitions. To access the partitions, a hash function of the H_{XOR} family is used. To receive a hash function, we concatenate three hash functions h_i as displayed by example on Figure 7. Each function h_i is calculated as shown in Figure 6. Figure 6 demonstrates how the random parameter a and b are applied on a stream of values x_j. Zest provides the mutating parameter as byte sequence as mentioned in 2.6. In order to implement the idea of HashFuzz, we transform the bytes into a bit sequence and calculate the hash keys on their binary structure.

This chapter aims to introduce the workflow of HashedZest, how the hash key calculation is implemented and further adjustments for the purpose of implementing HashedZest. What is more, two additional approaches of HashedZest are implemented and tested too. Their difference to HashedZest is demonstrated as well.

3.1 Workflow

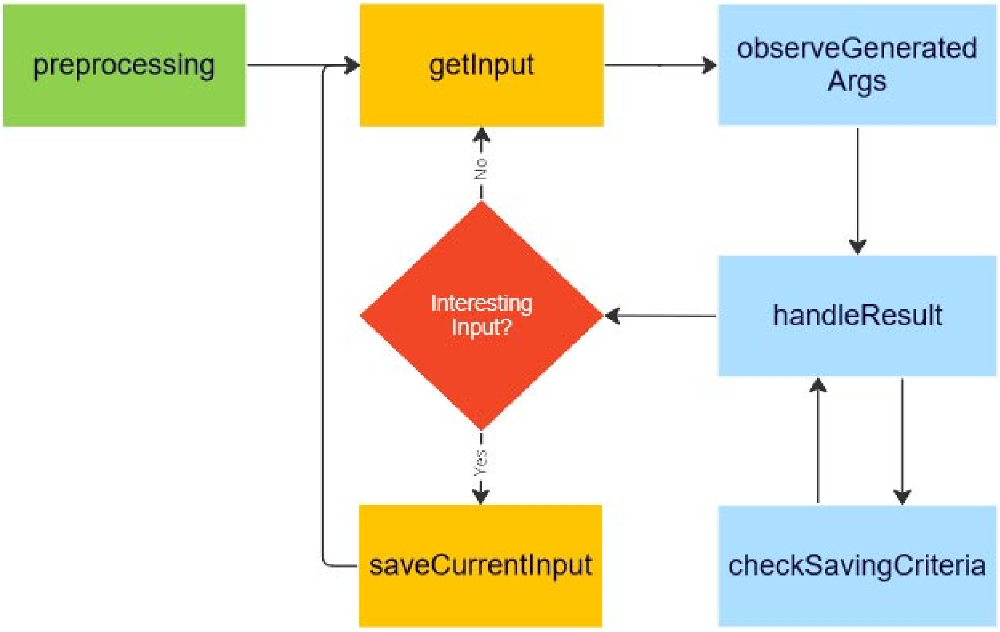


Figure 8: HashedZest Guidance Workflow. Green: Extra method introduced by HashedZest. Yellow: Methods already included at Zest. Red: Decisions already part of Zest. Blue: Method already part of Zest but overridden at the HashedZest guidance.

The JQF framework delivers a guidance class for every fuzzer. In the following, focus lies on Zest guidance, since implementation of HashedZest, HashedZest+ and RoundRobin are based on Zest. There are many more classes and methods in use during a fuzzing run such as classes to perform the parametric mutations explained in chapter 2.4. Those classes however, were not modified in this work, which is why they will not be explained on greater detail. The overall workflow, displayed in Figure 8, results from the *evaluate* method introduced in section 2.5. The following chapters demonstrates the steps performed for each trial such as the tasks each method is responsible for. If a method is adapted in order to implement HashedZest, those changes are highlighted as well.

3.1.1 Preprocessing

The method *preprocessing* is invoked within the constructor of the HashedZest guidance. In order to implement HashedZest, a few extra calculations are performed compared to Zest. As mentioned in chapter 2.7, HashFuzz divides the input space into eight partitions. Each partition can be accessed by a hash key. The hash key lies in range of zero to seven. The hash key depends on the binary structure of an input. Later on, the input is represented as byte sequence. Each byte is an integer in range of zero to 255. According to chapter 2.7, the hash key is calculated on the basis of the binary representation of those bytes. To reduce the time exposure of those extra steps, the *preprocessing* eases the hash value calculation. After *preprocessing*, HashedZest can map every possible byte value to zero or one. To explain the procedure in detail, Algorithm 1 is explained by an example byte value of 213.

In order to calculate the hash key, three different hash functions are applied. Algorithm 1 sketches the preprocessing of one hash function. The other two are preprocessed the same way. A hash function h_i with $i \in [1, 3]$ consists of a random bit b and a binary sequence of eight digits a as integer array. Both parameters were determined randomly in Line 1 and 2. Zest provides the input as parameter sequence in form of an integer array with values ranging from zero to 255, represented as *possibleValues* array in Algorithm 1. Therefore, the length of a has to be eight. Otherwise a the multiplication in Line 6 will not work as the input binary representation of each possible byte must equal the size of a.

Initially, to generate eight bits randomly to determine a was attempted. This led to non random sequences such as "0000 0000" and "1111 1111" that are two of three sequences we need for h_i. That is why a different approach is chosen: Firstly, an integer from zero to 255 is generated by chance and is turned into a bit sequence. Secondly, if the binary representation had less than eight digits, leading zeros were added up in the end. Imagine a random integer generator returned 123. Its binary representation is "111 1011". In order to assert a consists of eight digits, the leading zero was added and continued with "0111 1011".

Algorithm 1 preprocessing byte value calculation

```
    **Input:**; *possibleValues*
    **Output:** *digit*
 1: $a \leftarrow randomBitSequence$
 2: $b \leftarrow randomBit$
 3: for each $byteValue \in possibleValues$ do
 4:     $binaryRepresentation \leftarrow byteValue$
 5:     for each $bit \in binaryRepresentation$ do
 6:         $binaryRepresentation[i] \leftarrow binaryRepresentation[i] \cdot a[i]$
 7:     end for
 8:     $k = 0$
 9:     while $binaryRepresentation.size > 1$ do
10:         $resultXOR \leftarrow binaryRepresentation[k] \oplus binaryRepresentation[k+1]$
11:         $binaryRepresentation.set(k, resultXor)$
12:         $binaryRepresentation.remove(k+1)$
13:         if $k + 2 \geq binaryRepresentation.size()$ then
14:             $k \leftarrow 0$
15:         else
16:             $k \leftarrow k + 1$
17:         end if
18:     end while
19:     $digit \leftarrow binaryRepresentation[0]$
20: end for
```

For the random bit b *Math.round* is applied on *Math.random()*. The result is of type double, so it is casted to an integer for further processing. Because b is used in *observeGeneratedArgs* later on, the three random bits b are stored in an array *randomBit*: b_1 for h_1 in $randomBit[0]$, b_2 for h_2 in $randomBit[1]$ and b_3 for h_3 in $randomBit[2]$. Now that the random parameters a and b are settled, each integer in *possibleValues* (Line 3) is mapped to zero or one. In Line 4 the current byte is converted to the *binaryRepresentation* array.

Sticking to our example, 213 is converted to "1101 0101". This happens in two steps. 213 is converted into a binary string by the *Integer.toString(213 ,2)* function. The second property of *Integer.toString* determines the radix. Afterwards, each character is mapped back to its numeric value and added to an integer array. Now, the hash function is applied on the byte value due the multiplication with the random bit sequence a bit by bit in Line 6:

- $binaryRepresentation[0] = 1 \cdot 0 = 0$
- $binaryRepresentation[1] = 1 \cdot 1 = 1$
- $binaryRepresentation[2] = 0 \cdot 1 = 0$
- $binaryRepresentation[3] = 1 \cdot 1 = 1$
- $binaryRepresentation[4] = 0 \cdot 1 = 0$
- $binaryRepresentation[5] = 1 \cdot 0 = 0$
- $binaryRepresentation[6] = 0 \cdot 1 = 0$
- $binaryRepresentation[7] = 1 \cdot 1 = 1$

The output of Line 6, 0101 0001, is assigned to *binaryRepresentation*. The next step is to reduce *binaryRepresentation* on one bit. Therefore, the XOR operation is applied on adjacent bits in Line 10. The XOR operation returns one, if both adjacent bits have not the same value or zero, if adjacent bits are equal. The variable *k*, introduced in Line 8, keeps track on the index of the *binaryRepresentation* array. The result is stored at position *k* (Line 11) and the bit at the following position is removed (line 12). As the size of *binaryRepresentation* shrinks, Line 11 to 14 make sure *k* cannot grow out of index. Line 7 checks whether the *binaryRepresentation* is reduced on one bit successfully. For *binaryRepresentation*, it works like this:

1. $0 \oplus 1 = 1$
2. $0 \oplus 1 = 1$
3. $0 \oplus 0 = 0$
4. $0 \oplus 1 = 1$

At this point, *k* equals three while *binaryRepresentation* is reduced on size five. Thus, *k* is resetted to zero. The final result is $(1 \oplus 1) \oplus (0 \oplus 1) = 1$. The result is added to the global array *digit* at the position of the byte processed. Regarding our example, $firstDigit[213] = 1$. The global arrays represent the hash functions later on: h_1 is represented by *firstDigit*, h_2 by *secondDigit* and h_3 by *thirdDigit*. Every position in an array represents a possible byte value of the parameter stream representing the input later on. The hash key calculation is located in the*observeGeneratedArgs* method. This method is introduced in section 3.1.3 were we explain how the hash key is assembled.

