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

Quickly Filtering Inputs for Semantic Fuzzing

Bachelorarbeit

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

eingereicht von:	Felix Kummer
geboren am:	02.08.1998
geboren in:	Berlin
Gutachter/innen:	Prof. Dr. Lars Grunske
	Dr. Marcel Böhme

eingereicht am: 02.08.2020 verteidigt am: ....

Contents

1 Abstract

Emerging from recent research in software testing, fuzzing has become a popular and well-studied topic. Coverage-guided mutation-based fuzzing is particularly successful because of its ability to balance overhead and fuzzing capabilities. However, little research has focussed on supplying coverage-guided mutation-based fuzzers with meaningful seed corpora to bootstrap the fuzzing campaign. Previous approaches to seed corpora generation are not applicable in situations with limit resources and can't be adopted to fuzzers that don't leverage inputs as seeds. Alleviating these shortcomings, we propose a novel technique for quickly generating seed corpora for a fuzzer that uses seeds in the form of sequences of bytes that control its generators. Moreover, we develop four filtering techniques that minimize a set of inputs generated by a fast blackbox fuzzer and transform these inputs into seeds. We evaluate our approach on four real-world programs and report an increase in efficiency of discovering new failures and new coverage.

2 Introduction

In recent advances in software testing, fuzzing has emerged as a promising technique for finding bugs, resource bottlenecks and other unwanted or unexpected program behaviour. Fuzzing can be described as the process of randomly generating inputs to a program under test (PUT), executing the PUT with these inputs, collecting execution information and using this information to adapt the generation procedure for future inputs. In particular, greybox fuzzing has attracted industrial and scientific attention because of its ability to balance overhead and fuzzing capabilities. Greybox fuzzers apply lightweight techniques to obtain execution information such as *coverage* to determine the ability of inputs to trigger diverse behaviour in the PUT. Coverage is an approximate measurement of the parts of the PUT that were executed by an input. A particularly successful technique in the field of greybox fuzzing is *mutation-based fuzzing* that creates novel inputs by applying mutations to previous inputs.
In this work, we want to focus on the recently proposed greybox fuzzer Zest [45]. Zest is a mutation-based fuzzer, but also builds on top of ideas from the field of *property-based testing*. Concretely, inputs in Zest are created by generators which base their decisions on *parameter sequences*. Instead of mutating inputs directly, Zest mutates parameter sequences that are then used by a generator to create inputs.
We want to improve Zest by supplying it with a *seed corpus*. Seed corpora form the starting point of mutation-based fuzzers and contain a number of *seeds*. Seeds are used as initial inputs for mutation. Most fuzzers can be initialized without seeds, in which case they randomly generate first inputs out of thin air. However, in [52], [25] and [26] researchers have established that a large seed corpus can increase the ability of a fuzzer to cover diverse behaviour in the PUT and to uncover bugs. Moreover, they found that a *minimized seed corpus* is the best choice for bootstrapping a mutation-based fuzzer. Seed corpora minimization is the current state of the art for supplying a mutation-based fuzzer with seeds. The approach gets a large initial set of inputs that is usually crawled from the internet. Next, it tries to find a minimal subset of the initial set that maximizes the coverage in the PUT. Prominently, afl-cmin is part of the highly successful mutation-based greybox fuzzer AFL [67]. The tool executes every input in the initial set of inputs, collects coverage for each input, and adds an input to the seed corpus if it revealed new coverage. Other approaches have improved seed corpora minimization in different ways (e.g. [52], [25] [26]), but its foundations remain the collection of an initial set of inputs, the execution of the initial inputs to obtain coverage information and the minimization using the coverage information.
Recently, Herrera et al. [26] conducted a survey on the use of seed corpora in current fuzzing research. They found that a majority of fuzzing papers exclude information on seed corpora, don't explain the selection of the concrete corpus or start with empty or manually crafted corpus. They further found that this may lead to results that are biased based on the selection of the concrete corpus. We believe that the lack of examination of seed corpora is caused by the little research on the topic and the infeasibility of seed corpora minimization in some situations. Precisely, seed corpora minimization requires a large set of inputs to be available in the first place. Such an initial set might not be

available or infeasible to obtain due to resource restrictions. Another limitation of seed corpora minimization is that it requires the execution of each input in the initial set to obtain coverage data. This process can be extremely time-consuming for large sets of initial inputs. For our work, we want to supply Zest with an initial set of seeds. This task reveals another shortcoming of previous work on seed corpora minimization. Specifically, previous research focussed on seeds that are inputs to the PUT. However, Zest requires seeds to be parameter sequences. We also need to mention that Zest itself does not provide a procedure to create initial seeds (i.e. initial parameter sequences).
We tackle the problem of initial seed corpora from a new angle. Instead of minimizing a set of initial inputs, we automatically generate an initial seed corpus. First, our approach runs the quick blackbox fuzzer RLCheck [53] to obtain a large set of initial inputs. Blackbox fuzzers are known to generate orders of magnitude more inputs than greybox fuzzers like Zest in the same amount of time. Therefore, using RLCheck allows us to create inputs quickly instead of crawling inputs from the internet. We should note that RLCheck belongs to the same line of research as Zest, and thus also leverages generators to create inputs. Nevertheless, RLCheck also operates on inputs instead of parameter sequences. Therefore, we develop a technique to obtain parameter sequences alongside inputs generated in RLCheck. We call this part of our approach *transformation*. Transformation solves the issue of Zest requiring parameter sequences as seeds. We now have a large set of parameter sequences that we could hand to Zest as its seeds. However, previous research ([52], [25], [26]) suggests that minimizing the seed corpora yields better results in fuzzing. Moreover, blackbox fuzzers like RLCheck exploit little information on the execution of inputs, which leads to these fuzzers generating many similar inputs. Therefore, we want to minimize the set of parameter sequences by *filtering* them. Contrary to seed corpora minimization, we want to avoid the time-consuming execution of the PUT with the seeds to collect coverage. Therefore, we exploit data that we collect alongside the generation process of inputs in RLCheck to filter the set of parameter sequences. We design four filtering techniques that exploit different data collected in RLCheck's generators.
In order to evaluate our new approach, we compare it against Zest without seeds on four real-world programs that use two widely-used file formats (JavaScript and XML). Moreover, we discover that our new approach can significantly increase Zest's efficiency of uncovering new bugs and new coverage. This observation suggests that our approach is particularly useful when faced with limited resources for testing. However, we also observe that our approach can't significantly outperform Zest in terms of total coverage. We explain this observation by considering that we don't modify Zest's fuzzing loop and its internal mechanics, which enables Zest to find the same coverage as in our approach, given enough time.
In short, we make the following contributions:

- We design and implement a method for constructing parameter sequences inside the generators of RLCheck.
- We design and implement four methods for filtering parameter sequences based on three different types of data collected from RLCheck's generators.

- We evaluate our approach on a set of four real-world programs and determine if our approach can improve the fuzzing capabilities of Zest.

In the following sections, we will present the background of our approach and name related work (section 3), explain our approach in detail (section 4), present the experimental setup and results of the evaluation of our approach (section 5) , conclude our work and discuss future work. (section 6).

3 Background and Related Work

3.1 Fuzzing

Fuzzing is a software testing technique that lately received a lot of scientific attention due to its ability to consistently uncover faults in software. First described in 1990 by Miller et al. [40], fuzzing has developed into an important testing technique applied on large scale by big tech companies (e.g. [55]). The basic idea of fuzzing is to create random inputs to a program under test (PUT), observe the execution of the PUT with these inputs, and use the obtained information to construct new inputs. We do not aim to give a full picture of the field of fuzzing, but rather give a broad overview. For a more detailed view on the current state of the art of fuzzing, we refer to the recent survey [39]. Fuzzing can be classified in many ways. Typically, fuzzing is classified by the amount and type of information collected during execution (i.e. blackbox, greybox and whitebox fuzzing) or the way how new inputs are generated (i.e. generation-based and mutation-based).

Blackbox Fuzzing Collecting minimal information about the execution of the PUT is the main property of blackbox fuzzing (e.g. [28], [53], [27], [2], [54]). Typically, blackbox fuzzers only observe if an input caused the PUT to crash. However, some blackbox fuzzers exploit other sources of information to drive the fuzzing process. The advantage of blackbox fuzzing is its ability to generate numerous inputs in a short period of time. Blackbox fuzzers can often generate orders of magnitude more inputs than other approaches, but suffers from its limited ability to exploit information about the execution of the PUT. In other words, blackbox fuzzers can quickly generate many inputs, but the quality of these inputs is often underwhelming, leading to the repeated execution of the same parts of the PUT or getting stuck in error-handling code.

Whitebox Fuzzing Contrary to the little information exploited in blackbox fuzzing, whitebox fuzzers leverage detailed and almost perfect knowledge about the execution of the PUT. The aim of whitebox fuzzing is to trigger rare and hard-to-reach branches in the PUT.
Symbolic execution (e.g. [18], [30], [23], [10], [7], [14], [9], [66]) is a prominent whitebox fuzzing techniques. Symbolic executors collect *path constraints* along the execution path of inputs. Path constraints are sets of logical expressions that describe conditions that input variables have to satisfy to execute a certain path. By negating individual elements

in the path constraints, new paths can be discovered. This method can be used to systematically exercise *all* paths in a PUT. However, real-world PUT's often contain an enormous amount of unique paths, leading to scalability issues and the *path explosion problem*.
Another whitebox approach is *taint analysis* (e.g. [21], [64]), which aims to observe memory segments that are influenced by input variables. Concretely, memory segments that contain input variables are marked (i.e. they become tainted) and whenever data is read from these segments and flows into other parts of the memory, those new parts are marked as well. Additionally, taint analysis also collects the operations performed on tainted memory segments. Using this information allows fuzzers to observer conditional statements that are influenced by the input variables and change inputs variables to alternate the outcome of these conditions. As with symbolic execution, this approach suffers from scalability issues when applied to large PUT's because of its heavy use of program insights.
Overall, whitebox fuzzing can often effectively generate inputs that trigger diverse program behaviour, but fail to scale to real-world applications because of the number of unique paths.

Greybox Fuzzing Blackbox fuzzing and whitebox fuzzing offer their own advantages, but both also suffer from different limitations. A highly successful trade-off between exploitation of program insights and quickness in generating inputs is *greybox fuzzing*. Greybox fuzzers (e.g [67], [6], [46], [45] [50], [15], [34], [35], [12], [62], [11], [51], [36]) owe their success to their ability to balance fuzzing overhead and fuzzing capabilities. Usually, greybox fuzzers collect less fine-grained execution information than whitebox fuzzers while collection more sophisticated execution information than blackbox fuzzers.
A common approach in greybox fuzzing is to collect *code coverage* during execution. Code coverage is an approximate representation of the parts of the PUT that are executed. Collection of code coverage is carried out, through the use of *instrumentation*. Instrumentation is a lightweight technique that inserts additional code into the PUT that emits signals when specific parts of the PUT are executed. Depending on the instrumented locations, different coverage measures can be extracted (e.g. branch coverage, function coverage or basic block coverage). When a fuzzers leverages this coverage data to select inputs for further fuzzing, the fuzzer is considered to be *coverage-guided*. Algorithm 1 shows an example of coverage-guided fuzzing that we will explain later on.

Generation-based Fuzzing and Mutation-based Fuzzing Generation of new inputs is a key part of fuzzing, and fuzzers take different approaches to archive this.
Inputs in *generation-based fuzzing* are created using some *input format specification*. Common approaches use generators (e.g. [53], [17], [22], [45]) or grammars (e.g. [56], [61], [24], [19], [28], [57], [36]) for the production of new inputs. Generators directly produce inputs, while grammars are commonly passed to a procedure that generates inputs using the production rules of the grammar.
Another approach that has been very popular in recent research is *mutation-based fuzzing*

(e.g. [67], [6], [11], [46], [50], [15], [51], [34], [35], [12], [62]). The key idea is to apply mutations to an existing input. Mutations are commonly applied to inputs that satisfied some criteria. As an example, AFL [67] mutates inputs that previously increased coverage. Common mutation operators include bit flips, insertion of specific predefined bytes, random byte insertion or removal of bytes.

Hybrid Approaches Classifying fuzzer using the previously mentioned categories is not always possible. Moreover, there are many *hybrid approaches* to fuzzing that combine different ideas.
Fuzzers like Zest [45], LangFuzz [28] or Nautilus [8] combine mutation strategies with generation-based ideas to create new inputs.
Another idea to hybridize different fuzzing approaches is to combine the ability of whitebox fuzzers to explore all feasible paths with greybox fuzzing that can explore diverse behaviour in the PUT more quickly (e.g. [60], [43]). Notably, Driller [58] tries to combine the ability of symbolic execution to overcome strict token comparisons with the ability to quickly explore a large area of the PUT possessed by many greybox fuzzers. Moreover, Driller partitions the PUT into *compartments* that are separated by strict token comparison. Inside compartments, Driller relies on a mutational greybox fuzzer to quickly explore many paths. When the greybox fuzzer can't increase coverage any more, Driller switches to a symbolic executor to possibly overcome hard comparisons. Once the symbolic executor archives new coverage, the fuzzers takes over again and so on.