3.1.2 GetInput

The first step of each fuzzing trial starts with is the selection of an input to continue with. The *getInput* method determines either getting an input from the queue or creating a new one. If the queue is empty, a new input is generated considering the feedback information Zest gathered so far. In case the queue is not empty, *getInput* calls a method *getTargetChildrenForParent*. This method determines how many children a given parent input should gather. To do so, *getTargetChildrenForParent* considers the baseline of mutations per parent set by the user. It is common to set 100 to 300 mutations as baseline [32]. A 100 children were set as standard. Nonetheless there are reasons to increase parent input for more mutations. If the parent input for instance achieves a high coverage, the number of children is adjusted:

$$newNumberOfChildren = \frac{numberOfChildrenBaseline \cdot parentCoverage}{maxCoverage} \tag{1}$$

The variable *maxCoverage* represents the coverage accomplished so far. To avoid division by 0, this option is initiated only, if any coverage is achieved so far. If an input is favored, this is another reason to increase the number of maximum mutations per parent input. An input is preferred, when it is responsible for covering one branch at least. If an input is favored, the integer *favoredMultiplier* scales up the number of children. The *favoredMultiplier* is defined in the guidance, and by default set to 20:

$$newNumberOfChildren = numberOfChildren \cdot favoredMultiplier \tag{2}$$

Now, the amount of children that stem from current parent input is compared to the number of children the input already has. In case the number of maximum children for this input is not reached yet, another child is generated to continue. Otherwise the next input is selected from the queue and the number of current children is resetted to 0. The parent input is now used to fuzz a new input. Therefore, the parent input is used as template and mutates in the parametric generator explained in section 2.6. The method returns an input stream.

To transform the current input into an input stream, the method *createParameterStream* is invoked. Firstly, *createParameterStream* casts the input to a linear input. Secondly, the method attempts to read a byte value as integer from the linear input. If the attempt fails, a random value is generated and returned. The returned integers are collected as *randomFile* in the *FuzzStatement* method of JQF mentioned in section 2.5.

3.1.3 ObserveGeneratedArgs

Within the JQF framework, this method is intended to observe the parameter sequence representing the input before it is applied on the tested program. In Zest guidance, this method is not implemented from the JQF guidance interface. In terms of HashedZest, the hash value for the current input is calculated here. As the standard HashedZest is only interested in covering each partition of the input space at least once. To prevent unnecessary calculation, the hash key generation is guarded by a boolean *allKeysCovered*. This boolean is "false" by default and becomes true if any key was saved at least once. To save resources, the hash key calculation is only performed as long as *allKeysCovered* is false.
According to chapter 2.7, the hash key is calculated from three independent hash functions h_i. Every h_i returns a value either zero or one. The results of each function are concatenated as string and converted to an integer in range of zero to seven. The hash functions are represented by the preprocessed arrays *firstDigit*, *secondDigit*, and *thirdDigit*. They map each byte of the input sequence to zero or one. The procedure performs the following steps:

1. The *observeGeneratedArgs* is invoked with an input as argument. At this point, the input consists of all characters, even those to obtain the demanded input structure by the target.

2. To continue, the input is casted from *parameterByteStream* to *LinearInput*. A *LinearInput* comprises only the parameter sequences mutated by the input generator. The casting turns the input into a integer array containing the parameter as byte value: $inputArray = [23, 210, 1, 199]$

3. Let *firstDigit* be: $"firstDigit" = [0, 1, .., 1, .., 0, ..1, ..]$. This array represents the hash function h_1 and was generated during the *preprocessing* explained in section 3.1.1.

4. The input array is mapped on *firstDigit*: $firstDigit[1] = 1$, $firstDigit[23] = 1$, $firstDigit[199] = 0$, $firstDigit[210] = 1$

5. This leads to a mapped input array: $inputArray = [1, 1, 0, 1]$

6. As h_i returns a single number zero or one, the *inputArray* is reduced by the XOR operation: $firstKeyDigit = (1 \oplus 1) \oplus (0 \oplus 1) = 0 \oplus 1 = 1$

7. We map *secondDigit*, and *thirdDigit* the same way and receive a binary string. Image the mapping of *secondDigit* equals zero and *thirdDigit* equals one

8. The three numbers are concatenated as string of "101"

9. The binary string is converted to an integer which represents the current hash key: *Integer.parseInt("101", 2) = 5*

10. Concluding, the global string *currentKey* is set to five.

3.1.4 HandleResult

After a fuzzing trial is finished, the collected information are reviewed. A trial can have four different outcomes:

1. SUCCESS: The generated input was valid and Therefore, the trial completed successfully without unexpected errors. As mentioned in section 2.3, the tester determines what is the expected outcome under determined circumstances.
2. FAILURE: In case the trail threw an unexpected error, the trial is assessed as fail.
3. INVALID: The tester provides handwritten assertions as sketched in section 2.3. If those assertions are violated, the trial result is INVALID.
4. TIMEOUT: In case the trial exceeds a certain time limit. Within the scope the thesis, no such time out is determined.

In case the result was SUCCESS or INVALID, the input might be saved. The circumstances to save an input are displayed in section 3.1.5. If the *checkSavingCriteria* method returns at least one reason to save the input, it is saved in the *saveCurrentInput* method.

The method *handleResult* has a second important task to perform: different inputs can trigger the same bug. Counting a detected bug multiple times is prevented by the error deduplication. We realise this functionality by implementing the approach of Boehme et. al. [3]: To identify duplicated bugs, the root cause and parts of the stack trace must be saved. As the first stack trace entries are usually the same, the last three entries are more appropriate for deduplication. On each of the last three entries, the hash code is stored in an integer array *trace*. The hash code of the root cause is stored in *trace* as well. Finally, the hash code is calculated on *trace*. The set *errors* contains the hash code of each detected bug until now. If *errors* does not contain the hash code of *trace*, the code is added and the detected bug is considered for further processing. The error deduplication was also modified for Zest in order to conduct our research.

3.1.5 CheckSavingCriteria

This method returns a list of strings *reasonsToSave* to the *handleResult* method. Reasons to save input are added to the list. If the list is empty, the current input will not be saved. When a saving criteria is met, a short string is added. By default, *checkSavingCriteria* has three reasons to save an input:

1. The runcoverage contributes new explored branches to the overall coverage, "+cov" is added to the string.
2. The valid coverage was increased, "+valid" is added item The coverage bits were updated, "+count" is added

Another saving criterion is introduced with regard to implementing HashedZest. If the hash key of the current input is not represented in the queue yet, the saving criterion "+ partition" is added to the string. Afterwards the current hash key is stashed in a set *coveredKeys* of all keys saved up to this point. When the size of *coveredKeys* is eight, every key is saved at least once. Therefore, the *allKeysCovered* boolean introduced in 3.1.3 is set true and no further hash value calculation is performed.

3.1.6 SaveCurrentInput

The method performs multiple tasks. *SaveCurrentInput* is responsible to add interesting inputs to the queue. If the *checkSavingCriteria* method returned a not empty string to *handleResult*, the current input is added to the queue. During the trial of the current input, information such as the coverage accomplished and the inputs' responsibilities are tracked. When an input is saved, this information is added to the inputs' properties. At the end, the current input is saved in the output directory as text file.

3.2 Additional Concept: HashedZest+

The proposed technique Hashed Zest will add a maximum of eight inputs to the queue. For target programs such as ant, the queue contains around 150 inputs after one hour. On one hand, it seems reasonable that eight extra saved inputs might have a significant impact on the fuzzers' behaviour. On the other hand, closure produces many more inputs. It is hard to imagine eight extra inputs can make a difference. This technique suggests to reset the criterion after a certain number of inputs is saved. Determining the correct number of saved inputs is a matter of balance. With a more frequent reset, more redundant input would be added, contributing nothing new to coverage. Mutating them is less promising than the regular saved input. A less frequent reset of the saving criterion will diminish the effect on target programs like ant. Thus resetting the criterion every 100 saved inputs in the *saveCurrentInput* method was agreed upon and is further explained in 3.1.6.

3.3 Additional Concept: Round Robin

RoundRobin aims to maximize the diversification. HashedZest and HashedZest+ aim that each partition of the input space is represented by at least one input in the queue. The major difference between RoundRobin and its competitors is, that the inputs are saved in a hash map instead of the queue. At the linear queue, there is no guarantee that an input of each partition of the input space is fuzzed frequently. The map consists of eight partitions, one for each hash key as illustrated in Figure 9. In method *getInput*, a new parent input is chosen from the map to add it to the queue. We feed the linear queue with inputs from the hash map because Zest tracks properties such as input responsibilities. To manage those properties on a map instead of a linear queue was considered as too expensive. The new parent is to be picked from a different queue. As explained in 3.1.2, the method *getInput* is responsible to select or create the next parent input. For Zest, HashedZest and HashedZest+, the method works the same way. As sketched in section 3.1.2, a new parent input is chosen when it achieved its' maximum number of children. In that case, the next input of the queue is identified by taking the next saved input as long as the end of the queue is not reached yet. Otherwise, the queue starts again with the first saved input. In order to implement RoundRobin, the *getInput* method is modified.

Algorithm 2 displays the changes made on *getInput* in case a new parent has to be picked. To pick an input from the map, the *parentKey* integer and index have to be defined. The *parentKey* integer tracks the partition the current parent input belongs to. It is increased by one in order to get the next parent input from a different partition (Line 9). As the input space is divided into eight areas, the *parentKey* has to remain within a range of zero to seven. This is asserted by Line 1 and 2. At the beginning of a run, possibly not every partition of the map contains at least one input.

Algorithm 2 modified getInput

```
 1: if $parentKey + 1 == 8$ then
 2:     $parentKey \leftarrow 0$
 3: else
 4:     $parentKey \leftarrow parentKey + 1$
 5: end if
 6: while $inputMap(parentKey).isEmpty$ do
 7:     if $parentKey + 1 == 8$ then
 8:         $parentKey \leftarrow 0$
 9:     else
10:         $parentKey \leftarrow parentKey + 1$
11:     end if
12: end while
13: $index \leftarrow indexOfPartitions(parentKey) + 1$
14: if $index \geq inputMap.size$ then
15:     $index \leftarrow 0$
16: end if
17: $indexOfPartitions \leftarrow index$
18: $nextParent \leftarrow inputMap(parentKey, index)$
19: $queue \leftarrow nextParent$
```

Therefore, Line 6 - 10 guarantee that the *parentInput* is selected from a non-empty partition of the map. Because the path illustrated in Algorithm 19 is only entered when the hash map contains at least one input, the loop will always terminate. Now, the *parentKey* is settled, the index needs to be identified.

The common Zest algorithm increments the queues' index or reset the index to zero when the end of queue is reached. A structure is required, that keeps track of each partition with their indexes. The *indexOfParts* array provides this functionality. In Line 13, the indexes are drawn from the array. As the index was incremented, the index could lie out of the range for the current part. Line 14 - 16 reset the index to zero, if the end of the current partition is reached. In Line 17, the index is updated in *indexOfParts*. Now that the *parentKey* integer and the index are defined, the *parentInput* is selected from the map (line 18). In the end, the *parentInput* is added to the queue.

To implement RoundRobin, the *saveCurrentInput* method, introduced in section 3.1.6, needs minor modifications too. Zest, HashedZest, HashedZest+ and RoundRobin start with an empty queue. For any algorithm but RoundRobin, the first saved input is added to the queue. But for RoundRobin, interesting inputs are saved in a map and added to the queue later on. Thus, the first partition an input is stored has to be identified. As mentioned in section 3.1.6, the *numSavedInputs* counter tracks the amount of inputs saved so far. If *saveCurrentInput* is invoked and *numSavedInputs* equals zero, the first input will be saved. The hash key generated for the current input is set as first parent key. The input is added straight to the queue also as first parent input. Further the *indexOfPartitions* is increased at the position of the current key. The last extra step performed in *saveCurrentInput* method is to save the current input in the map.

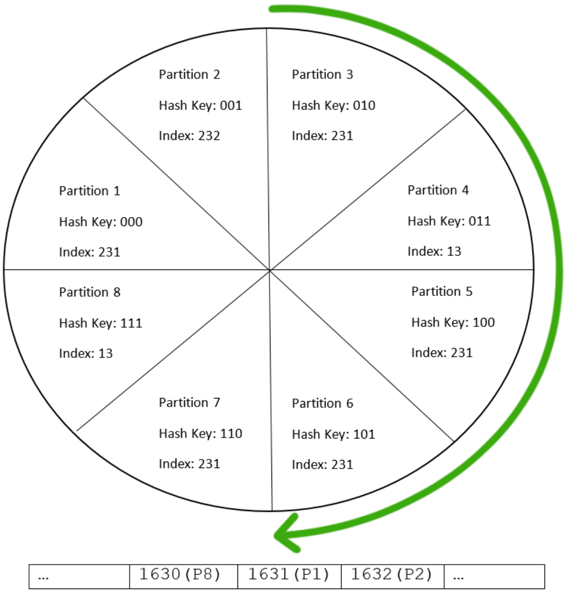


Figure 9: The hash map to save interesting inputs as circle and a snippet of the linear queue. The hash keys are displayed as binary string to connect with the explanations regarding the hash key calculation in 3.1.3. The index represents the value stored in *indexOfPartitions* array. For partition 4 the index set to zero because the end of the partition was reached. The queue contains the parent indices and the partition the parent belongs to.

4 Evaluation

The thesis intends to answer the following research questions:

RQ1: Will an implementation of HashedZest lead to an significant increased coverage compared to Zest? (chapter 4.5.1)

RQ2: Will an implementation of HashedZest lead to significant more identified bugs as compared to Zest? (chapter 4.5.2)

RQ3: Will an implementation of HashedZest be able to generate a significant higher share of valid inputs compared to Zest? (chapter 4.5.3)

RQ4: Do the extra steps performed by HashedZest' implementations lead to a significant reduced execution speed? (chapter 4.5.4)

RQ1 and RQ2 compare how Zest, HashedZest, HashedZest+ and RoundRobin perform in terms of coverage and bug detection. In addition, RQ3 and RQ4 aim to answer how outcomes in RQ1 and RQ2 are achieved, in more detail.

In order to increase expressiveness of the results, the principles of Klees et al. [19] are applied to the testing setup as well as the evaluation of results. Klees et al. suggest to conduct multiple runs for each tested approach for a minimum time limit of three hours for instance. Furtermore, significant relevance needs to be investigated by statistic test such as the *t-test* or *Mann-Whitney-U-Test*[9]. How the principles of Klees et al. are considered during the experiment is lined out in the following section.
What is more, we try compare our results to Menendez and Clarks' observations [24]. In order to do so, three main differences between their experimental setup and our approach should be considered:

1. Menendez and Clark tested their approach on targets such as *C-ares*[42], *LibPNG* [37],*LibXml2*[44], *Pcre2*[14] and *Re2*[12]. These targets are based on C respectively C++. We use five common java applications to test our techniques. Most of the targets use different input files: while bcel, closure and rhino work with JavaScript-files, Menendez and Clarks targets work with png and regular expressions. Because LibXml2 and maven/ant deal with XML input files, it seems most reasonable to compare our results on this target programs.

2. The authors apply HashFuzz on unstructured fuzzers such as AFL [47], LibFuzz and Fairfuzz [22]. Zest is a structured fuzzer. It will be interesting to see if fuzzers like Zest can also benefit from HashFuzz.

3. Menendez and Clark do not provide any information on their experimental setup in terms of timeouts or runs performed. Regarding Klees et al. [19], it is possible

Menendez and Clark only performed a single run for each technique. This would make it even harder to compare our results.