Mutation-based Coverage-guided fuzzing In this work we want to improve Zest [45] which is a coverage-guided fuzzer that incorporates both mutation- and generation-based fuzzing. Therefore, we want to present how a coverage guided mutation-based fuzzer works and present the generation-based part of Zest in section 3.3. Algorithm 1 presents a conceptual overview of the coverage-guided mutation-based fuzzing approach. The fuzzer gets a program that it should test as its input. This program is commonly called "program under test" (PUT). Additionally, a set of initial inputs called *seeds* is handed to the fuzzer. First, the fuzzer initializes a set of inputs that should be mutated favourably by adding all seeds to the set (line 1). Inputs in this set are called *interesting inputs*. Next, a set is initialized that should store the coverage that was emitted by all executed inputs (line 2). Line 3 starts the process that is often called the *fuzzing loop*. The fuzzing loop is executed until a stopping criterion is reached. Common stopping criterion are timeouts or the first discovery of a bug in the PUT. Inside the fuzzing loop, a candidate for mutation is selected from the set of interesting inputs (line 4). This selection is often referred to as *seed selection*, and the procedure that schedules seed selection is known as *seed scheduling*. In line 5 a so-called *energy* is assigned to the candidate according to a *power schedule*. Power scheduling determines how many mutations are applied to a candidate. Next, the candidate is mutated according to its energy (line 6) and executed with the PUT to obtain coverage data and the result of the execution (line 7). If the execution resulted in a failure, a failure handling routine is executed (line 8 and 9). Failure handling would typically save the failure's stack trace and the corresponding

Algorithm 1 Mutation-based Coverage-guided Fuzzing

```
    Input: $\mathcal{P}$                                                    ▷ $\mathcal{P}$ Program under test
    Input: $\mathcal{S}$                                                    ▷ $\mathcal{S}$ Set of initial seeds
 1: interestingInputs $\leftarrow \mathcal{S}$                             ▷ Set of inputs that should be mutated
 2: totalCoverage $\leftarrow \emptyset$                                   ▷ Current coverage of all executed inputs
 3: while !stoppingCriterion do
 4:     candidate $\leftarrow Select(\mathcal{S})$
 5:     energy $\leftarrow AssignEnergy$(candidate)
 6:     mutant $\leftarrow Mutate$(energy, candidate)
 7:     coverage, result $\leftarrow run(\mathcal{P}$, mutant)
 8:     if result == FAILURE then
 9:         $handleFailure()$
10:     else if |totalCoverage $\cup$ coverage | $>$ | totalCoverage | then
11:         interestingInputs $\leftarrow$ interestingInputs $\cup$ {mutant}
12:         totalCoverage $\leftarrow$ totalCoverage $\cup$ coverage
13:     end if
14: end while
```

input that produced it, but can also perform approach-specific tasks. Lines 10,11 and 12 are responsible for handling coverage data. If an input archived novel coverage, it is added to the list of interesting inputs and its coverage is added to the set of known coverage. For our approach, we will not modify the core concept of the fuzzing loop, but instead focus on supplying the fuzzing process with a set of seeds.

3.2 Boosting Fuzzing

As a popular subject of research, fuzzing has been improved by numerous ideas. In this section, we will give an overview of existing improvements, focusing mainly on coverage-guided mutation-based fuzzing. We note that every work presented in this section is orthogonal to our new technique. Specifically, we focus on initial seeds, while the approaches in this section improve other parts of the fuzzing loop.

Power Schedules and Seed Schedules In [11] Böhme et al. present Entropic that applies principles from the field of *information theory* to mutation-based greybox fuzzing. Concretely, they measure the *information gain* that an input archives when it is executed with the PUT. The information gain is then used to assign energy to inputs, which assembles a new power schedule.
In AFLFast Böhme et al. [12] modelled the power scheduling of coverage-guided mutation-based greybox fuzzing as *markov chains*. Furthermore, they discovered that classic mutation-based fuzzers like AFL [67] tend to generate inputs, trough mutation, that continuously execute *high density regions*. To overcome this, they introduce a novel power schedule. For a given input, energy is assigned that is inversely proportional to the number of inputs that previously executed the path of the given input. Furthermore,

they also exponentially increase the amount of offspring from the input every time it is selected for mutation.
In [63] Wang et al. designed *multi-level coverage metrics* that combine strengths of differently coarse coverage metrics to select inputs for further mutations. They organize inputs in a cluster tree that includes more fine-grained coverage measures in deeper levels of the tree. Moreover, they interpret seed scheduling as traversing this cluster tree and model the seed scheduling problem as a multi-armed bandit problem that they solve with existing algorithms.
Cerebro [36] elaborates both on power schedules and seed schedules. First, Cerebro proposes a novel *multi-objective online algorithm* for seed scheduling that uses coverage and code complexity among other objectives. For power scheduling, Li et al. proposes an approach that adaptively rates the *potential* of an input to uncover new coverage and uses this potential as the criterion for power scheduling.

Mutation Strategies AFLSMART [50] is an approach that tries to improve the mutation strategy of coverage-guided mutation-based fuzzing by enhancing it with information on the expected file format of inputs. In order to make mutations *structure-aware*, they exploit an input file format specification, which increases the probability of generating valid inputs. Moreover, AFLSMART also defines a new *validity-based power schedule* that assigns more energy to inputs are likely to be more valid than others.
FairFuzz [34] focuses on biasing mutations to produce inputs that execute rare branches. Whenever an input triggers a rare branch, the input is mutated at each possible location to determine which mutations can be applied while still reaching the new rare branch. This information is forged into *mutation masks* that restrict mutations. Using those masks allows FairFuzz to focus on rare branches.
Superion [62] tries to *overcome the grammar-blindness* of mutation strategies. Concretely, inputs are parsed into their abstract syntax trees (ASTs) which are then mutated. Superion mutates ASTs by exchanging entire subtrees with subtrees present in other inputs in the fuzzing queue.

Overcoming Hard Conditional Statements Angora [15] extends branch coverage by enriching it with *context* on (un)visited branches. Moreover, Angora keeps track of unvisited branches and collects *path constraints* for these branches using scalable bit-level taint tracking. The approach applies a search algorithm based on gradient-descent to solve path constraint that they model as constrained functions.
VUzzer [51] leverages dynamic taint analysis to track bytes that flow into magic byte comparisons to enable targeted mutation of those bytes. Furthermore, VUzzer also incorporates a fitness function that weights inputs according to the basic blocks they execute. The approach uses a static analyser to assign weights to basic blocks according to the probability of reaching them, and assigns low weights to error handling code.
Matryoshka [16] identifies control flow dependent and taint flow dependent conditional statements of a targeted conditional statement. Moreover, Chen et al. propose three techniques to simultaneously fulfil all dependent conditional statements.

Steelix [35] tries to overcome magic byte comparisons without dynamic taint analysis or symbolic execution. Moreover, Steelix first identifies interesting magic byte comparisons (e.g. multiple byte magic bytes) in assembler code. Leveraging lightweight binary instrumentation and the information about interesting magic byte comparisons enables Steelix to fuzz these comparisons. Furthermore, Steelix guides fuzzing by using *comparison progress* and coverage. Comparison progress is the progress in satisfying magic byte comparisons.
T-Fuzz [49] takes a different approach to overcome hard conditional statements. Concretely, T-Fuzz *removes hard conditional statements* when the fuzzer can't solve them and continues with the transformed PUT (i.e. *transformation fuzzing*). This approach will lead to false positives. Therefore, T-Fuzz leverages a symbolic execution-based post-processing approach to eliminate false positives.

Other Boosting Techniques FuzzFactory [46] tries to make mutation-based coverage guided fuzzing applicable to any domain. The technique allows testers to specify domain-specific properties (i.e. *waypoints*) that interesting inputs should possess. Moreover, FuzzFactory guides its fuzzing campaign using multiple waypoints and classical coverage measures.
In [24] Godefroid et al. proposes a method for training a generative neuronal network with inputs to learn an input grammar and generate novel inputs from the learned model. Furthermore, the paper introduces the *learn and fuzz-challenge* for machine learning used in fuzzing. This challenge represents the conflict between learning a good model while also generating malformed inputs that may violate the model. They tackle this challenge by allowing the model to generate tokens that does not correspond to the best learned token at certain points in the generation process.
In [20] Fioraldi et al. improves the ability of fuzzers to detect failures that are dependent on specific *program states*. Moreover, they represent the program states with *likely invariants* learned from a corpus of inputs. These likely invariant describe common relations between variables in the PUT and can be negated to reach new program states.

3.3 Zest and JQF

The target for our novel seed generation technique is the coverage-guided state-of-the-art fuzzer Zest [45]. Zest is implemented in the fuzzing framework JQF [44] that enables coverage guided fuzzing for Java programs. In this section we will present the concept of Zest accompanied by its implementation in JQF. Zest is JQF's default approach, so distinguishing between the ideas behind Zest and the implementation in JQF is not always possible. JQF is highly customizable and is already delivered with alternative approaches to Zest (e.g. an AFL [67] implementation).
Zest inherits from two mayor lines of research, namely *mutation-based coverage-guided fuzzing* and *property-based testing*. We already discussed mutation-based coverage-guided fuzzing in section 3.1, but Zest takes a slightly different angle to this approach. Conceptually, Zest works in a loop consisting of a *guidance* generating *parameter sequences*, a *parametric generator* consuming those sequences to produce inputs and a test runner

executing inputs to create coverage data that is led back into the guidance. In the following paragraphs, we will first describe the research line of property-based testing and then describe the individual components in detail.

Property-based Testing *Property-based testing* (e.g. [48], [17], [37], [38], [33]) introduces the idea to test for faults that don't cause the PUT to crash. Classical testing will only report a bug when the PUT crashes or fails to terminate. With this testing approach, errors that don't crash the PUT are not detectable. As an example, consider a simple calculator implementation that produces a wrong result for additions if a specific value is used as an operand. Such fault would not cause a crash of the calculator, but rather terminate with a wrong result. Therefore, property-based testing defines *properties* to find fault that have no classical indicators. Properties can be described as specifications of expected input-output behaviour. That is, properties define a set of *assumptions* and *assertions*. If an input fulfils all assumptions and fails to fulfil any assertion, it is considered to violate a property and thus a fault is detected. To generate inputs, property-based testing approaches utilize some kind of *input format specification.*The input format specification can be a textual specification that is then used by some procedure to generate inputs, or it could be an implemented unit that directly generates inputs, such as QuickCheck's [17] generators. Using the input format specification allows to automatically generate inputs, execute them with the PUT and check the properties. Zest uses generators and also allows testing for properties.
As Zest uses both mutations and generators, it is both a mutation-based and a generation-based fuzzer.

Parametric Generators Inspired by property-based testing approaches like QuickCheck [5], Zest relies on generators to create novel inputs. The advantage of using generators is that they have embedded information about the expected syntax of a specific file type, which allows Zest to consistently generate syntactically valid inputs. This is particularly useful when a PUT expects highly structured inputs like XML files. Generating completely random inputs can hardly create any syntactically valid inputs for such structured formats. Generators in QuickCheck use randomness to create inputs. Moreover, when a generator has to make a choice, it requests bytes from a source of randomness to determine the choice. The nondeterminism introduced by the use of randomness can create diverse inputs, but also leads to many unusable or redundant inputs. To alleviate those shortcomings, Zest introduces *parametric generators* that are deterministic and dependent on *parameter sequences*. Specifically, parametric generators get a parameter sequence as their source of randomness. Parameter sequences are sequences of bytes constructed in the Zest's guidance to enforce a specific sequence of choices in the generator. The authors of Zest made the observation that mutating a byte in a parameter sequence and handing it to a generator leads to a change in the input spice while maintaining syntactical validity. Using this insight allows Zest to mutate parameter sequences and feed them into generators that create syntactically valid inputs.

Conversely, other state-of-the-art mutation-based fuzzers like AFL [67] or libFuzzer [6] usually mutate inputs directly. Directly mutating input bytes leads to many invalid inputs that only execute error-handling code and thus reduce the time to fuzz the core program logic.
Generators in JQF get a SourceOfRandomness object as an input. This object does not necessarily have to be a parameter sequence (e.g. it could just be a stream of random bytes). Thus, other approaches can make different use of generators than Zest. To modify the input generation, users can write own generators and also modify the way generators make choices. As an example, RLCheck [53], a JQF extension, implements new generators that make choices based on policies obtained through reinforcement learning.

Property-based Tests and Test Runners JQF allows tester to specify own unit test in the nature of property-based testing. Property-based testing in JQF is enabled through the JUnit-quickcheck [5] extension of JUnit [4]. Tester can specify assumptions and assertions to define properties inside test methods. We note that assumptions are commonly used in PUT's to ensure syntactic validity, meaning that users can extend existing validity constraint by adding own assumptions. Additionally, tester can annotate test methods with generators that should be used to automatically create inputs (e.g. parametric generators for Zest). Generators used in test methods can be implemented by the tester, selected from a range of generators provided by JQF, or automatically selected by JQF if the test method is annotated with type instead of a generator. Coverage guided fuzzing is enabled through instrumentation of the PUT. JQF builds its instrumentation on top of the Java bytecode manipulation tool ASM [1]. By default, *branch coverage* is collected. Branch coverage is obtained by instrumenting every point in the PUT where the control flow can diverge. In other words, if two inputs execute two different branches after a conditional statement, they have different branch coverage. After the PUT is instrumented and the generator has created an input, JQF executes the PUT with the input, collects coverage date and return this information to the fuzzing guidance.
Execution of the PUT is done using test runners provided by JUnit. Test runners will emit execution results. When JUnit executes an input it can have three results: success, failure or invalid. Success as the result means that JQF executed the PUT without encountering an error and all assertion and assumptions were fulfilled. If the result is invalid, then JQF detected a violation of an assumption. Lastly, a failure result means that JQF encountered an unexpected error during execution (including violation of assertions).
A general idea of Zest [44] is to split the program under test (PUT) into a *syntactic analysis stage* and a *semantic analysis stage*. The syntactic analysis stage is where the input is checked for syntactic validity. Zest targets the semantic analysis stage, where the core program logic is situated. In the previous paragraph, we discussed how Zest uses generators to create syntactically valid inputs. In reality, not all inputs generated by a generator are considered syntactically valid. This invalidity has different reasons, reaching from incorrect generators, PUT's that don't support all features of a given file

format or a specific definition of validity inside the test methods. Zest therefore aims to use validity information as an additional selection criterion for inputs for further fuzzing. To archive this, Zest makes use of JQF's property-based testing abilities. If an input fails an assumption, it is considered to be invalid. Using this feedback allows Zest to prioritize fuzzing valid inputs that exercise program logic.

Driving Fuzzing with Guidances Guidances are where the main fuzzing algorithm is executed. They implement the majority of the fuzzing loop, including seed selection, mutation and power scheduling. Guidances pass information to the generators and process coverage data and execution results obtained from the test runners. Tester can define their own guidances to add novel fuzzing algorithms to the JQF framework. Concretely, JQF provides a guidance interface containing four methods that should be implemented. Two of those methods are used to get data from the guidance that is used to create new inputs. Moreover, this data could be parameter sequences as in Zest or, alternatively, inputs to the PUT for fuzzing approaches that don't use generators. The other two methods in the guidance interface are used to receive execution data.
The fuzzing loop of Zest is a modified version of algorithm 1. Instead of interesting inputs, Zest's guidance maintains a queue of *interesting parameter sequences*. Therefore, the set in line 1 of algorithm would be a set of parameter sequences. As Zest collects coverage of valid inputs alongside all coverage, there is another set storing valid coverage like in algorithm 1 line 2. Zest's seed selection (i.e. line 4) iterates over the set of interesting parameter sequences beginning with the first sequence added to the set. Power scheduling (i.e. line 5) is completely random in Zest, meaning that a parameter sequence will receive a random number of mutations. More concretely, energy is sampled from a geometric distribution with a mean of four but no upper bound. A single mutation to the parameter sequence (i.e. line 6 in algorithm 1) is carried out by choosing a random length l from the same geometric distribution as used for the energy, choosing an offset k in the parameter sequence, and replacing l continuous bytes starting at the byte at the offset k with random bytes. This mutational step is applied multiple times according to the energy. Next, in line 7 of algorithm 1 the PUT is executed with an input. However, Zest would first pass the mutated parameter sequence to a parametric generator to obtain an input. This input would then be executed to collect coverage data and the execution result. Failure handling (i.e. lines 8 and 9 in algorithm 1) is performed by saving the parameter sequence that produced the failure. Coverage handling (i.e. lines 10-12) is also extended by Zest. Concretely, Zest collects coverage archived by valid inputs alongside coverage of all inputs. In addition to lines 10-12, Zest would also add the coverage archived by a valid input to the set of valid coverage. Moreover, if an input increased the valid coverage, its parameter sequence is added to the set of interesting parameter sequences. We can summarize Zest's guidance as a procedure that continuously selects a parameter sequence that previously increased coverage or valid coverage, mutates the sequence by applying random byte-level mutations, passing the parameter sequence to a parametric generator to obtain an input, passing the input to a test runner that emits coverage data

and execution results, and updating coverage, valid coverage and the set of interesting parameter sequences using coverage and execution results.

Seeds in Zest JQF allows testers to initialize fuzzing campaigns with seeds files. Each seed file contains a single initial input to the fuzzing loop inputs. Zest also supports the initialization with seed files. Specifically, Zest considers every seed file as a sequence of bytes that is uses as a parameter sequence. Thus, a tester can't give Zest an input to the PUT as a seed, but have to construct a parameter sequence representing the input. This is a mayor limitation of Zest, as it restricts the availability of seeds. Moreover, Zest neither provides a tool to generate parameter sequences for a given input nor comes with a procedure to generate seeds quickly. Zest is usually run without seeds, and thus uses the first minutes of its fuzzing campaign to create meaningful inputs that can execute deep program logic. We try to improve this part of Zest by presenting a technique that quickly and automatically generates parameter sequences that produce diverse inputs and handing them to Zest.

Improvements to Zest In this section, we present approaches that elaborate on the ideas proposed by Zest and also extend the JQF framework. These improvements are orthogonal to our work.
In [42] Nguyen et al. classify choices inside Zest's generators into those that are responsible for values for input variables (i.e. *value choices*) and those that are responsible for the generation of different structures (i.e. *structure choices*). Furthermore, they identify parts of parameter sequences that flow into structure choices or value choices. Using this information, they design structure-aware and structure-preserving mutations that can be systematically applied to equally traverse the input space. They also record parameter sequences responsible for new structures to enable the quick discovery of diverse structures.
Another improvement to Zest is Confetti [32] that integrates symbolic execution into Zest. Concretely, Confetti adds an independent thread to Zest that runs a symbolic executor. However, the goal is not to replace Zest's fuzzing loop, but instead to give Zest *hints* at runtime. Confetti hands interesting parameter sequences, saved by Zest, to the symbolic executor that uses taint tracking to obtain *targeted hints*. Targeted hints point at concrete bytes in the parameter sequence that Zest should change to a specific value to uncover new branches. Moreover, targeted hints can also be saved as *global hints* that Zest can insert into a parameter sequence instead of applying a targeted hint or normal mutation.

3.4 RLCheck

Another fuzzer extending the JQF [44] framework is RLCheck [53]. RLCheck is a blackbox fuzzer utilizing reinforcement learning to quickly generate diverse and syntactically valid inputs. As its blackbox classification suggests, RLCheck is not relying on execution insight like coverage data. However, is uses information from inside its generators alongside the results of the execution to drive the generation. Specifically, RLCheck makes use of the validity feedback provided by JQF to learn what makes up valid inputs.

Learning Choices for Generators In RLCheck, generators have two mayor concepts, *states and choices*. Each generator traverses trough different states by taking choices. Concretely, the generators enters a new state after every choice. RLCheck tries to base these choices on previous choices in the same state. This idea is realized with *reinforcement learning* and a *state abstraction function*.
States traversed by a generator are captured in a state sequence. Generators for complex file formats tend to traverse hundreds or thousands of states. Therefore, to efficiently process these state sequences, RLCheck makes use of a state abstraction function to compress the state sequence to a fixed size. Additionally, state sequences are combined with their respective choices at each state to form *choice sequences*.
Reinforcement learning is used to learn a policy for supplying the generators with choices. This process is based on *Monte Carlo Control* [59]. Whenever an input is executed, the execution result, input values and choice sequences used for generation are recorded. RLCheck assigns a fitness value to each state-choice pair in the choice sequence. High fitness values are assigned to choice sequences that generate syntactically valid inputs that also contain values never seen before (i.e. *diverse values*). Furthermore, when a generator has to make a choice, it prefers a choice that has previously archived high fitness values. This procedure tends to heavily exploit validity feedback, restricting RLCheck to a set of choices that created valid inputs before. Thus, exploration is added by occasional random generator choices to increase diversity of generated inputs.

RLCheck's Implementation in JQF RLCheck implements new generators that contain additional functionality to capture and exploit choice sequences. The generation process is mostly the same as in Zest [45]. However, when a generator in Zest would request bytes from its parameter sequence, RLCheck's generator request a decision from a *guide*. Guides represent the interface between generators and the reinforcement learners. A generator creates multiple learner instances through the guide. Moreover, each type of choice has its own learner (e.g. RLCheck has separate learners for choosing integers and for choosing characters). Generators also forward fitness values, for the learners, through the guide. RLCheck implements state abstraction functions by simply trimming choice sequences to a fixed size of the last choices.
RLCheck also implements a new guidance. Contrary to Zest, the generators are directly invoked inside the guidance and no mutation or seed selection is performed. That is because Zest operates on parameter sequences that are handed to generators, while RLCheck only reuses execution results (invalid, success or failure). Therefore, the guidance forwards execution results to the generators and their learners to reward valid inputs with diverse values. This process is repeated until the user-defined timeout is reached.
JQF's test execution is untouched by RLCheck and the validity feedback created by the test runners is vital to RLCheck. However, RLCheck implements own property-based

tests that are not annotated with generators as in Zest. More specifically, the novel unit tests directly get inputs from the guidance, which itself gets the inputs from the generators.

3.5 Initial Seeds

In 2014 Rebert et al. [52] suggested the importance of an initial seeds for covering large parts of the program under test (PUT) by evaluating their novel family of *Minset algorithms* for selecting an initial seed corpus from a large set of inputs. More recently, Herrera et al. surveyed current research in the field of mutation-based greybox fuzzing in regard to their use and discussion of initial seeds ([26]). They found that the majority of fuzzing research either excludes information about their seed corpus, use a single seed, or use no seeds at all. Moreover, they discovered that many of these research papers don't discuss the impact or motivation behind the choice of seeds. In addition, Herrera et al. also confirmed the advantage of using a larger initial set of seeds for finding bugs and for increasing coverage ([25], [26]). Furthermore, their results suggest that a *minimized* set of seeds is optimal.

Seed Corpora Minimization The current state of the art for the selection of seed corpora is *seed corpora minimization* (e.g. [25], [26], [52]).The idea behind seed corpora minimization is to select a minimal subset of a large set of inputs that maximizes the behaviour executed in the PUT. The large set of inputs is usually crawled from the internet, and we will call the set *initial set* in this section.
Prominently, afl-cmin is a tool included in the highly successful mutation-based coverage-guided greybox fuzzer AFL [67]. The tool takes an initial set, executes the PUT with each input inside the set, and greedily selects those inputs that increase coverage. The selected inputs form the seed corpora that is used as a starting point for fuzzing in AFL. In [52] Rebert et al. formalize seed corpora minimization as an instance of the *minimal set cover problem* (MSCP). The MSCP gets a set X and a finite list $F = \{S_1, \ldots, S_n\}$ of subsets of X such that every object in X is a member of at leats one $S_i \in F$. The goal of MSCP is to find a minimal $C \subseteq F$ such that every object in X is a member of at least one $S_i \in C$. The minimal set C is called *minset*. Rebert et al. also explain the use of the weighted version of the MSCP (WMSCP) which additionally assigns a weight to each $S_i \in F$ and tries to minimize these weights instead of the size of C. Both the MSCP and WMSCP are known to be NP-hard. Therefore, Rebert et al. leverage greedy polynomial-time approximation algorithms to solve both problems. In order to incorporate (W)MSCP with seed corpora minimization, they extract the set of coverage archived by each input in the initial set. Moreover, they use the union of all individual coverage sets (i.e. the coverage archived by a single input from the initial set) as the superset X in (W)MSCP and all individual coverage sets as the finite list of subsets. Afterwards, they apply an approximation algorithm to solve the (W)MSCP. They call their MSCP and WMSCP algorithms *unweighted Minset* and *weighted Minset*. For weighted Minset they use execution time of the input and file size of the input as weights,

resulting in *time Minset* and *size Minset* respectively. Their experiments suggest that the unweighted Minset algorithm performs best. Moreover, they find that seed corpora minimization is important to cover large parts of the PUT, by showing an advantage of a minimized seed corpus over using all inputs in the initial set.
Herrera et al.[25] has elaborated on the idea of formalizing seed corpora minimization as an instance of the (weighted) minimal set cover problem ((W)MSCP) to minimize a seed corpus (as in [52]). Their new approach is *MoonLight* that builds a matrix from the initial set. Inside this matrix, each row is an input whereas each column represent a single coverage measurement. In other words, the value in row i and column j of the matrix represents if the i-th input emitted the j-th coverage measure when executed with the PUT. MoonLight solves the MSCP by applying dynamic programming with a set of rules to continuously remove rows and columns from the matrix and select inputs as seeds in the process. They design a set separate set of rules for the WMSCP. This approach is still producing no optimal solution to the (W)MSCP, as it applies heuristic rules when no other rule can be applied.
More recently, Herrera et al. have proposed *OPTIMIN* [26] that can generate an optimal solution to the seed corpora minimization problem. OPTIMIN archives this optimal solution by representing seed corpora minimization as an instance of the maximum satisfiability problem (MAXSAT). MAXSAT gets a list of boolean formulas divided into *soft constraints* and *hard constraints*. The task in MAXSAT is to satisfy all hard constraints and to maximize the total number (or weighted total) of satisfied soft constraints. OPTIMIN formalizes each coverage measurement as a hard constraint. Additionally, for each input in the initial set, the removal of that input from the seed corpus is formalized as a soft constraint. In other words, to solve this instance of MAXSAT each coverage measurement has to be in the minimized seed corpora (i.e. satisfy all hard constraint) and a maximum number of input should be dropped from the seed corpus (i.e. satisfy the maximum amount of soft constraints). To solve the MAXSAT, Herrera et al. leverage an existing MAXSAT-solver.
Seed corpora minimization requires a set of inputs to be available. However, a set of inputs might not be available due to undocumented input file formats. This issue was reported in [25] for some of their benchmarks. Additionally, collecting the initial set (e.g. by crawling inputs from the internet) requires manual effort and might not be feasible because of limited time or other restrictions. Another limitation of seed corpora minimization is that it requires execution of the instrumented PUT to obtain coverage data. Collecting coverage for every input in a large initial set is time-consuming, wasting resources that could be used for fuzzing instead. Our approach tries to alleviate those limitations by not relying on the availability of an initial set of inputs. Moreover, we generate a large initial set using a quick blackbox fuzzer. Instead of collecting coverage to minimize the seed corpus, we exploit data collected during the generation of inputs to filter them. Additionally, we free the user from any manual involvement by automating the entire process.