4.1 Introduce Testing Setup

For the sake of external validity, five target programs Apache Ant, Apache Byte Code Engineering Library (bcel), Google Closure Compiler, Apache Maven and Mozilla Rhino are tested. The JQF-Framework contains all of them as they the most common testing targets in Java. Zest, HashedZest, HashedZest+ and RoundRobin are compared along them. All algorithms run twenty times for three hours with each target. The algorithms will only terminate when the time budget is consumed. To avoid running each execution sequential for 300 hours, the terminal multiplexer *tmux* is used to make 20 runs at the same time. At each window of tmux, a script is executed to run one algorithm for three hours on each target program sequential. As common notebooks can barely handle a single run, this tests are conducted at the computing centre of the Humboldt University of Berlin. To guarantee comparability through all runs, we booked a server with 256 GB RAM exclusive. The server is equipped with an AMD EPYC 8813P 64-Core Processor, 1.5 GHz CPU.

Zest is a well performing fuzzer, even without seeds. For each execution no seeds are provided. JQF can be used with two different instrumentations: *Fast* and *Janala* [31]. Janala is JQFs' default coverage instrumentation. It estimates the map size by saving branch IDs' in an array. This could lead to collisions and harms the fuzzers performance. The Fast instrumentation is not keeping track on the map and therefor less collision-prone. All algorithms use the Fast instrumentation instead of Janala. It is still possible to compare the coverage by counting the branches reached.
Every run saves its collected data at a *plotdata.txt*-file. For all executions, those files are downloaded for further processing. The tables and figures below are generated by python scripts. The scripts also calculate the mean, standard deviation and median displayed at all tables below. The last line of the *plotdata.txt*-file contains the values of each benchmark when timeout is reached.

The values of interest are coverage, bugs detected, valid inputs to total inputs ratio and the total inputs. All benchmarks ask for the final result at the end of each trial. Therefore, an array *values* captures the value at the last row for the requested benchmark over all twenty runs. Only exception is the execution speed, capture in order to answer RQ4, where the median of each run is stored in *values.* Each algorithm has its own array to collect the result. The arrays are reset for every new benchmark or target program. Then, *numpy* works out the mean, standard deviation and median on the *values* array.

To identify significant outcomes, the Mann-Whitney-U-Test is applied on the *value* arrays of each algorithm [9]. We use a significance level of $\alpha = 0.05$. Higher significance levels are highlighted in the following tables.

4.2 Mann-Whitney-U-Test

The Mann-Whitney-U-Test, also known as Wilcoxon signed-rank test, outlines the significance of two small ($n < 30$) samples [8]. The test is used for three reasons: Firstly, we have twenty runs for each algorithm on each target as raw data. Consequentially, with a sample size of $n = 20$, the compared samples are small. What is more, we cannot assume a standardised normal distribution for the performance of fuzzers [19].
Python provides the *scipy.stats* library. The library contains the *mannwhitneyu* property, which computes significance of results [10].

4.3 Effect Size Computation by Vargha and Delaney

While the Mann-Whitney-U-Test sets **whether** two techniques differ significantly in performance, the effect size provides information on **how meaningful** the actual difference between two competing techniques is. This is why the Vargha and Delaney effect size measurement is reported together with the p-value [43]. Unlike other effect size tests' like Cohens D [6], Vargha and Delaney's test does not assume a standardised normal distribution. The test is implemented as library of Menzies [25] test implementation. The test returns a standardised effect size (*VD*) range from zero to one. According to Vargha and Delaney, the result *VD(A,B)* of the collected data from two fuzzers *A* and *B* can be interpreted as follows:

- $VD = 1$ → complete stochastic dominance of A over B
- $VD > 0.71$ → huge effect, A is much better than B
- $VD > 0.64$ → medium effect, A is better than B
- $VD > 0.56$ → small effect, , A is slightly better than B
- $0.45 < VD < 0.55$ → no effect
- $VD < 0.44$ → small effect, B is slightly better than A
- $VD < 0.36$ → medium effect, B is better than A
- $VD < 0.29$ → huge effect, B is much better than A
- $VD = 0$ → complete stochastic dominance of B over A

For the sake of simplicity, we report the magnitude of VD from 0.5. In the following the VD ranging from 0.5 to 1 and describe the direction of the effect textual. Consequently, an effect size VD < 0.55 implicit a not noteworthy effect.

4.4 Results

4.4.1 RQ1: Coverage

RQ1 aims to answer the question, whether any implementation of HashedZest achieves a significant higher coverage than Zest. In the context of Zest, the *branch coverage* is recorded [7]. The branch coverage aims to traverse each branch at least once. That is why trace events, mentioned in section 2.4, are recorded in Zest. The fuzzer gathers the information to satisfy branch conditions, guarded by an if-statement for instance, over time. This enables the fuzzer to produce inputs to cover new branches. Zest distinguishes between two types of coverage: valid coverage and total coverage. The valid coverage represents the coverage achieved by valid inputs. These inputs met the syntactical requirements of the target program. The total coverage keeps track of all branches reached.

Target	Algorithm	Mean	Std	Median
ant	Zest	4644.80	36.57	4640
	HashedZest	4656.40	21.80	4653.5
	HashedZest+	4659.35	36.74	4654.5
	RoundRobin	4657.10	27.16	4656
bcel	Zest	6772.05	198.22	6656
	HashedZest	6797.60	212.66	6925.5
	HashedZest+	6700.90	182.83	6596
	RoundRobin	6655.80	235.67	6598
closure	Zest	34492.90	222.16	34461.5
	HashedZest	34622.80	290.94	34520.5
	HashedZest+	34519.05	262.21	34545.5
	RoundRobin	34288.10	310.07	34321.5
maven	Zest	2981.35	121.05	2982
	HashedZest	3010.65	102.12	2993
	HashedZest+	3016.55	107.00	3040
	RoundRobin	2841.40	123.08	2848.5
rhino	Zest	9346.35	96.64	9362
	HashedZest	9412.05	105.53	9386
	HashedZest+	9374.45	121.65	9387.5
	RoundRobin	9346.85	54.69	9348.5

Table 1: Mean, standard deviation and median for total coverage as number of branches detected.

Table 1 lists coverage achieved as mean, standard deviation and median. As mentioned in section 4.1, we use the Fast instrumentation instead of the default Janala. Therefore we compare the absolute number of paths achieved rather than how much of the map is covered. No implementation of HashedZest managed to achieve a significant advantage over Zest. We observed significant differences between HashedZest and its' implementations.

HashedZest+ achieves an advantage over RoundRobin on closure ($p < 0.05$, $VD > 0.64$) and maven ($p < 0.001$, $VD > 0.79$). Thus, we have a medium effect on closure and a huge effect on maven. HashedZest also reaches significant more total coverage than RoundRobin on closure ($p < 0.01$, $VD > 0.64$), maven ($p < 0.001$, $VD > 0.71$) and rhino ($p < 0.05$, $VD < 0.54$). The gap in performance between HashedZest and RoundRobin on rhino can be considered as significant, but not relevant. Zest also performs better on maven compared to RoundRobin ($p < 0.01$, $VD > 0.71$). In the end, RoundRobin lacks in overall coverage compared to Zest, HashedZest and HashedZest+. The other techniques perform similar in terms of total coverage.

Target	**Algorithm**	**Mean**	**Std**	**Median**
ant	Zest	3956.50	195.12	4042
	HashedZest	3976.20	206.37	4084.5
	HashedZest+	3973.05	224.73	4088
	RoundRobin	4050.45	142.93	4073
bcel	Zest	4451.80	32.67	4444
	HashedZest	4471.20	103.51	4444
	HashedZest+	4451.75	46.34	4440
	RoundRobin	4440.85	21.65	4442
closure	Zest	31441.95	1376.92	31618
	HashedZest	31842.50	1386.17	32656
	HashedZest+	31479.85	1382.43	31575.5
	RoundRobin	31522.85	1385.28	32267.5
maven	Zest	2180.30	132.38	2205.5
	HashedZest	2198.30	102.90	2180.5
	HashedZest+	2184.40	143.74	2205.5
	RoundRobin	2058.25	149.44	2054.5
rhino	Zest	8632.10	118.80	8617
	HashedZest	8653.95	141.72	8586
	HashedZest+	8619.10	146.10	8579
	RoundRobin	8577.80	112.10	8549.5

Table 2: Mean, standard deviation and median for valid coverage as number of branches covered by valid Inputs.