Other Approaches Skyfire [61] tries to generate a diverse, rare and valid seed corpora. Moreover, the approach takes as inputs a context-free grammar and an initial seed corpus. Next, Skyfire combines the initial corpus with the grammar to create a probabilistic context-sensitive grammar. This novel grammar consists of rules enriched with contexts and probabilities according to their usage in the given corpus. Skyfire then exploits this grammar to heuristically generate rare and size-restricted inputs. Those inputs are then executed with the PUT and coverage is collected. Next, coverage information is used to filter coverage-redundant inputs. In a last step, Skyfire mutates the filtered inputs by exchanging entire subtrees in their abstract syntax trees with subtrees derived from the same non-terminal symbols. Skyfire is restricted by the availability of initial inputs and a production grammar. Contrary, our new approach does not require any inputs or grammars. Skyfire *generates* new inputs, which is a shared property with our approach. However, we leverage generators to create inputs, while Skyfire uses a grammar. Contrary to our approach, Skyfire requires the execution of inputs to obtain coverage data.
Similar to Skyfire, MoonShine [47] requires additional data for seed generation. MoonShine is a greybox fuzzer targeting operating system interfaces. Inputs to such interfaces are sequences of system calls. The challenge in fuzzing interfaces for operating systems is that system calls have dependencies that have to be fulfilled in order for the system calls to work (e.g. a read-call requires the previous opening of an input file descriptor in read mode). Moonshine tries to generate seeds that already satisfy these dependencies by collecting a large set of real-world system call traces, greedily choosing system calls that increase coverage inside the operating system interface, and using lightweight analysis to determine dependencies using the extracted system call traces. Our approach is hardly comparable to MoonShine as we target Java PUT's while MoonShine targets interfaces of operating systems. Similar to the other approaches, MoonShine relies on the collection of initial data for seed corpora creation. Contrary, we avoid collecting initial data and rely on generators to create inputs.
Another approach for dealing with initial seeds is seedless fuzzing (SLF) [65]. The motivation behind SLF is the potential unavailability of initial seeds. SLF tries to generate seeds alongside regular coverage-guided mutation-based fuzzing (i.e. they use AFL [67]). It starts with an input consisting of four random bytes and fuzzes regularly until a validity check in the PUT fails. When a validity checks fails, SLF tries pre-defined mutation- and search strategies to identify which bytes of the input influence the validity check. These bytes are grouped into a *field*. Next, SLF applies other mutation- and search strategies to classify the failed validity check. Equipped with the knowledge about the type of the validity check and the fields influencing it, SLF determines *interdepended validity checks*. Interdependent validity checks are previously resolved checks that depend on the same fields as the current check. Lastly, SLF applies a multi-goal search algorithm to satisfy all interdependent validity checks and create a new valid seed. Contrary to our approach, SLF works concurrent with the fuzzing campaign. SLF tries to ensure validity of seeds by solving validity checks. However, this approach is limited by the classification of validity checks. In particular, the authors note that SLF is currently unable to deal with one type of validity check. When detecting this particular validity check, SLF would return to normal fuzzing. Contrary, our technique leverages the power of generators to

create valid inputs with RLCheck [53]. Moreover, RLCheck's reinforcement learning is tuned to prioritize valid inputs. However, our approach shares the motivation of not relying on the availability of initial inputs with SLF.

Motivation of our new Approach Previous research in regard to seed corpora has focussed on coverage-guided mutation-based fuzzing that requires seeds to be inputs to the PUT. For Zest [45], an approach that requires seeds to be parameter sequences, this previous research is not applicable. However, Zest is a promising technique that also incorporates coverage-guided mutation-based ideas. Therefore, the results of [52], [25] and [26] that a minimized seed corpus is optimal for uncovering bugs and increasing coverage in the PUT should also apply to Zest. To apply these insights to Zest, we design a novel seed corpora generation approach that is able to quickly supply Zest with a set of seeds.
Except for SLF [65], all previous approaches for seed corpora generation require the availability of an initial set of inputs. We argue that such a set might not be available, or its acquisition might be infeasible due to resource restrictions or undocumented file types. To alleviate those shortcomings, we leverage the blackbox fuzzer RLCheck top quickly generate a large set of inputs. Additionally, due to RLCheck's focus on validity and we obtain a set of mostly valid inputs.
Reducing the size of an initial set of inputs is a common task in previous approaches. Furthermore, previous research tries to reduce the size of the initial set by executing each input, collecting coverage and selecting a minimal set of inputs that maximizes the coverage in the PUT. We argue that execution of each input is a time-consuming task for large initial sets. Therefore, we propose an approach that filters the set of inputs, obtained by RLCheck, by exploiting different information collected during the generation of these inputs. It is important to note that we also execute the PUT during the generation of inputs with RLCheck. However, RLCheck does not capture coverage during execution, which enables RLCheck to execute orders of magnitude more input than approaches executing an instrumented PUT (to capture coverage).

3.6 Adaptive Random Testing

Adaptive random testing (ART) is an extension to classical random testing (RT) that tries to randomly select meaningful test cases to a PUT from the entire input space. Contrary to RT, ART tries to distribute the selected test cases so that the entire input space is equally covered. There are many ART approaches that leverage different definitions of similarity or distance between test cases. For more details, we refer to the recent survey [29] on ART.
Our approach aims to filter seeds so that they trigger diverse behaviour in the PUT, which represents a similar task to distributing test cases like ART tries to do. A key difference is that we filter a preselected set of inputs (i.e. the inputs RLCheck generated) instead of selecting them from the entire input space. Moreover, we don't directly filter inputs, but rather use other information for filtering (more details in section 4.2). Another difference is that we don't want to execute the PUT for filtering, while many ART approaches

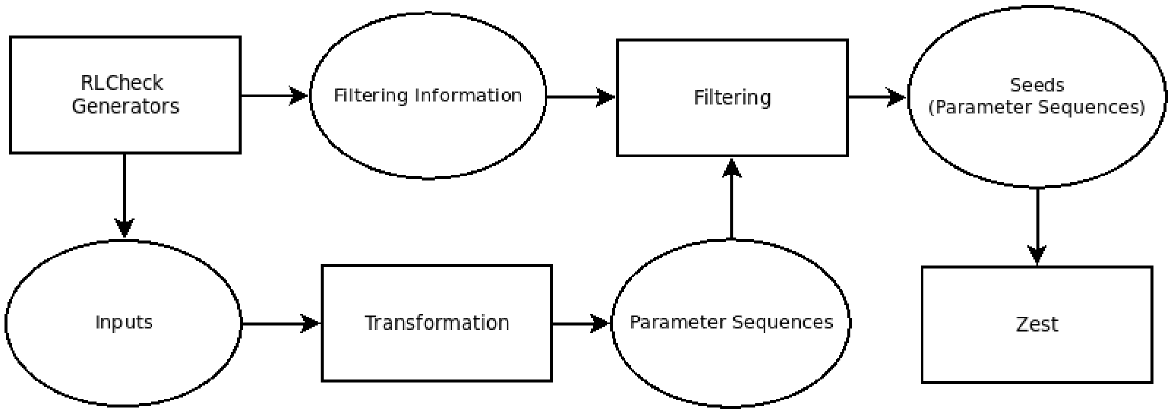


Figure 1: Conceptual overview of our approach.

involve executions. Although there are many differences between ART and our approach, both share the goal to select dissimilar inputs from a larger set. Therefore, for two of our four filtering techniques, we borrow the *select-test-from-candidates strategy* from ART. Select-test-from-candidates strategy is a simple ART technique that relies on the definition of a *similarity measure* and does not require the execution of the PUT. The approach continuously chooses a random set of inputs from the input space (in our case the set of inputs obtained from RLCheck) called *candidate set*. Next, a candidate is chosen from the candidate set that minimizes the similarity measure. In other words, the chosen input has a minimal similarity to all previously chosen inputs. We describe our usage of these techniques and the selected similarity measures in section 4.2.

4 Our Approach

Figure 1 depicts an overview of our approach to seed corpora generation. We first run RLCheck [53] and its generators, obtaining inputs and additional filtering information. Afterwards, we transform inputs into parameter sequences and filter them, potentially using additional filtering information. When the filtering process is finished, parameter sequences are written to seed files that are used by Zest [45] to start its fuzzing campaign. Our approach consists of two mayor parts that we will describe in this section. First, the *transformation* is where we construct parameter sequences usable as seeds for Zest. More specifically, we transform inputs generated in RLCheck into parameter sequences that would produce the same input when given to a parametric generator in Zest. The second part is the *filtering* of inputs generated by RLCheck to reduce the number of seeds handed to Zest. Additionally, we will show how we integrated those two parts into the existing RLCheck implementation. In the following three sections we present details on the transformation and our new generators (section 4.1), filtering techniques (section 4.2) and the integration of both parts in the existing implementation of RLCheck (section 4.3).

4.1 Transformation

Transformation is the part where we construct parameter sequences usable as seeds for Zest [45], which is crucial to our approach. In figure 1 we indicate that transformation is carried out after RLCheck is finished with generating inputs. However, we want to construct these parameter sequences alongside the generation of inputs in RLCheck [53]. We argue that constructing parameter sequences alongside the inputs saves computational resources. Concretely, at a given state of a generator, all possible information about the recently generated part of the input are available. Constructing the parameter sequence after the generation of the input is finished would essentially restore all of this information again and thus waste resources in comparison to constructing the sequences alongside the generation process. Moreover, RLCheck generators are similar to their counterparts in Zest, which also eases on the fly construction of parameter sequences.

Interface for new Generators To allow construction of parameter sequences and collection of filtering data, we extended RLCheck's generators. Furthermore, we created a novel interface for our new generators that extend the interface for RLCheck generators. The interface mainly supplies functionality for transmitting information to and from the generators. On the incoming side, generators get instructions on which information they should store for filtering. Parameter sequences are always saved. In the outgoing direction, generators supply parameter sequences and additional filtering information through the interface.

Generators Implementing the generators can be described as extending RLCheck's generators to emit parameter sequences that enforce a similar control flow through Zest's generators. Although RLCheck's generators are similar to Zest's, this work was not trivial and posed many challenges that we will discuss in this section. First, recall that whenever a generator in Zest has to make a decision, it reads bytes from a parameter sequence to decide on the next generation step. Thus, for every input generated in a RLCheck generator, we need to construct a parameter sequence that guides Zest's generator to create the same input. Our approach to this problem is to find each read from a parameter sequence in Zest's generators and identify the context of the read (i.e. what is generated using the obtained bytes). Next, we look at RLCheck's generator for the same file type and find the part of the generator that has the same context (i.e. the part that generates the same part of an input). Then, we determine how many bytes Zest's generator reads and how they are used. Finally, using the information about context and usage of bytes, we extend RLCheck's generator to construct a parameter sequence that enforces the same generational behaviour in Zest's generator than the behaviour in RLCheck's generator. To illustrate how we realized this, we will discuss an example.

Figure 2 shows a function inside Zest's JavaScript generator that is responsible for generating identifiers for functions and variables. The function gets a SourceOfRandomness object as a parameter (line 1). In particular, that SourceOfRandomness is an object that *reads bytes from a parameter sequence.* The generator keeps track of already generated

```
 1 private String generateIdentNode(SourceOfRandomness random) {
 2         String identifier;
 3         if (identifiers.isEmpty() || (identifiers.size() < MAX_IDENTIFIERS &&
         ↪  random.nextBoolean())) {
 4             identifier = random.nextChar('a', (char) 123) + "_" +
             ↪  identifiers.size();
 5         } else {
 6             identifier = random.choose(identifiers);
 7         }
 8
 9         return identifier;
10     }
```

Figure 2: Function to generate a JavaScript identifier in Zest's generator.

```
 1 protected String generateIdentNode(String[] stateArr) {
 2         stateArr = updateState(stateArr, "node=ident");
 3         if(identifiers.isEmpty()){
 4             return generateIdentifier(stateArr);
 5         } else if (identifiers.size() < MAX_IDENTIFIERS) {
 6             if ((Boolean) guide.select(stateArr, boolId)){
 7                 ParameterSequenceUtils.appendBoolean(true,parameterSequence);
 8                 return generateIdentifier(stateArr);
 9             } else {
10                 ParameterSequenceUtils.appendBoolean(false, parameterSequence);
11                 return chooseIdentifier(stateArr);
12             }
13         } else {
14             return chooseIdentifier(stateArr);
15         }
16     }
```

Figure 3: Function to generate a JavaScript identifier in our new generator.

identifiers using the "identifiers" container that we will assume to be a list for simplicity (in reality it is a linked hash set). A new identifier is generated if no identifier has been generated so far (lines 3-4). Additionally, if the number of identifiers already generated is lower than the maximum number of allowed identifiers and a byte representing "true" is read from the parameter sequence, a new identifier is created (lines 3-4). If none of these conditions are true, an existing identifier is selected (lines 5-6). Boolean values are determined using a single byte read from the parameter sequence (line 3). When an existing identifier is chosen (line 6), four bytes representing an integer are read from the parameter sequence to determine an index. Moreover, when a new identifier is generated (line 4), four bytes are read from the parameter sequence to generate a single character (the identifier is made unique through the concatenation of the current size of "identifiers"). Overall, there are three occasions where the generator could read bytes from the parameter sequence (lines 3,4,6). Concertizing our previous explanations, we can derive the following four distinct behaviours and respective reads from the parameter sequence in the function in figure 2:

case 1: "identifiers" is empty → four bytes read from the parameter sequence in line 4 for a character case 2: "identifiers" has more entries than the maximum allowed number of entries → four bytes read from the parameter sequence in line 6 for an index (the read for a boolean value in line 3 does not happen because of the evaluation order of logical expressions in Java)

case 3: "identifiers" has less or equally many entries then the maximum allowed number of entries and a read from the parameter sequence represents the value "true" → one byte read in line 3 for a boolean value and four bytes read in line 4 for a character

case 4: "identifiers" has less or equally many entries then the maximum allowed number of entries and a read from the parameter sequence represents the value "false" → one byte read in line 3 for a boolean value and four bytes read in line 6 for an index