In terms of valid coverage the results almost remain the same. Zest shows a significant advantage over RoundRobin with a medium effect on maven ($p < 0.05$, $VD > 0.64$). Compared to RoundRobin, HashedZest ($p < 0.01$, $VD > 0.71$) and HashedZest+ ($p < 0.05$, $VD > 0.71$) achieve a huge effect on maven. What is more, HashedZest also demonstrates medium advantage over RoundRobin on rhino ($p < 0.05$, $VD > 0.64$). Regarding the total coverage, RoundRobin catches up a little bit as no significant disadvantages on closure are recorded. We cannot make any reasonable comparisons between Zest, HashedZest and HashedZest+ due to the lack of significant differences.

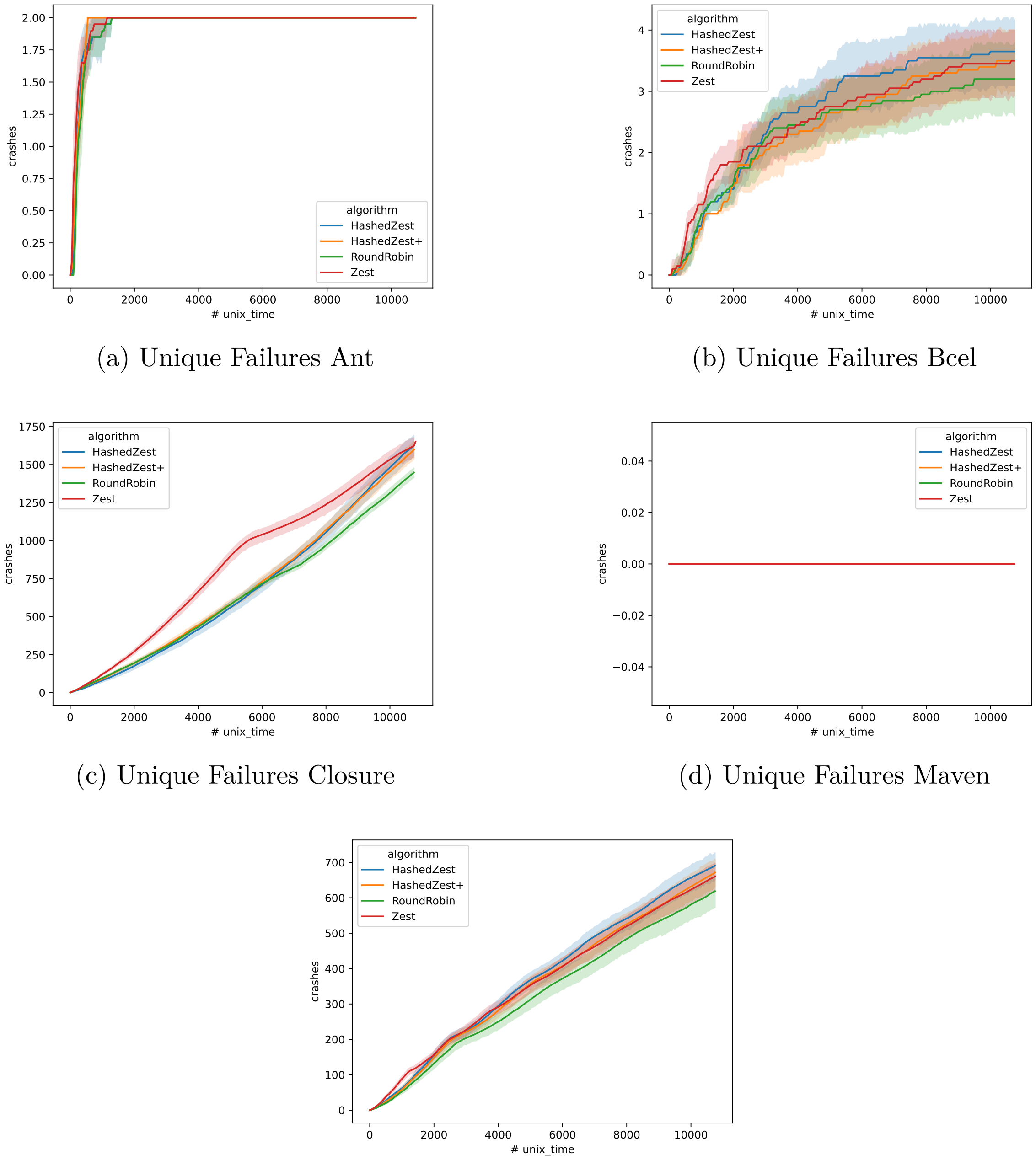
In conclusion, Zest does not benefit from HashFuzz in terms of coverage. At least a minor significant advantage of HashedZest+ over Zest, in terms of valid coverage, was expected. However, HashedZest+ was supposed to fuzz more valid inputs and therefore gain more valid coverage. How many extra inputs are saved by the approaches and the share of valid to total inputs is demonstrated in section 4.3.3. RoundRobin uses the same saving criteria as HashedZest+. Remarkably, Zest, HashedZest and HashedZest+ show a significantly increased performance on closure, maven and rhino compared to RoundRobin. We assume the execution speed of RoundRobin is responsible for this outcome. RoundRobin performs many extra steps in comparison with Zest. In section 4.3.4 we expound the fuzzers performance regarding execution speed.

Menendez and Clark observed statistically significant improvements on their targets, mentioned in section 2.7 [24]. For AFL [47] and Fairfuzz [22], Menendez and Clark claim an advantage provided by HashFuzz. AFL recorded an increased branch coverage of 4.8 %, FairFuzz 1.9 %. The absence of information on the effect size on LibXml2 makes it hard to compare their outcomes with our results. They only deliver the total effect size over all targets of 0.516 for AFL and 0.508 for FairFuzz. Accordingly, the effect on LibXml2 can barely exceed a small advantage of their modified techniques over the regular implementations of AFL and FairFuzz. Consequently, our results fit into this picture.

4.4.2 RQ2: Bugs

RQ2 aims to answer the question, whether any implementation of HashedZest detects significant more bugs than Zest. Regarding Klees et al. [19], counting unique failures requires error deduplication such as described in section 3.1.4. Because error deduplication is implemented for all fuzzers in the *handleResult* method, detected bugs are considered as unique failures.

Figure 10 depicts an overview of bugs detected by Zest, HashedZest, HashedZest+ and RoundRobin over time. Usually, fuzzing ant, maven and bcel results in only a few bugs identified ([20], [36]). On these targets, all introduced techniques perform similar. On bcel, closure and rhino, RoundRobin seems to struggle the more time passes. In relation to ant and maven, more inputs are saved for these targets as illustrated in the following chapter. Maintaining the hash map to save inputs may consumes too many resources.



(a) Unique Failures Ant

(b) Unique Failures Bcel

(c) Unique Failures Closure

(d) Unique Failures Maven

(e) Unique Failures Rhino

Figure 10: The amount of unique failures over time. For each graph illustrates the results over all twenty runs of its' fuzzer

Zests' performance on closure highlights the importance of timeouts for the evaluation of fuzzers. While HashedZest, HashedZest+ and RoundRobin continually grow in detected bugs, Zest exposed more than half of its bugs before half of the given time budget was consumed. Then the graph flattens. When the time budget is consumed, Zest, HashedZest and HashedZest+ are almost equal in the total amount of detected bugs. An a longer run time on each target would be very interesting to see if the course of the graphs continue.

Table 3 depicts the unique failures exposed after three hours. Once again, no implementation of HashedZest can achieve a significant advantage over Zest. What stands out, is the performance of RoundRobin on closure. Zest, HashedZest and HashedZest+ perform significantly better ($p < 0.001$, $VD > 0.71$). Since the $VD = 0.91$ for the fuzzers in comparison with RoundRobin, they almost accomplished stochastic dominance over RoundRobin. HashedZest also demonstrates a significant advantage over RoundRobin on rhino ($p < 0.05$, $VD > 0.56$). Both targets, rhino and closure, dealing with JavaScript files as input.

Target	**Algorithm**	**Mean**	**Std**	**Median**
ant	Zest	2	0	2
	HashedZest	2	0	2
	HashedZest+	2	0	2
	RoundRobin	2	0	2
bcel	Zest	3.50	1.24	4
	HashedZest	3.65	1.24	4
	HashedZest+	3.50	1.16	3
	RoundRobin	3.20	1.33	3
closure	Zest	1628.65	149.81	1654.5
	HashedZest	1630.90	159.16	1655.5
	HashedZest+	1606.60	130.05	1608.5
	RoundRobin	1455.40	71.81	1456.5
maven	Zest	0	0	0
	HashedZest	0	0	0
	HashedZest+	0	0	0
	RoundRobin	0	0	0
rhino	Zest	663.80	91.13	671
	HashedZest	692.85	88.92	702
	HashedZest+	674.15	90.64	677.5
	RoundRobin	621.25	99.82	652.5

Table 3: Mean, standard deviation and median for unique failures exposed.

A possible justification might be RoundRobin struggles with detecting bugs on targets demanding JavaScript files. RoundRobins' modifications are close to HashedZest+. However, HashedZest+ shows no lack in performance in comparison to HashedZest and

Zest. What is more, bcel also processes JavaScript files. The significant differences in performance, recorded on rhino and closure, do not occur here. Consequently, it seems unlikely the significant worse outcomes of RoundRobin can be explained by the type of files a target program needs for this particular case. Another reason might be the handling of many saved inputs, having a negative effect on RoundRobins' execution speed.

Menendez and Clark recorded significant improvements for AFL and FairFuzz in terms of bug detection. The fuzzer combines AFL and HashFuzz exposed eight instead of one bug. FairFuzz increased its crash count from eleven to seventeen. Although AFL found 700% more errors, the effect size over all eight targets equals 0.562 which outlines a small effect. With a VD of 0.539, the impact of HashFuzz on FairFuzz is even less.

To be in line with Menendez and Clarks' outcomes, an implementation of HashedZest used to demonstrate significant advantages on ant or maven. This is not the case. Our observations do not lead to an significant conclusion. Regarding Figure 10, the course of time suggests that HashedZest and its' subversion could benefit from an increased time out. Another possible explanation for the absence of significant results is, Menendez and Clark augmented unstructured fuzzers with HashFuzz. We modified Zest, a highly structured fuzzer that may not benefit as much as an unstructured fuzzer from this approach of input diversification.

While all fuzzers perform alike on ant and maven, they differ on bcel, closure and rhino. To conclude, that fuzzers augmented by HashFuzz only benefit on certain targets, longer trials of each approach are necessary. Also adding new targets, that require different inputs, would substantiate this hypothesis.

4.4.3 RQ3: Ratio valid to total inputs

RQ3 aims to answer the question, whether any implementation of HashedZest generates a significant bigger share of valid to total inputs than Zest. As displayed in 3.4.2, the counter for valid inputs is increased at the *handleResult* method. An input is considered as valid if the testers assertion were not violated and no unexpected exception were thrown. In 3.4.3, the extra saving criterion introduced by HashedZest, was explained. An input is only saved, if it is valid and belongs to an empty partition of the hash map. HashedZest and its' implementations aimed to save extra interesting inputs with a unique binary structure in order to increase Zests' performance. It was assumed, Zest could benefit from more saved valid and diverse inputs in terms of total coverage, valid coverage and bug detection. Section 4.3.1 and 4.3.2 demonstrated, that no significant improvements were recorded. This chapter tries to figure out whether the introduced criterion had a positive impact in terms of saved valid inputs on HashedZest and its' sub versions. Therefore, we distinguish between the overall saved inputs in Table 4 and the ratio of valid to total saved inputs in Table 5.

The most important conclusion drawn from Table 4 is, that HashedZest+ carries out significant advantages over Zest in terms of saved inputs. The advantages are present on all five target programs. With a significance level of p < 0.01 on closure, we can claim HashedZest+ saved more inputs than Zest. Furthermore, a higher significance level of p < 0.001 is achieved for ant (VD > 0.71), bcel (VD > 0.71) , maven (VD > 0.56) and rhino (VD > 0.56). Therefore, the extra saving criterion and its resetting caused at least a medium effect on the saved inputs by a fuzzer over all targets.

Target	**Algorithm**	**Mean**	**Std**	**Median**
ant	Zest	469.60	15.59	475.5
	HashedZest	471.00	17.31	469.5
	HashedZest+	512.75***	16.23***	514.5***
	RoundRobin	507.65***	18.91***	509.5***
bcel	Zest	972.10	20.09	973
	HashedZest	969.70	20.09	969
	HashedZest+	1041.95***	33.78***	1046***
	RoundRobin	1040.20***	28.94***	1040***
closure	Zest	4831.80	155.53	4855
	HashedZest	4794.20	100.76	4810.5
	HashedZest+	4967.70**	110.02**	4970.5**
	RoundRobin	4653.30	149.86	4682.5
maven	Zest	627.55	32.37	637.5
	HashedZest	637.00	28.56	635
	HashedZest+	695.05***	34.98***	692.5***
	RoundRobin	617.95	38.01	617
rhino	Zest	1508.90	48.27	1512.5
	HashedZest	1563.50**	68.59**	1561.5**
	HashedZest+	1690.65***	45.01***	1690***
	RoundRobin	1579.05***	49.08***	1592.5***

Table 4: Mean, standard deviation and median of saved inputs. Outcomes of HashedZest, HashedZest+ and RoundRobin with a significant advantage over Zest, are annotated with *, ** or ***.
* p < 0.05 ** p < 0.01 *** p < 0.001

As mentioned in section 3.1, HashedZest aims to ensure that each partition of the input space is represented at least by one input. Bearing in mind, the queue can consist of thousands of inputs after a three hour run, it was questioned whether eight extra saved inputs would make any difference. But on rhino HashedZest, was able to exceed Zest (p < 0.01, VD > 0.71). Nevertheless, to put the idea of adding extra valid inputs to the queue a little further by implementing HashedZest+, the advantages grow. Remember HashedZest+ enables to re-activate the extra saving criterion every 100 saved inputs.

The comparison of HashedZest and HashedZest+ shows the impact of resetting the saving criteria. HashedZest+ demonstrates a significant improvement over HashedZest on all targets ($p < 0.001$, $VD > 0.71$). Because RoundRobin performed as good as HashedZest+ on ant, bcel and rhino, we cannot claim this advantage regarding all competitors. On one hand, RoundRobin outperformed Zest and HashedZest on ant, bcel and rhino, achieving a huge effect ($p < 0.001$, $VD > 0.71$). On the other hand, RoundRobin lacked in performance on closure and maven compared to Zest and HashedZest ($p < 0.01$, $VD > 0.71$). A possible explanation for the behaviour on closure might be minor execution speed of RoundRobin. In section 4.3.2 we suggested a huge amount of saved inputs could lead to difficulties in handling the hash map. Since RoundRobins' hash map contains less inputs on maven than on rhino or bcel, the deduction seems to be incorrect. The type of demanded input by the target also does not matter because maven uses XML files, closure JavaScript. For both types RoundRobin achieved significant advantages on ant, bcel and rhino. From the recorded data, we cannot provide an explanation for this behaviour.

However, HashedZest+ achieved significant advantage over Zest on all targets. Therefore, the impact of the extra saving criterion to save more valid inputs is significant. As the introduced saving criterion only allows to save valid inputs, we expect that HashedZest+ also provide a significant advantage in terms of the valid to total inputs ratio.

Table 5 illustrates that Zest, HashedZest and HashedZest+ have a similar share of valid to total inputs. Although HashedZest and its implementations saved so many extra inputs, the ration of valid to total inputs remains the same. This is an unexpected outcome, since the extra saving criterion only saves valid inputs. Table 4 shows that HashedZest+ stores significant more inputs in queue than Zest on all targets. As explained at 3.1.4, an input is only saved if its' result is "SUCCESS" or "INVALID". Therefore, the extra amount of saved inputs must have been invalid. In that case, children from the additional saved valid inputs must be evaluated as "INVALID". Thus, they would not be saved because of our introduced saving criterion. A default saving criteria only acknowledge an invalid input as interesting when its *runcoverage* is not a subset of the coverage explored so far. Consequently, the extra invalid saved inputs must have reached new areas of the code. In that case, HashedZest+ used to demonstrate significant improvements in terms of overall coverage, discussed in section 4.3.1. Because the data do not support this hypothesis, we cannot reason the recorded behaviour.