Remember that our goal is to enforce the generation of a specific input. To archive this, we need to control which bytes are read at a given state in the generator. Constructing bytes that represent a boolean value (for line 3) is archived by either creating a byte representing the value 0 for "false" or by creating any other byte for "true". Indices (for line 6) are generated by reading four bytes from the parameter sequence, interpreting them as an integer and possibly bringing the integer in the correct range by applying a modulo operation with the current size of "identifiers". Therefore, bytes for an index can be constructed by simply using the bytes that make up the index. Characters (for line 4) are a bit different. Zest applies JQF's FastSourceOfRandomness Object that aims to randomly generate certain objects fast but with less statistical guarantees than classical random number generators. In particular, an integer is read from the parameter sequence and some quick calculations are performed with it to create the character. To construct the four bytes to generate a particular character, we had to reverse those calculations to derive the integer that can be read from the parameter sequence. Constructing the bytes for boolean values, characters and indices is rather easy. The real challenge is to determine which bytes need to be present in a parameter sequence to strictly enforce a particular behaviour in the generator. Specifically, the required bytes are dependent on the state of the generator. For example, the read of a boolean value on line 4 depends on the size of "identifiers" (i.e. the read only happens if "identifiers" is not empty and has fewer entries than the maximal allowed number of entries).
Figure 3 shows the function in our new JavaScript generator that is also responsible for generating identifiers. Moreover, the function is constructing parameter sequences on the fly to enforce equivalent generational behaviour in the function in figure 2. First, the function gets an array of strings as an input and appends the string that signals the generation of an identifier (figure 3 line 1-2). This array is the current choice sequence of RLCheck (see section 3 for details) and the processing and updating of choice sequences is left untouched by our approach. As in RLCheck, our generators take choices based on previously learned choice sequences (i.e. RLCheck's reinforcement learning). Therefore, our generator and has to update and propagate choice sequences through the generation process to allow RLCheck to learn its choice policies. In lines 8,11 and 14 in figure 3 functions for either generating or choosing an identifier are called. These functions append the bytes for indices or characters to the parameter sequence as we described

above, they also update the current choice sequence. We omit the implementation of these functions. Instead, we focus on how we construct parameter sequences to reflect the behaviour required for the cases 1 to 4.
Case 1 is handled in figure 3 lines 3-4 where "identifiers" is checked for emptiness. If that's the case, only four bytes representing a character are appended to the parameter sequence. We should mention that it is crucial that no more bytes are appended. Moreover, if "identifiers" is empty and the function in figure 2 is called, then only four bytes are read (i.e. for a character). If we would still add more bytes to the parameter sequence it would be read in a different component of the generator causing a deviation from the desired flow through the generator.
Cases 3 and 4 are both handled in lines 5-12. First, it is checked if the number of already generated identifiers is less than the limit (line 5). Only if this is true, line 6 is entered. This step is crucial since the function in figure 2 only reads a byte for a boolean value on line 3 if "identifiers" has fewer entries than the limit. Furthermore, in line 6 in figure 3 RLCheck's guide is asked to emit a boolean value which corresponds to the read from a parameter sequence in line 3 in figure 2. The guide's answer is then appended to the parameter sequence (lines 7,10 figure 3 and an identifier is either generated or chosen (lines 8,11). Specifically, in both cases, the corresponding bytes for either a character or an index are appended to the parameter sequence.
Finally, case 2 is handled in lines 13 and 14. In this case, only bytes for an index are appended to the parameter sequence. Notably, we don't append a byte for a boolean value for the if condition in figure 2. That is, because case 2 implicates that identifiers has more entries than the limit. Therefore, the conjunction is falsified by its first component and the second component (i.e. read from the parameter sequence for a boolean value) is never evaluated.

Correctness of Parameter Sequences The core challenge of the example we presented above is to analyse the nested if-else conditions to derive a counterpart in our new generator that can reliability append the correct bytes to the parameter sequence. Analysing nested conditional statements that incorporate reads from parameter sequences in Zest's generators proved to be an error-prone and tedious task. Furthermore, we want to emphasize again that any incorrectly appended byte in a parameter sequence can lead to a completely different behaviour in Zest's generators. To undermine this, we present another example. Consider the functions in figures 2 and 3 and suppose a generator state where "identifiers" has reached its limit. Thus, the first component of the conjunction in figure 2 line 3 would be false. According to our implementation in figure 3 we would only append one index to the parameter sequence in line 14. Let's say the chosen index is 42 resulting in the parameter sequence $(\ldots, 42, \ldots)$. To clarify, for readability we present the integer value of the index inside the sequence, in reality it would be four bytes representing an integer. Additionally, we indicate that the parameter sequence may have more bytes added by other functions (i.e. with "$\ldots$"). Now consider a small change to our implementation in figure 3 regarding case 2. Precisely, we assume that we ignore the evaluation order of logical expressions and thus add a boolean value to the parameter sequence even if "identifiers" has reached its limit. Let's also assume that we would add "false" and that the addition to the parameter sequence would happen before line 14 in figure 3. According to those assumptions, we would get the parameter sequence $(\ldots, \text{false}, 42, \ldots)$. We should also mention that in reality, a boolean value is represented by a single byte in parameter sequences. Now let's have a look how this parameter sequence would be used in the function in figure 2 for the generation of one identifier. First, since "identifiers" has maximal entries, the conditional statement in line 3 would be false, leading us to line 5 without a read from the parameter sequence. Afterwards, in line 6 four bytes would be read from the parameter sequence to choose an index. Concretely, the single byte representing "false" would be read alongside three bytes from the index of 42, leaving one byte of this index unread. This will lead to an index that is different from 42 and might dramatically change the semantic of the resulting JavaScript code. Even more problematic is that there is on byte left in the sequence that will be used somewhere else in Zest's generator. This byte was only supposed to be used in generation of identifiers. Instead, it could now be incorrectly read to decide on the structure of the following JavaScript code. Additionally, this single incorrectly appended byte shifts every following byte in the parameter sequence by one. Thus, this shift affects the generation of every following part in the generation of the JavaScript code. Moreover, it is highly likely that multiple identifiers are chosen following case 2 and thus many more bytes are incorrectly injected in the parameter sequence. Judging from our experience, even one incorrectly appended byte leads to a completely different input generated in Zest's generator to the actual result of our generator.

Adapting Generators Looking at the two example above, it becomes obvious that in order to create the parameter sequences, we need our new generator to be similar to the generator used by Zest. We extended the generators of RLCheck as describe in our first example. Although the RLCheck generators are clearly derivatives from Zest's generators, we still had to assign a lot of effort to bring both generators in line with each other. A simple example for the adaptation is the "MAX_INDETIFIERS" value in figures 3 and 2. This value has to be the same in our new generator as in Zest's generator to correctly construct parameter sequences. An observation we made during implementation is that generators themselves had logical errors. Those errors were sometimes shared between both generators, while sometimes being exclusive to one generator. An example for this is line 4 in figure 2 where a character is generated. In this example, we had to change the right border to "(char) 123)" that was previously just "z". We had to change that because the right border of the "nextChar" function is exclusive, meaning that the generator was not able to generate the character "z". On the other hand, RLCheck's generator was able to generate the character "z". Without fixing the function in figure 2, it would have generated the character "a" whenever RLCheck generated "z". Such divergent identifiers can lead to semantically different JavaScript code, especially considering that the method can generate identifiers for functions. We made the observation that ranges of allowed values were particularly error-prone in both generators. In particular, upper value bounds imposed the most problems due to their (non-)exclusiveness.

We already discussed how bytes for boolean values, integers and characters can be constructed, which represent almost all possible data types constructed using bytes from parameter sequences. The last type are strings (i.e. sequences of characters) that proved to require additional adaptation of the generators. Specifically, Zest's generators created strings by first sampling the length of the string from a geometric distribution and afterward generating individual characters. The problem with this approach is the length of the string being generated by applying floating point number arithmetic to four bytes read from the parameter sequence. This process is subject to a loss of precision, which makes it impossible to accurately generate a parameter sequence to enforce a specific string length. This is a great problem for our approach, since an imprecise length implies an imprecise number of reads from the parameter sequence to generate the individual characters. We already described the problems with appending an incorrect number of bytes in our second example. According to these observations, we simplified the generation of Strings in Zest to read the length of a string from four bytes, representing an integer. Moreover, we also modified the way characters are generated for those strings. Each character in a string is generated by reading a single byte from the parameter sequence and directly interpreting it as a character. Furthermore, when constructing a parameter sequences for string generation, it will contain four bytes containing the length of the string followed by a number of bytes each representing a character. We should mention that we did not change the process of generating character for identifier as in figure 2 line 4.

4.2 Filtering Techniques

Now that we have described how we generate parameter sequence, we will describe the filtering process. To motivate this part, we can take a look at the numbers of inputs generated by RLCheck [53] and Zest [45]. As an example, in our evaluation of the Rhino benchmark our RLCheck-based generators generates around 70000 inputs in one minute while Zest generates around 700 inputs in the same amount of time. Additionally, in one hour of fuzzing Zest can generate around 150000 inputs. So if we were to hand every parameter sequence created in our generator to Zest for a campaign that runs for one hour, it would spent roughly half of its resources on fuzzing the seeds. This would hinder Zest's ability to discover deep program states. Moreover, the authors of RLCheck described that their approach is too exploitative of valid inputs. Thus, our RLCheck-based generators will generate many similar inputs. To alleviate those problems, we want to *minimize* the set of inputs and their respective parameter sequences produced by our generators. Currently used techniques like afl-cmin [67] would execute the PUT with all seeds to determine which seeds can be dropped from the seed corpora. This process is time-consuming and ignores information that we can easily extract from the generation process. Specifically, our generators already emit parameter sequences that have embedded information about the generation process. Additionally, we can extract the choice sequences that RLCheck uses to learn its choice policies. Another idea that we want to apply is that we can instrument our generator instead of the PUT to reason about coverage gains inside the generator. Using this data allows us to define *quick* *filtering techniques* that don't need to execute the PUT with the seeds. In short, our filtering techniques get all parameter sequences created inside our new generators, filter them using different sources of information and write a set of parameter sequence to seed files. In this section, we will describe how we extract and exploit different sources of data inside the generation process of inputs. Specifically, we describe how we filter seed corpora, resulting in a small set of seeds. Three out of four filtering techniques require the specification of the amount of seeds that the filtering technique should emit, we will call this parameter *output size* from now on.

Random Filtering Random filtering will be the baseline for our filtering techniques. It randomly selects parameter sequences until the output size is reached. Although this approach is simple, it still may be able to deliver solid result. That is, the unfiltered seeds corpora will contain many similar inputs. If we consider similar inputs to form clusters, random filtering with the right output size might be able to select only a few inputs each a cluster. Furthermore, this technique does not require any expensive computations, which might give it a time advantage over the other approaches.

Parameter Sequence Filtering Parameter sequences are always emitted by our generators because they are required as seeds for Zest [44]. Additionally, parameter sequences contain information about the flow through the generator that created them. Parameter sequence filtering tries to leverage this information to determine whether two inputs are similar. So instead of comparing inputs directly, we will compare parameter sequences created alongside the inputs.

Algorithm 2 shows pseudocode for parameter sequence filtering. The technique receives a set of parameter sequences as its input. First, it chooses a first parameter sequence randomly and saves it to the output set (lines 1-3). Next, until the desired output size is reached a candidate set is randomly selected from all given parameter sequences, a candidate that has minimal similarity to the currently selected parameter sequences is chosen and added to the output set (lines 4-7). This procedure is an implementation of the select-test-from-candidates strategy explained in section 3.6.

The computation of similarity is show in the procedure starting at line 11. It takes a single candidate and the set of already selected parameter sequences. Intuitively, we compute similarity as a form of prefix matching in parameter sequences. The key idea is that an equal prefix in a parameter sequence represents an equal flow through a generator. Thus, a long matching prefix implies that both parameter sequences were generated similarly. When computing similarity, we loop over all inputs in the set of already selected parameter sequences to compute the similarity with the candidate (line 13). The similarity between the candidate and single parameter sequence is initialized to zero and incremented for each matching byte (lines 12-20). A different byte implies a divergence in the control flow through the generators. But this also has exceptions. Concretely, similar to the idea of value choices in [42], a number of diverging bytes in the parameter sequence may be used to make a choice that does not influence the structure of an input. For example, two diverging sequences of four bytes used to choose an integer that is

Algorithm 2 Parameter Sequence Filtering

```
    Input: $\mathcal{S}$                    ▷ $\mathcal{S}$ initial unfiltered set of parameter sequences
    Output: $\mathcal{O}$                   ▷ $\mathcal{O}$ filtered set of parameter sequences
 1: $\mathcal{O} \leftarrow \emptyset$
 2: choose $c_1$ randomly from $\mathcal{S}$
 3: $\mathcal{O} \leftarrow \mathcal{O} \cup \{c_1\}$
 4: while $|\mathcal{O}| < output\_size$ do
 5:     randomly choose candidate set $\mathcal{C}$ from $\mathcal{S}$
 6:     select $c_{min} = \arg\min_{c \in \mathcal{C}} computeParaSeqSimilarity(c, \mathcal{O})$
 7:     $\mathcal{O} \leftarrow \mathcal{O} \cup \{c_{min}\}$
 8: end while
 9: return $\mathcal{O}$
10:
11: procedure $computeParaSeqSimilarity(c, \mathcal{O})$
12:     totalSimilarity $\leftarrow 0$
13:     for $p \in \mathcal{O}$ do
14:         similarity $\leftarrow 0$
15:         minLength $\leftarrow \min(|p|, |c|)$
16:         $i \leftarrow 0$
17:         while i < minLength do
18:             if $p[i] == c[i]$ then
19:                 similarity $\leftarrow$ similarity + 1
20:                 $i \leftarrow i + 1$
21:             else if $p[i + 1] == c[i + 1]$ then
22:                 similarity $\leftarrow$ similarity + 0.5
23:                 $i \leftarrow i + 2$
24:             else if $p[i + 4] == c[i + 4]$ then
25:                 similarity $\leftarrow$ similarity + 2
26:                 $i \leftarrow i + 5$
27:             else
28:                 break
29:             end if
30:         end while
31:         totalSimilarity $\leftarrow$ totalSimilarity + $\frac{similarity}{|c|}$
32:     end for
33:     return totalSimilarity
34: end procedure
```

assigned to an identifier will probably not influence the semantics of the resulting inputs. Inspired by this observation, we extended the prefix matching to allow inequalities under certain conditions. In particular, if we encounter an unequal byte at index i we check the bytes at positions $i + 1$ and $i + 4$ for equality (algorithm 2 lines 21, 24). If any of those bytes are equal, we continue regular prefix matching behind these new matching

indexes (lines 21-26). Additionally, if bytes were skipped, we increase the similarity by 0.5 or 2 for one or four skipped bytes respectively (lines 22, 25). The idea here is that parameter sequence that differ in a value choice and continue equal afterwards are still similar, but less similar than two sequences sharing the same values. This exception to prefix matching may lead to false positives, but we argue that if the control flow of the generator diverged, then following bytes will be highly likely to differ shortly after the divergence. The similarities of the candidate and each parameter sequence is divided by the size of the candidate for normalization and all similarities are summed up to get the total similarity (line 31). The candidate with the least such total similarity is then selected as a seed.
This process includes many comparisons and continuously increasing size of the set of parameter sequence for similarity computation. Still, we observed that this approach is competitive with random filtering in regard to time consumption. We explain this observation with a combination of random selection of candidates and quick termination of prefix matching for dissimilar sequences. First, the randomness in selecting parameter sequence candidates appears to pick diverse sequences. Therefore, many prefix comparisons will end after only a small fraction of bytes considered, possibly even at the first byte.
To illustrate this approach, we will present an example. Consider the following three parameter sequences that can created by our JavaScript generator:

1. (0,6,0,0,0,2,0,0,0,1,1,0,0,0,1,1,0,0,0,153,1,6,0,0,0)
2. (1,3,0,0,0,1)
3. (0,6,0,0,0,2,0,0,0,1,1,0,0,0,1,1,0,0,0,160,1,6,0,0,0)

For simplicity we used integers to represent byte values. Sequence 2 generates code that just contains a single "return" statement. Both parameter sequences 1 and 3 create a new identifier followed by an empty statement. Consider parameter sequence 1 as the only currently selected sequence (i.e. sequence 1 is the only element in set $\mathcal{O}$ in algorithm 2). Moreover, consider parameter sequences 2 and 3 as two candidates, and we want to select one of them. To do so, we will compute the similarity for both candidates to parameter sequence 1 using the procedure in algorithm 2 that starts at line 11. As described in line 6 of algorithm 2, we will select the candidate with the smaller similarity. When comparing sequences 1 and 2 the approach would find the first index unmatched (line 18). Next, it would look ahead one position (line 21), which is still not equal. Then, our approach would look ahead four positions (line 24) and find that both bytes at index 4 (indexing started with 0) are "0", resulting in a similarity score of 2 (line 25). Finally, the last index of parameter sequence 2 is reached and is not equal to the byte at the same index in 1. Since the end of sequence 1 is reached, no more comparisons are preformed and the current similarity value, which is 2, is divided by the length of the candidate 2. The length of sequence 1 is 5 resulting in a similarity score of $\frac{2}{5} = 0.4$ (line 31).
Looking at sequences 1 and 3 we can observe that they share the first 19 bytes, resulting in an initial similarity of 19 (lines 18,19). The bytes at index 19 differ and thus the techniques looks at index 20 (line 21) where it finds a match again. This increases the

similarity to 19.5 (line 22). The other remaining six bytes match, leading to a similarity score of 25.5 (lines 18,19). Divided by the length of parameter sequence 3 we get a similarity of roughly 0.98 (line 31). Thus, according to our approach, sequences 2 and 3 are nearly the same while 1 and 2 are more different from one another. Specifically, in line 6 of algorithm 2 parameter sequence 3 is selected.
It is important to note that in reality, most parameter sequences will have hundreds or even thousands of bytes.

Choice Sequence Filtering In the previous approach, we used information embedded in parameter sequences to determine similarity. The problem with that approach is that information inside the parameter sequence are encoded with knowledge about the generator they are created for. In other words, to determine where a specific byte is being used, one would need to run the targeted generator with the parameter sequence until the specific byte is being read. In *choice sequence filtering*, we want to exploit more accessible information on the inputs. Furthermore, we want to utilize the choice sequences that RLCheck [53] uses to learn its choice policies. A choice sequence from RLCheck's JavaScript generator could look like this:

```
[node=statement, node=throw, node=expression, node=ident].
```

Single choice points are separate by a comma. The sequence can be interpreted as follows: First, a statement should be generated, RLCheck decides to generate a throw statement, within this throw node RLCheck decided to throw something that is an expression, lastly, RLCheck decides to generate an identifier as the expression. In other words, the generated input will be a throw statement that contains a single identifier. We also want to emphasize the importance of the auxiliary nodes expression and statement. Those nodes represent a choice point where the generator can generate different structures within the input. So to determine structural similarity of inputs, it would be a great idea to look at the elements that follow after expression and statement nodes. More general, we can observe similarity of inputs by looking at transitions between different entries in the choice sequences used to generate these inputs. Another shortcoming of the parameter sequence filtering is that similar structures in the input can only be detected when they appear in the same position. Using transitions between different elements of choice sequences allows us to determine similar structures anywhere in the input. Using these observations, we designed choice sequence filtering.
Before we present the details of this approach, we will need to discuss the *availability of choice sequences.* In particular, RLCheck trims the choice sequences to a fixed size to maintain efficiency in its reinforcement learning process. We leave RLCheck's handling of choice sequences untouched, but additionally create a duplicate choice sequence that contains all elements. We activate the collection of complete choice sequences process through our generator interface described in 4.1. Moreover, we also obtain all choice sequences through this interface.
Algorithm 3 shows pseudocode for this approach. The approach gets a set of parameter sequences and a set of choice sequences. Notably, each parameter sequence has exactly

Algorithm 3 Choice Sequence Filtering

```
    Input: $\mathcal{S}$                              ▷ $\mathcal{S}$ Unfiltered set of parameter sequences
    Input: $\mathcal{C}$                              ▷ $\mathcal{C}$ Set of choice sequences of same size as $\mathcal{S}$
    Input: $\beta : \mathcal{C} \to \mathcal{S}$      ▷ $\beta$ Maps each element of $\mathcal{C}$ to its corresponding element in $\mathcal{S}$
    Output: $\mathcal{O}$                             ▷ $\mathcal{O}$ Filtered set of parameter sequences
 1: $\mathcal{O} \leftarrow \emptyset$
 2: $\mathcal{C}_{curr} \leftarrow \emptyset$         ▷ Set of currently selected choice sequences
 3: choose $c_1$ randomly from $\mathcal{C}_{curr}$
 4: $\mathcal{C}_{curr} \leftarrow \mathcal{C}_{curr} \cup \{c_1\}$
 5: while $|\mathcal{C}_{curr}| < output\_size$ do
 6:     randomly choose candidate set $\mathcal{C}_{can}$ from $\mathcal{C}$
 7:     select $c_{min} = \arg\min_{c \in \mathcal{C}_{can}} computeChoiceSeqSimilarity(c, \mathcal{C}_{curr})$
 8:     $\mathcal{C}_{curr} \leftarrow \mathcal{C}_{curr} \cup \{c_{min}\}$
 9: end while
10: for $c \in C_{curr}$ do
11:     $\mathcal{O} \leftarrow \mathcal{O} \cup \beta(c)$
12: end for
13: return $\mathcal{O}$
14:
15: procedure $computeChoiceSeqSimilarity(c, \mathcal{C}_{curr})$
16:     totalSimilarity $\leftarrow 0$
17:     for $c' \in \mathcal{C}_{curr}$ do
18:         transitions $\leftarrow getAllPossibleTransitions(c, c')$
19:         transition_counts$_c \leftarrow getTransitionCounts(transitions, c)$
20:         transition_counts$_{c'} \leftarrow getTransitionCounts(transitions, c')$
21:         totalSimilarity $\leftarrow$ totalSimilarity + $cosineSimilarity($transition_counts$_c$,transition_counts$_{c'})$
22:     end for
23:     return totalSimilarity
24: end procedure
```

one corresponding choice sequence, which we visualize with the bijective function β. Additionally, it is important to remember that we want to write parameter sequences to seed files, but we use their corresponding choice sequences for filtering. For ease of understanding, we will talk about filtering choice sequences instead of filtering parameter sequences based on their corresponding choice sequences in this section.
First, we initialize a set of choice sequences to keep track of already selected sequences (line 2). A first choice sequences is selected randomly and added to this set (lines 3,4). Next, we perform a loop representing the select-test-from-candidates strategy as described in 3.6. Concretely, until the output size is reached, a set of random candidates is selected, the similarity of the candidate with the set of already selected choice sequences is computed, and the candidate with minimal similarity is selected (lines 5-8).
So far, this has been similar to parameter sequence filtering. The difference lies in the computation of similarity that is visualized in the procedure starting at line 15. Similarity

computation for choice sequence filtering is based on two pillars, namely *transition counts in choice sequences* and *cosine similarity*. The general idea is to compare choice sequences based on frequency of transition between entries in the choice sequences. In other words, for each ordered pair of entries, we count the frequency of the pair in the choice sequence and call this *transition counts*. We also call ordered pairs of entries in choice sequences *transitions*. Notably, we increase the transition count of a transition both items of the transition appear in the choice sequences in the correct order. To compute the similarity score, we decided to use the cosine similarity, which is a well-known method to compute the similarity of documents [3].

Definition 4.1 (Cosine similarity)**.** Let x, y be vectors with values in $\mathbb{Z}$.
The cosine similarity is defined as:

$$sim(x, y) = \frac{x \cdot y}{||x||||y||} \tag{1}$$

Where $||x||$ is the euclidean norm of vector x.

Definition 4.1 shows how the cosine similarity is computed. Mathematically, this formula computes the cosine of the angle between two vectors. More intuitively, if the cosine similarity of two vectors is high, then both vectors point in a similar direction. Therefore, we encode the transition counts of choice sequences as vectors. Concretely, we first derive all transitions that are present in any of the two in choice sequences we want to compare (algorithm 3 line 18), compute the transition counts for each of those transitions in both choice sequences (lines 19,20) and compute the cosine similarity of both vectors of transition counts (line 21). This comparison is repeated with each already selected choice sequence for a single candidate (line 17). Furthermore, the similarities of the selected choice sequences and the candidate are added up to obtain the total similarity of the candidate (line 21). This total similarity is used in line 7 to select the candidate with minimal similarity.
In our implementation, we make heavy use of caching to avoid recomputing transition counts and euclidean norms. Still, we observed filtering times of up to ten minutes in preliminary experiments. The problem of the approach is that we compare each candidate with each already selected choice sequence, while also increasing the size of the set of selected sequences continuously (line 8). Therefore, we decide to compare the candidate only to a fixed size of already selected choice sequences. Concretely, we compare the candidate to the last hundred selected sequences.
To illustrate this filtering approach, we will present an example. Consider the following three choice sequences from our JavaScript generator:

1. `[node=statement, node=expression, node=literal, index=0, node=expression, node=binary, binary=-, node=expression, node=ident, node=expression, node=litera`

2. `[node=statement, node=expression, node=literal, index=0, node=expression, node=ident]`

3. `[node=statement, node=expression, node=binary, binary=-, node=expression, node=ident, node=expression, node=ident]`

Choice sequence 1 is used to generate a literal with one element inside itself (index=0 means that the literal is some container that has an entry at index 0). This element is a binary expression using the subtraction operator. Furthermore, the first element of the binary operator is an identifier and the second element is another literal. Choice sequence 2 is responsible for generating another literal with one element. This time, the element in the literal is an identifier. Choice sequence 3 is used to generate a binary expression using the subtraction operator. Both arguments of the binary expressions are identifiers. Although the inputs generated by these choice sequences seem different, they still share some properties that are likely to be treated similarly by the PUT. For example, choice sequences 1 and 3 both contain a subtraction expression, while sequences 1 and 2 share a literal that is some container holding one item. It is also worth mentioning that some of these shared properties could not be detected by parameter sequence filtering since they appear in different locations in the different choice sequences (e.g. the binary expression in sequences 1 and 3). Continuing our example, let's assume choice sequence 1 as the set of already selected choice sequences and let's consider that sequences 2 and 3 are candidates. Furthermore, we want to select one of those candidates that has minimal similarity to the already selected choice sequence 1). We will use the procedure in algorithm 3 lines 15-23 to compute the similarities. The set of already selected choice sequences only contains sequence 1. Thus, we only need to compare both choice sequences 2 and 3 to 1 and select the one with the smaller similarity (algorithm 3 line 7). Table 1 presents the transitions

Transition	Transition Count 1	Transition Count 2	Transition Count 3
statement, expression	1	1	1
expression, literal	2	1	0
literal, index=0	1	1	0
index=0, expression	1	1	0
expression, binary	1	0	1
binary, binary=-	1	0	1
binary=-, expression	1	0	1
expression, identifier	1	1	2
identifier, expression	1	0	1

Table 1: Transitions and their counts for choice sequences 1, 2 and 3.

and their counts for the three choice sequences. The first column presents all transition present in the choice sequences (we omitted "node=" for readability). Column two to four presents the transition counts of choice sequences 1, 2 and 3 for the transition in the first column. In other words, column one presents the result of line 18 in algorithm 3 and columns two to four present the results of lines 19 and 20. We can now use columns two

to four as vectors and use them in definition 4.1 to compute similarity (algorithm 3 line 21). After performing those computations, we get a similarity of roughly 0.78 for choice sequences 1, 2 and roughly 0.67 for sequences 1, 3 respectively. Consequently, choice sequence 3 would be selected in line 7 in algorithm 3 as it has the smaller similarity to the already selected sequence.
We should mention that the example we presented is kept short for readability. In reality, most choice sequences have hundreds of entries. Furthermore, the set of already selected choice sequences usually contains more than one choice sequence, resulting in many more similarity computations.

Generator Coverage Filtering Our last filtering approach, generator coverage filtering, is based on the ideas of coverage-guided fuzzing. Instead of collecting coverage from the PUT, we gather coverage from the generator. We will call this coverage data *generator coverage*. Equipped with the generator coverage, we can reason about the control flow through a generator that led to the generation of an input. Moreover, we can use this information to determine if a run through the generator archived new generator coverage. Inputs that increased generator coverage can be considered interesting in accordance to the terminology of coverage-guided fuzzing. For such inputs, we want to write their corresponding parameter sequences to seed files.