Target	Algorithm	Mean	Std	Median
ant	Zest	0.31	0.04	0.31
	HashedZest	0.30	0.03	0.30
	HashedZest+	0.28	0.04	0.27
	RoundRobin	0.30	0.05	0.29
bcel	Zest	0.24	0.01	0.24
	HashedZest	0.24	0.01	0.23
	HashedZest+	0.23	0.01	0.23
	RoundRobin	0.23	0.01	0.23
closure	Zest	0.55	0.01	0.55
	HashedZest	0.55	0.01	0.55
	HashedZest+	0.55	0.01	0.55
	RoundRobin	0.56***	0.01***	0.56***
maven	Zest	0.18	0.01	0.18
	HashedZest	0.18	0.01	0.18
	HashedZest+	0.17	0.02	0.17
	RoundRobin	0.17	0.02	0.18
rhino	Zest	0.57	0.02	0.58
	HashedZest	0.58	0.02	0.58
	HashedZest+	0.58	0.02	0.58
	RoundRobin	0.59***	0.02***	0.59***

Table 5: Mean, standard deviation and median of the share of valid to all saved inputs. Outcomes of HashedZest, HashedZest+ and RoundRobin with a significant advantage over Zest, are annotated with *, ** or ***.
* p<.05 ** p<.01 *** p<.001

However, RoundRobin achieves significant advantages on closure ($p < 0.001$, VD > 0.71) and rhino ($p < 0.001$, VD > 0.64) compared to Zest. Because RoundRobin also shows significant improvements over Zest in terms of saved inputs on rhino, this is the impact we tried to achieve by all implementations of HashedZest. Furthermore, an improvement in this ratio was expected to lead to more identified bugs or coverage explored. As outlined in section 4.3.1 and 4.3.2, RoundRobin performs significant worse than Zest on closure and rhino. We earlier mentioned that a reduced execution speed could explain this observation. If the data support this assumption, is outlined in the following chapter.

4.4.4 RQ4:

RQ4 investigates the impact of execution speed, caused by the modifications made on Zest in order to implement HashedZest, on the fuzzers ability to detect bugs and branches. Fuzzers rely on many executions to gather information about their target program and explore areas of the code. Therefore, a reduced execution speed caused by HashedZests' functionalities could explain the lack of advantages in terms of coverage and bug detection.

Target	Algorithm	Mean	Std	Median
ant	Zest	34.77	4.27	34.97
	HashedZest	33.96	3.76	33.91
	HashedZest+	36.38	5.84	36.67
	RoundRobin	36.86	4.83	36.59
bcel	Zest	64.58	1.33	64.30
	HashedZest	64.01	1.09	63.89
	HashedZest+	66.73	1.14	66.47
	RoundRobin	64.10	0.83	64.20
closure	Zest	52.43	3.33	52.79
	HashedZest	51.58	3.69	53.08
	HashedZest+	51.20	2.91	51.67
	RoundRobin	48.60	2.04	49.32
maven	Zest	75.65	1.31	75.40
	HashedZest	81.62***	1.90***	81.57***
	HashedZest+	81.95***	1.68***	81.86***
	RoundRobin	63.99	1.55	64.67
rhino	Zest	53.25	3.52	54.35
	HashedZest	56.90**	4.38**	58.06**
	HashedZest+	55.24*	4.36*	56.04*
	RoundRobin	49.11	3.78	50.47

Table 6: Mean, standard deviation and median of executions per second. Outcomes of HashedZest, HashedZest+ and RoundRobin with a significant advantage over Zest, are annotated with *, ** or ***.
* p<.05 ** p<.01 *** p<.001

RoundRobin performs significant worse than Zest on closure ($p < 0.001$, $VD > 0.71$), maven ($p < 0.001$, $VD = 1$) and rhino ($p < 0.001$, $VD > 0.71$). On all three targets, the effect size ranges from VD 0.91 to 1. We conclude the modifications in order to implement RoundRobin have a huge impact on the fuzzers performance. This could explain why RoundRobin missed to achieve significant advantages over Zest in section 4.3.1 and 4.3.2 on closure, maven and rhino. The data do not answer why this lack of speed is not visible on ant and bcel. The amount of saved inputs do not explain this relationship because the map contained less inputs for maven as compared to bcel. On both targets, RoundRobin has a decreased ratio of valid to total inputs while holding a significant advantage in terms of total inputs saved compared to Zest. Regarding closure and rhino, it seems like a higher share of valid inputs slow RoundRobin down. The outcomes for maven do not support this idea. RoundRobin demonstrated a lower share of valid inputs, but also a significant reduced execution speed compared to Zest.

Remarkably, HashedZest and HashedZest+ achieved significant more executions per second on maven ($p < 0.01$, $VD = 1$) and rhino ($p < 0.05$, $VD > 0.71$) than Zest. This results are surprising because the effect is not visible on other target programs. The implementation of HashedZest+ and RoundRobin differ in the methods *saveCurrentInput* and *getInput*. Because we only applied small changes in *saveCurrentInput*, we assume the gap in execution speed stem from the adjustments made in *getInput*. On bcel ($p < 0.001$, $VD > 0.64$), closure ($p < 0.001$, $VD > 0.64$), maven ($p < 0.001$, $VD = 1$) and rhino ($p < 0.001$, $VD > 0.71$) HashedZest+ shows a significant higher execution speed than RoundRobin. However, this relation is not present on ant. Nevertheless, we can conclude the implementation of the *getInput* method is responsible for the gap between HashedZest+ and RoundRobin.

Now all research questions were discussed, it is time to draw final conclusions. Firstly, we perceive RoundRobin as interesting target for further modifications. Although the technique lacks in execution speed on all targets but ant, only two significant disadvantages to Zest were exposed: valid coverage on maven and bug detection on closure. Eliminating the disadvantages in terms of execution speed could lead to significant benefits provided by RoundRobin over Zest.

HashedZest and HashedZest+ perform very similar. Only in terms of saved inputs, a significant gap was recorded. It is not surprising HashedZest+ demonstrates an advantage here since the same criterion used by HashedZest is reset over time. This minor adjustment did not carry out other significant effects.

Neither HashedZest nor HashedZest+ provide benefits compared to Zest. In terms of overall coverage, valid coverage and bug detection, no significant differences were exposed. Regarding Figure 10, a longer duration of the fuzzing runs could lead to further insights whether the observed similarity of Zest, HashedZest and HashedZest+ in terms of bug detection and coverage achieved remain.

4.5 Threats to Validity

4.5.1 Internal Validity

Although we aimed to provide a certain quality of research, there is one part of testing to improve at least. As Klees et al. said, "Fuzzing performance can vary over the course of a run. This means that short timeouts (of less than 5 or 6 hours, [...]) may paint a misleading picture. [...] AFL outperformed AFLFast at 6 hours, with statistical significance, but after 24 hours the trend reversed." [19]. Therefore, running Zest, HashedZest, HashedZest+ and RoundRobin over 24 hours would lead to more meaningful outcomes.

HashedZest tried to increase detected bugs and coverage due to increased diversification of valid input for Zest. We did not observe this correlation, because neither HashedZest nor HashedZest+ displayed improvements in terms of the ratio valid to total inputs. It probably needs a different saving criteria to ensure a higher input diversity. The execution speed is mentioned as confounding variable, because RoundRobin achieved a higher share of valid inputs on certain targets, but due the lack of speed this had no impact on the coverage or bug detection. Furthermore, we only considered the branch coverage in terms of *richness*. Zest recorded, how many branches where traversed at least once. Regarding to Nguyen and Grunske [27], the *evenness* is also important. The evenness expresses the distribution of hits for each branch. To reach a guarded branch often by a novel technique, could be considered as improvement. Recording the evenness of HashedZest, HashedZest+ and RoundRobin may expose strengths we did not observe because of the absence of data.

4.5.2 External Validity

A possible threat to validity is the lack of extra target programs. We used five real world java applications also tested in [34]. The targets are dealing with popular input files such as XML and JavaScript. Using more divers target programs demanding different types of inputs would increase the external validity. Further the tests where conducted at the computing centre of the Humboldt University of Berlin in order to provide a solid external validity. We assume, testers would run our techniques on comparable machines rather than notebooks or desktop PCs'.

4.5.3 Construct Validity

To outline significant results, we use the Mann-Whitney-U-Test to identify significant gaps in certain benchmarks for each algorithm. Thus, the test is used six times on each target. Overall we compute the significance level 30 times over the course of our research. Although the data differs from target the target, the possibility of a false-positive significance must be named as threat to construct validity [2].

5 Conclusion

We introduced the idea of HashedZest as extension for the structured fuzzer Zest. In order to tackle the problem of input diversity, we split the input space into eight parts, as suggested by Menendez and Clark [24]. Three versions of HashedZest were introduced, implemented and tested: HashedZest, HashedZest+ and RoundRobin. HashedZest and HashedZest+ differ in the implementation of the extra saving criterion. RoundRobin maintains a hash map for saved inputs and feeds the queue by switching the partition every time a new parent input is required. We evaluated our findings with the help of Mann-Whitney-U-Test in order to prove differences between the algorithms in terms of our benchmarks overall coverage, valid coverage, bug detection, valid to total inputs ratio and execution speed. The Vargha and Delayney test outlined meaningful results. Finally we cannot demonstrate improvements of at least one implementation of HashedZest in terms of coverage or bug detection over Zest. Nonetheless, the fuzzers achieved remarkable results in terms of execution speed and valid to total input ratio. The implementations of HashedZest were expected to be slower than Zest due to extra calculations regarding hash keys. Remarkably on maven and rhino, HashedZest and HashedZest+ outperformed Zest. What is more, speed in combination with the extra saved valid inputs had no impact on the valid to total input ratio, bug detection or coverage achieved.

Although HashedZest was not shown to be a superior approach, further research should focus on refining the idea of HashFuzz for structural fuzzers such as Zest. For example the input space could be divided into more than eight partitions. This modification needs a different way to calculate hash keys. Every novel technique demonstrated its strengths and weaknesses during the research. Two directions for improvements were carried out in section 4.3.3 and 4.3.4:

1. HashedZest and HashedZest+ achieved an increased execution speed on certain targets, but they do not demonstrate a better ratio of valid to total inputs than Zest. An increased ratio with no harm to the execution speed could lead to significant results.

2. RoundRobin struggles the other way around. The lack in execution speed might prevent significant improvements on performance regarding Zest. The implementation of the *getInput* method is considered to be responsible for this. The fix of this issue with no harm to the share of valid inputs, could also be promising for future work.

Beside the implemented fuzzers, the experimental setup also gives room for improvement. A longer timeout as suggested by Klees et. al. [19] could draw an entire different picture. Namely, the diversified input of HashedZest, HashedZest+ and RoundRobin increased the amount of bugs exposed and valid coverage.
Further research could also benefit from introducing different benchmarks. To compare coverage in terms of evenness, mentioned in section 4.3.1, rather than richness, could also provide further insights into how HashFuzz effects Zest.

References

[1] Akcura, K., Shalchian, R., Patil, A., Singh, R., and Tanna, J. Static versus dynamic source code analysis.

[2] Bender, R., and Lange, S. Adjusting for multiple testing—when and how? *Journal of clinical epidemiology 54*, 4 (2001), 343–349.

[3] Böhme, M., Szekeres, L., and Metzman, J. On the reliability of coverage-based fuzzer benchmarking. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE* (2022), vol. 22.

[4] Chakraborty, S., Meel, K. S., and Vardi, M. Y. A scalable approximate model counter. In *Principles and Practice of Constraint Programming* (Berlin, Heidelberg, 2013), C. Schulte, Ed., Springer Berlin Heidelberg, pp. 200–216.

[5] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not. 35*, 9 (sep 2000), 268–279.

[6] Cohen, J. *Statistical Power Analysis for the Behavioral Sciences.* Routledge, 1988.

[7] Dr. Poehler, S. Grundlagen der code-coverage-messung.

[8] Durango, A., and Refugio, C. An empirical study on wilcoxon signed rank test, 12 2018.

[9] Eid, M., Gollwitzer, M., and Schmitt, M. *Statistik und Forschungsmethoden: Lehrbuch. Mit Online-Materialien.* Beltz, 2013.

[10] Geiger, F. `https://statologie.de/mann-whitney-u-test-python/`, 2020. Accessed: 10.01.2023.

[11] Gomes, C. P., Sabharwal, A., and Selman, B. Near-uniform sampling of combinatorial spaces using xor constraints. In *Advances in Neural Information Processing Systems* (2006), B. Schölkopf, J. Platt, and T. Hoffman, Eds., vol. 19, MIT Press.

[12] Google. `https://github.com/google/re2`, 2010. Accessed: 12.01.2023.

[13] Gopinath, R., and Zeller, A. Building fast fuzzers. *CoRR abs/1911.07707* (2019).

[14] Hazel, P. `https://www.pcre.org/`, 1997. Accessed: 12.01.2022.

[15] Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., and Hosking, A. L. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2021), ISSTA 2021, Association for Computing Machinery, p. 230–243.

[16] Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., and Hosking, A. L. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2021), ISSTA 2021, Association for Computing Machinery, p. 230–243.

[17] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, Aug. 2012), USENIX Association, pp. 445–458.

[18] Holser, P. `https://github.com/pholser/junit-quickcheck`, 2020. Accessed: 02.12.2022.

[19] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. *CoRR abs/1808.09700* (2018).

[20] Kukucka, J., Pina, L., Ammann, P., and Bell, J. Confetti: Amplifying concolic guidance for fuzzers. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), pp. 438–450.

[21] Lemieux, C., Padhye, R., Sen, K., and Song, D. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2018), ISSTA 2018, Association for Computing Machinery, p. 254–265.

[22] Lemieux, C., and Sen, K. *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage.* Association for Computing Machinery, New York, NY, USA, 2018, p. 475–485.

[23] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering 47*, 11 (2021), 2312–2331.

[24] Menendez, H. D., and Clark, D. Hashing fuzzing: Introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering 48*, 9 (2022), 3540–3553.

[25] Menzies, T. `https://gist.github.com/timm/5630491`. Accessed: 10.01.2023.

[26] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33*, 12 (dec 1990), 32–44.

[27] Nguyen, H. L., and Grunske, L. BeDivFuzz. In *Proceedings of the 44th International Conference on Software Engineering* (may 2022), ACM.

[28] Nyberg, F., and Skogeby, J. Code quality & quality gates.

[29] OGHENEOVO, E. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications 02* (01 2014), 1–16.

[30] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)* (2007), pp. 75–84.

[31] PADHYE, R. `https://github.com/rohanpadhye/JQF/wiki/Implementation-details`, 2021. Accessed: 03.01.2023.

[32] PADHYE, R. `https://github.com/rohanpadhye/jqf#tutorials`, 2022. Accessed: 19.09.2022.

[33] PADHYE, R., LEMIEUX, C., AND SEN, K. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2019), ISSTA 2019, Association for Computing Machinery, p. 398–401.

[34] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND LE TRAON, Y. *Semantic Fuzzing with Zest.* Association for Computing Machinery, New York, NY, USA, 2019, p. 329–340.

[35] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND TRAON, Y. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv preprint arXiv:1812.00078* (2018).

[36] REDDY, S., LEMIEUX, C., PADHYE, R., AND SEN, K. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), ICSE '20, Association for Computing Machinery, p. 1410–1421. Accessed: 06.01.2023.

[37] SCHALNAT, G. E. `http://www.libpng.org/pub/png/libpng.html`, 2004. Accessed: 12.01.2023.

[38] SEREBRYANY, K. libfuzzer a library for coverage-guided fuzz testing, 2015.

[39] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012* (2012).

[40] SHE, D., SHAH, A., AND JANA, S. Effective seed scheduling for fuzzing with graph centrality analysis, 2022.

[41] SIROS, I. `https://www.esat.kuleuven.be/cosic/blog/wireless-network-protocol-fuzzing/`, 2022. Accessed:04.01.2023.

[42] STEINBERGER, D. `https://c-ares.org/download/`, 2004. Accessed: 12.01.2023.

[43] VARGHA, A., AND DELANEY, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics 25*, 2 (2000), 101–132.

[44] VEILARD, D. `xmlsoft.org`, 1999. Accessed: 12.01.2023.

[45] VIKRAM, V., PADHYE, R., AND SEN, K. *Growing A Test Corpus with Bonsai Fuzzing.* IEEE Press, 2021, p. 723–735.

[46] WANG, Y., WU, Z., WEI, Q., AND WANG, Q. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access 7* (2019), 36340–36352.

[47] ZALEWSKI, M. American fuzzy loop. `https://lcamtuf.coredump.cx/afl/`, 2017. Accessed: 09.10.2022.

[48] ZHANG, Q., WANG, J., GULZAR, M. A., PADHYE, R., AND KIM, M. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2020), pp. 722–733.

Declaration of Authorship

I hereby confirm that I have authored this Bachelor's thesis independently and without use of others than the indicated sources. All passages which are literally or in general matter taken out of publications or other sources are marked as such.

Berlin, January 19, 2023