Algorithm 4 shows how generator coverage filtering works. The algorithm gets the

Algorithm 4 Generator Coverage Filtering

```
    Input: Generator $\mathcal{G}$                          ▷ $\mathcal{G}$ generator that is used
    Output: $\mathcal{O}$                     ▷ $\mathcal{O}$ filtered set of parameter sequences
 1: Instrument $\mathcal{G}$
 2: totalCoverage ← $\emptyset$                  ▷ current generator coverage of $\mathcal{O}$
 3: $\mathcal{O} \leftarrow \emptyset$
 4: while !timeout do
 5:     coverage, parameterSeq ← $\mathcal{G}$.generate()
 6:     if |totalCoverage ∪ coverage | > | totalCoverage | then
 7:         totalCoverage ← |totalCoverage ∪ coverage|
 8:         $\mathcal{O} \leftarrow \mathcal{O} \cup$ {parameterSeq}
 9:     end if
10: end while
11: return $\mathcal{O}$
```

targeted generator as an input and should return a set of filtered parameter sequences. First, the generator is instrumented so that we can collect coverage data from it (line 1). In the previous approaches, we filtered after the generation process was finished. Contrary, for generator coverage filtering, we filter during the generation process and thus are bound to the timeout given to the generator (line 4). Until the timeout is reached, the generator generates inputs, emitting generator coverage data and the resulting parameter sequence (line 5). The generator coverage is then compared to the already known generator coverage from previous generation runs (line 6). If the new input increased the

generator coverage, its corresponding parameter sequence gets selected as a seed and its new coverage gets added to currently known generator coverage (lines 7,8).
Instrumentation of the generator is conducted using JQF's [44] implementation of instrumentation that is based on the ASM byte code manipulation tool for Java [1]. Furthermore, we excluded every class from instrumentation except for our generators. The concrete coverage we collect is branch coverage, meaning that executed branches characterize the coverage data.

4.3 Integration into RLCheck

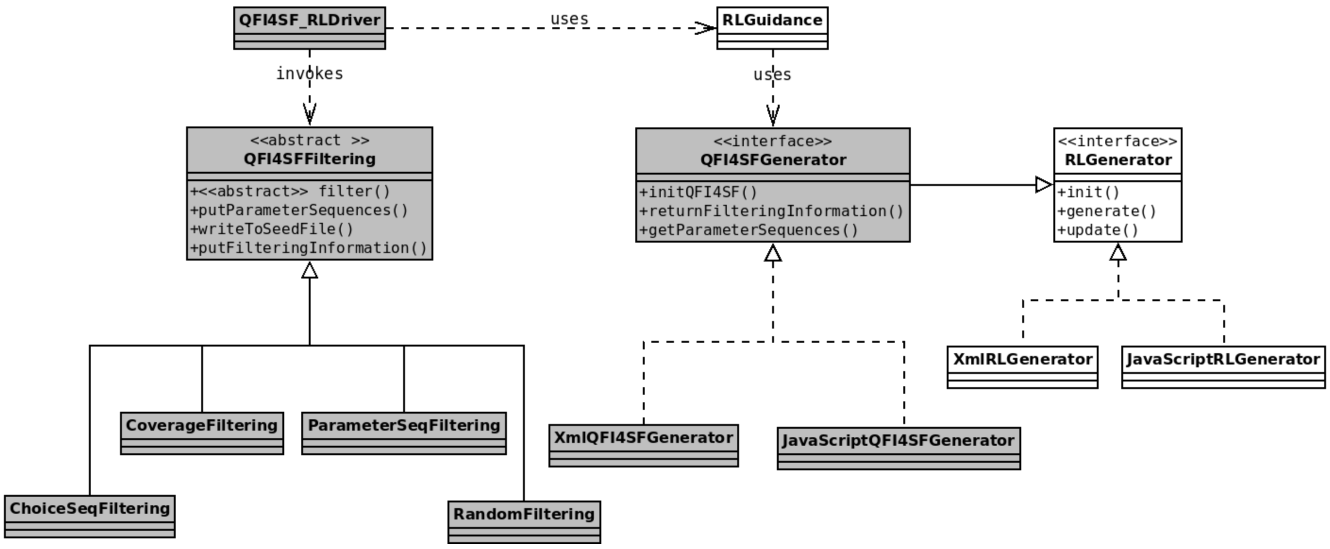


Figure 4: Model of our implementation. Grey classes and interfaces are novel parts, white classes and interfaces are part of RLCheck. The model only contains classes relevant to our approach and omits many methods for readability.

A conceptual overview of our implementation is given in figure 4. Grey boxes indicate units developed as part of our work, white box are part of RLCheck [53]. The model is incomplete and only depicts parts relevant to our implementations and its understanding. Our implementation is tightly integrated into RLCheck which itself extends JQF [44].
Following section 4.1 we developed a novel interface for generators. The interface itself extends RLCheck's own interface for generators to enable communication of parameter sequences and filtering information. Additionally, we implemented two novel generators for the two file formats XML and JavaScript. Both generators possess equal generational behaviour as their counterparts in RLCheck. However, they have additional functionality to construct parameter sequences as described in section 4.1.
We extended JQF to be able to filter parameter sequences as described in section 4.2. Moreover, we developed an abstract class for filtering techniques and extended that class with four concrete implementations.
Our approach is stitched together by the novel driver for our approach. The driver

gets arguments specifying the filtering approach, test method and target program. It then initializes the generator to capture filtering-specific information. The generator is passed to RLCheck's standard guidance that starts the fuzzing process. During fuzzing, the generators collect parameter sequences and filtering information. When fuzzing is finished, the driver extracts parameter sequences and filtering information, transmits them to the filtering technique and invokes filtering. Whenever the filtering technique selects a parameter sequence as a seed, it writes the parameter sequence to a new seed file.
Our approach uses the same property based tests as RLCheck. Moreover, as we use RLCheck's guidance, we also leverage JQF's test runners and instrumentation to run the PUT and collect coverage and execution results.

5 Evaluation

In this section, we describe how we empirically evaluated our new approach. We compare Zest [45] without seeds to Zest that uses seed corpora generated by our new approaches. In particular, we aim to answer the following research questions:

RQ1 Can our new seed corpora generation approach improve the efficiency of a state-of-the-art fuzzer to discover new coverage? (section 5.2)

RQ2 Can our new seed corpora generation approach improve the fault-finding capability of a state-of-the-art fuzzer? (section 5.3)

RQ3 Is any particular filtering technique superior to the other techniques in terms of coverage discovery or fault-finding capability? (section 5.4)

5.1 Study Design

Techniques for Comparison We compare five approaches to answer our research questions. The first approach is Zest [45] without any seeds, which is the baseline we seek to improve. The other four techniques consist of Zest with a seed corpora generated by using one of our four new filtering approaches. Concretely, we refer to the four new techniques as CHO (choice sequence filtering and Zest), RND (random filtering and Zest), PAR (parameter sequence filtering and Zest), and COV (generator coverage filtering and Zest).

Subjects The subjects to our evaluation are the four real-world Java programs Apache Ant, Apache Maven, Google Closure Compiler and Mozilla Rhino. We use these benchmarks because they were used in the original evaluation of Zest [45] and RLCheck [53]. Moreover, each of these programs is widely used, making them a good representation of real-world software. Ant and Maven use XML files as inputs, while Closure and Rhino make use of JavaScript files.

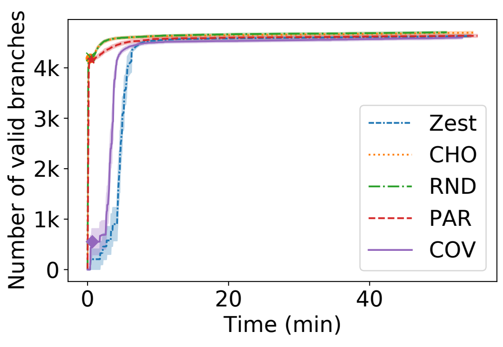
Configuration For generation of initial seeds, we use the latest version of RLCheck [53] (from the 20th September 2021). For Zest, we use the latest version of JQF [44] available when conducting the experiments (from the 7th June 2022). In RLCheck we use the configuration files delivered with RLCheck for all benchmarks. However, for each trial of our experiment, we changed the "seed" entry in the configuration file to a random value. This entry is used to seed the souce of randomness used by RLCheck. XML generators in RLCheck and Zest make use of dictionaries to obtain strings. We use the dictionaries that come with RLCheck and Zest for both Maven and Ant (both have their own dictionaries). We use the default configuration for each subject.
When generating initial seeds, we use a timeout of one minute for RLCheck. We choose this value because the coverage archived by RLCheck starts to plateau after around one minute in RLCheck's evaluation. CHO, RND and PAR require an output size to be defined (see section 4.2). We decide to set the output size to roughly five percent of the amount of inputs that Zest can generate in the timeout for the experiment. We made this choice for the output size to allow Zest to build on top of the seeds we supplied instead of being stuck with evaluating seeds for the majority of the experiment. Concretely, for timeouts of one hour this resulted in output sizes of 3500, 7600, 7800 and 3400 for Ant, Maven, Rhino and Closure respectively. Contrary, COV does not require an output size, since the number of seeds is dependent on the number of unique branches in the generators of RLCheck. We observe that COV always generates around 50 seeds for JavaScript generators and around 25 seeds for XML generators.

Experimental Design We don't change Zest's mutation strategy or other parts of its fuzzing loop. Therefore, our new approach won't uncover coverage or failures that Zest without seeds can't eventually find, given enough time. Consequently, we focus on investigating if our approach can increase Zest's efficiency of finding new failures and coverage. Hence, we choose a timeout of one hour for a single trial. For Zest, a trial is simply executing Zest for one hour with one of the subjects as the target program. A trial for our new approaches consists of executing RLCheck with the subject to obtain initial parameter sequences, invoke filtering to produce seed files, and hand the seed files to Zest to start its fuzzing campaign. Moreover, for a fair comparison, we record the time used for generating the seed corpus(i.e. RLCheck and filtering), subtract the recorded time from the timeout and let Zest fuzz until the new timeout is reached. We include the generation of seed corpora in trials because it involves non-deterministic choices both in RLCheck and in the filtering process. However, in practice, seed corpora generation has to be conducted only once. To account for statistical validity in experiments with non-determinism, we repeated each trial twenty times (as suggested in [31]). In total, we ran each of the five techniques with each of the four benchmarks with twenty repetitions, which results in 400 trials in total.
During each execution of Zest (including executions with seeds) we collect branch coverage to answer RQ1 and RQ3. Moreover, for the techniques with seeds, we also record the timestamp and coverage level when Zest finished using the seeds. When Zest is initialized with seeds, it first runs all seeds with the subject to determine which it should keep for

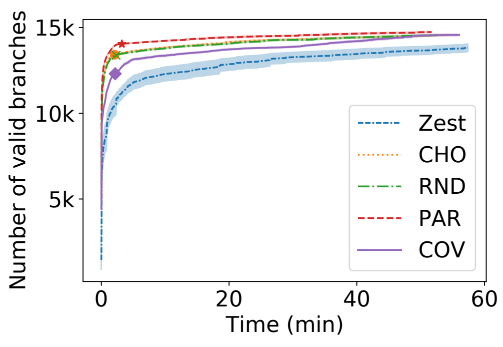
further mutation. In that way, we can observe the coverage archived by only executing the seeds. We collect coverage archived by valid inputs (i.e. *valid coverage*) and coverage archived by all inputs. However, we found that both coverage measures lead to the same observations and omit non-valid coverage from our discussions. Additionally, we decided to choose valid coverage, because Zest's primary goal is to create valid inputs.
To answer RQ2 and RQ3 we collect failures triggered during the executions of Zest (including executions with seeds). As suggested by Klees et al. [31], unique failures should be used to evaluate fuzzing techniques. Identifying unique failures requires deduplication of stack traces obtained alongside the failures. To do so, we leverage the technique described in [13]. Concretely, for every two failures from the same subject, we compare the root cause of the failures (i.e. the exception) and the following three stack frames. If these four elements match, we discard one of the failures. Repeating this process for all failures leaves us with the set of unique failures. To determine efficiency of failure discovery, we also record the *time-to-error* for each unique failure. Time-to-error describes the timestamp of the first encounter of a specific failure in a given trial. We also determine *reliability* of detecting failures by counting the number of trials that can find a given failure.
To detect statistical significance, we use the Mann-Whitney-U-Test with a significance level of $\alpha = 0.01$.
We conducted our experiments using a server with an Intel(R) Xeon(R) E7-4880 2.5GHz CPU and 1 TB of RAM running openSUSE Leap 15.

5.2 RQ1: Coverage

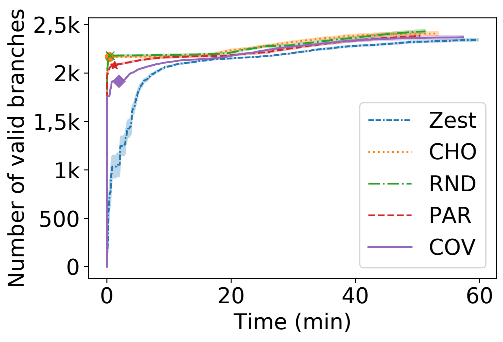
To answer RQ1 we present the coverage archived by the five approaches. Figure 5 shows the mean of the total number of branches covered by valid inputs over twenty runs, including the standard deviation. The graphs show the coverage of executions of Zest including the runs with seeds (for CHO, RND, PAR and COV). We note that the different length of plots is a result of the timeout reduction for CHO, RND, PAR and COV according to the time they needed to create the seed corpora.
Looking at valid coverage archived after one hour, we can't claim a clear advantage of our filtering approaches against Zest. In particular, for Ant, we observe a similar coverage level for all approaches after one hour and also report that none of the differences between coverage levels after one hour is statistically significant. Moreover, for Rhino we make a very similar observation but report a statistical significance of differences in valid coverage levels for Zest and CHO (p-value ≈ 0.002), for Zest and RND (p-value ≈ 0.009), and for Zest and COV (p-value ≈ 0.001). After one hour CHO, RND and COV can achieve a small advantage over Zest, which is significant due to low deviations for Rhino. Hence, for Rhino and Ant, we can't claim an advantage of our new techniques in regard to total valid coverage after one hour when compared to Zest. However, we make different observations for Closure and Maven. In particular, for Closure every single filtering technique (CHO, RND, PAR and COV) archives significantly more valid coverage than Zest after one hour (with p-values of $0,0002$ and smaller). Moreover, the differences in levels of valid coverage are in the range of 700 to 1000. For Maven, we



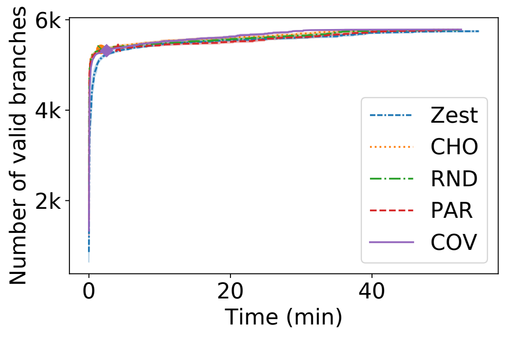
(a) Valid Coverage Ant



(b) Valid Coverage Closure



(c) Valid Coverage Maven



(d) Valid Coverage Rhino

Figure 5: Mean of total number of branches covered by valid inputs (i.e. valid branches) over time for twenty trials. Markers on the lines of CHO, RND, PAR and COV show the time and coverage level when their instances of Zest stopped using seeds. Higher is better.

observe a diverse picture. Concretely, PAR and COV fail to archive significantly more coverage after one hour, ending up with tiny advantages over Zest. However, RND and CHO can archive significantly more coverage than Zest after one hour (with p-values of ≈ 0.0009 and ≈ 0.009 respectively). Wrapping up the observations regarding total valid coverage after one hour, we observe a mixed picture. For two subjects (Ant and Rhino) our new filtering techniques can't record an improvement of Zest, while for the other subjects (Maven and Closure) at least some of our approaches can outperform Zest in regard to total valid coverage after one hour.
However, our new approaches focus on the starting point of Zest (i.e. the seeds) not its internal mechanics. Therefore, we expect a gain of efficiency in uncovering valid coverage instead of advantages in regard to total valid coverage. Looking at the graphs in figure 5 we observe that our filtering techniques can drastically increase coverage in the first few minutes. In particular, for Ant, Closure and Maven we observe a clear advantage of our filtering approaches as compared with Zest. However, for Rhino we can't observe mayor advantages of our techniques over Zest. We believe that our approaches perform similar to Zest for Rhino because Rhino has the weakest validity constraints. Therefore,

even randomly generated initial inputs can archive high coverage in the core program logic, which limits the impact of initial seeds. We claim that the advantage in early coverage can be attributed to the coverage archived by the initial seeds. To present evidence for this claim, we added a marker to the graphs in figure 5 to indicate the mean time and levels of valid coverage that are recorded when Zest finished using the seeds. A common observation is that the peak in coverage begins to plateau right after the seeds have been executed. This suggests that the seeds are responsible for the initial peak of branches covered. However, we should note that after Zest finished running the seeds, it can still increase coverage itself, suggesting that Zest's mechanics are still necessary to cover certain parts of the PUT. To further discover the increase in efficiency, we present table 2.
In table 2 we present the average time that Zest without seeds took to reach the same

Technique	**Subject**	**Seed Coverage**	**Seeds finished**	**Zest catch up**
CHO	Ant	4187.3	0.40	6.27
CHO	Maven	2165.75	0.46	24.00
CHO	Closure	13414.3	2.09	37.93
CHO	Rhino	5344.85	1.70	5.30
RND	Ant	4216.9	0.41	6.27
RND	Maven	2178.7	0.54	26.17
RND	Closure	13378.8	2.31	36.20
RND	Rhino	5340.25	1.82	4.98
PAR	Ant	4160.6	0.57	6.27
PAR	Maven	2079.5	1.25	10.65
PAR	Closure	14041.05	3.21	Never reached
PAR	Rhino	5372.8	4.12	6.13
COV	Ant	552.45	0.68	2.72
COV	Maven	1915.05	1.95	5.93
COV	Closure	12305.05	2.16	10.28
COV	Rhino	5315.6	2.50	4.32

Table 2: The time that Zest takes to get to the same valid coverage as the seeds archived in CHO, RND, PAR and COV. Column one and two present the filtering technique and the subject. Column three presents the mean number of branches discovered by valid seed inputs. Column four presents the mean time in minutes that Zest spent to run the subjects with the seeds for the technique and subject in columns one and two. Column five presents the mean time in minutes that Zest without seeds took to get to the valid coverage presented in column three for the subject in column two. Cells in column two are coloured corresponding to their benchmark to aid comparability.

level of valid coverage as archived by the seeds generated by our new techniques. Looking at a single row in table 2 we present the filtering technique F in column one. The second column represents the subject S being executed. In column three, we present the mean number of branches C discovered by valid seeds generated by technique F for subject S. Column four presents the mean time that Zest executed S with seeds obtained from F. The last column presents the mean time Zest without seeds spent to archive valid coverage C when fuzzing S. In other words, the last column presents the time Zest without seeds required to archive the same coverage as the seeds. Time data is always presented in minutes. The key observation from this table is that the seeds can reach a certain level of coverage in less time than Zest. Concretely, for Ant CHO, RND and PAR can get to the seed coverage between 12× and 15× faster when Zest. The data for Maven and Closure reveals even bigger advantages of our approach. For Maven, we observe that Zest takes between around 8× and 50× longer to reach the valid seed coverage compared against CHO, RND and PAR. We observe similar advantages of CHO, RND and PAR for closure. Notably, Zest without seeds is never able to reach the valid seed coverage archived by PAR for closure and takes more than 35 minutes to reach the seed coverage of for CHO and RND. For Rhino, we observe a smaller advantage. In particular, for Rhino, Zest takes between around 1.5× (for PAR) to around 3× (for CHO) to reach the seed coverage. The worse performance on Rhino can be explained with the weak validity constraints it poses on inputs. Concretely, randomly generated inputs to Rhino will quickly explore its core logic, which limits the impact of seeds. We also observe that COV usually generates less seed coverage than other approaches, and Zest without seeds needs less time to reach the seed coverage of COV. We will discuss the difference between our filtering techniques in section 5.4. Overall, we observe an advantage of our new techniques in efficiently discovering new coverage. This suggests that our approach is especially useful in situations with limit resources, where quickly fuzzing an PUT is required.

5.3 RQ2: Failures

Failure ID	Zest	CHO	RND	PAR	COV
ant_0	20	20	20	20	20
closure_0	20	20	20	20	20
rhino_0	20	20	20	20	20
rhino_1	20	20	20	20	20
rhino_2	20	20	20	20	20
rhino_3	0	2	3	1	1
rhino_4	13	9	6	3	8
rhino_5	1	0	1	0	2

Table 3: Reliability of failure discovery. Integers present the number of trials that found a particular failure out of twenty trials. Grey cells highlight the highest reliability for a particular failure.

We want to answer RQ2 by analysing reliability and time-to-error of each unique failure.

During our experiments, we found eight unique failures. Concretely, we found one unique failure in Ant, no failures in Maven, one unique failure in Closure, and six unique failures in Rhino in our trials of one hour.
In table 3 we present the reliability of failure discovery for every unique failure. We found that the failures of Ant and closure alongside failures rhino_0, rhino_1 and rhino_2 can be found in each trial, suggesting that they can be triggered easily. Failure rhino_3 cannot be found in any trial of Zest, but can be found at least once for every other technique. This suggests that the discovery of rhino_3 is more likely when seeds are being used. However, the reliability of finding rhino_3 is at most three out of twenty trials (for RND) for our filtering techniques, which is a relatively small value. Failure rhino_4 is most reliably found by Zest (thirteen out of twenty trials), followed by CHO (nine out of twenty trials) and COV (eight out of twenty trials). Therefore, we believe that the chance of detecting rhino_4 is increased if Zest can apply its own exploration of the program logic without being biased by seeds. However, the good reliability of CHO and COV suggest that the presence of diverse program structures inside the seed corpora also benefits the detection of rhino_4. Concretely, CHO and COV are the techniques that pay the most attention to actual structures inside the inputs. The most unreliably discovered failure is rhino_5. This failure can only be discovered once by Zest, once by RND and twice by COV. This suggests that rhino_5 is deep inside the program logic, which makes is unlikely to discover the failure within one hour of fuzzing. Overall, we can't argue that any approach has a clear advantages against the other approaches in regard to fault-finding reliability. Furthermore, in terms of total bugs found for each technique, there is only one statistically significant difference. In particular, Zest can statistically significantly find more failures than PAR in the Rhino subject (p-value ≈ 0.001).
The previous results on reliability only present half of the picture. Specifically, the

Failure ID	**Zest**	**CHO**	**RND**	**PAR**	**COV**
ant_0	7.503	3.31*	5.19	5.278*	5.946
closure_0	3.313	0.15*	0.129*	0.072*	0.362
rhino_0	1.501	0.342*	0.55*	0.304*	0.305*
rhino_1	1.405	0.162*	0.233*	0.147*	0.042*
rhino_2	2.507	0.121*	0.158*	0.486*	1.646
rhino_3	NaN	3.8	32.7	32.967	10.133
rhino_4	19.976	29.306	33.097	40.883	13.398
rhino_5	46.017	NaN	19.083	NaN	47.508

Table 4: Mean time-to-error in minutes over twenty trials. "*" behind a number means that the technique has a statistically significant difference in time-to-error compared to Zest. Grey cells mark the smallest time-to-error for the failure in column one.

efficiency of uncovering these failures is important too, especially for our approach that aims to discover new coverage and failure faster than Zest without seeds. Therefore, we present table 4. The table shows the mean time-to-error over twenty trials for each technique and each subject. Grey cells highlight the smallest time-to-error for a particular failure. Numbers marked with a "*" symbolize that the mean time-to-error is statistically significantly smaller when Zest's time-to-error (without seeds). Looking at the five unique failures with 100% reliability for each technique (i.e. ant_0, closure_0, rhino_0, rhino_1 and rhino_2), we observe that every mean time-to-error of our new filtering technique is smaller than Zest's. This observation suggests that the presence of seeds can increase the efficiency of discovering these failures. Moreover, for each of the failures with 100% reliability, CHO and PAR have a significant difference in their time-to-error to Zest (without seeds) and for four of these failures CHO or PAR archive the smallest mean time-to-error. Interpreting these result, we attribute the success of CHO and PAR to their strong similarity criterion, which allows them to supply diverse seeds that can trigger the failure frequently with each seed corpora. Notably, PAR found closure_0 parameter faster than every other approach, which aligns with PAR's superior ability to increase coverage in Closure (see section 5.2 for details). Another interesting observation from the failures with 100% reliability is that COV can significantly outperform every other technique for rhino_1. Concretely, for rhino_1, COV archives the smallest mean time-to-error of all measurements. This suggests that rhino_1 can be quickly triggered by a type of input that increased coverage in RLCheck's generator. Time-to-error for rhino_3 and rhino_5 is hard to interpret, as the reliability for these failures is very small. Therefore, to reason about the efficiency of triggering these failures, we would need more data. However, we note that rhino_3 is found very quickly by the CHO when compared to other approaches, which implies that CHO might be able to select seeds that can be easily be mutated into inputs triggering this failure. Failure rhino_4 is the only case where Zest can find the failure faster than three of our new techniques. Concretely, Zest has a smaller time-to-error than CHO, RND and PAR for rhino_4. However, we note that none of these differences is statistically significant (p-values ≈ 0.39 for CHO, ≈ 0.18 for RND and ≈ 0.04 for PAR). Another observation for rhino_4 is that COV archives a smaller mean time-to-error than Zest. However, the difference between Zest and COV is not statistically significant, with a p-value of ≈ 0.28. The success of COV for rhino_4 together with its reliability data (table 3) leads us to the conclusion that rhino_4 is a failure that has a higher chance of being discovered when Zest is not biased by seeds. Concretely, COV supplies a much smaller set of seeds, which gives Zest more freedom. However, the lower reliability for COV in rhino_4 suggest that the seeds in COV are not directly responsible for the discovery of rhino_4 and require the right mutations to be made in Zest.

Another interesting observation is that closure_0, rhino_0, rhino_1, and rhino_2 have a smaller time-to-error for all our filtering techniques than the time that Zest executes the seeds displayed in table 2, which means that these failures are directly discovered using the seeds.

Concluding our observations for failures, we observe no significant difference in the reliability of our new techniques compared against Zest. This observation is in line with our claim that we won't increase Zest's effectiveness. However, we observe a significant advantage in terms of efficiency in failure discovery. This is a shared observation with the results of RQ1 for coverage discovery in section 5.2 which suggest a connection between both results. In other words, the more efficient discovery of coverage might benefit the more efficient discover of unique failures.

5.4 RQ3: Comparing Filtering Techniques

In order to answer RQ3, we will revisit the results presented in sections 5.2 and 5.3. Concretely, we compare CHO, RND, PAR and COV against each other. First, we present table 5. In table 5 we present the mean time in minutes that each filtering technique spend

Technique	Subject	Mean time to seed corpous (min)	Standard deviation
CHO	Ant	2.24	0.02
CHO	Maven	3.58	0.08
CHO	Closure	2.19	0.05
CHO	Rhino	3.89	0.12
RND	Ant	2.70	0.07
RND	Maven	4.74	0.11
RND	Closure	2.68	0.02
RND	Rhino	4.38	0.24
PAR	Ant	2.77	0.02
PAR	Maven	4.74	0.12
PAR	Closure	2.61	0.14
PAR	Rhino	5.01	0.09
COV	Ant	1.02	0.01
COV	Maven	1.03	0.01
COV	Closure	1.01	0.01
COV	Rhino	1.02	0.00

Table 5: Mean duration in minutes, over twenty runs, for generating the seed corpora for each technique and each subject. Column four presents the standard deviation for the mean in column three. Cells in column two are coloured according to their subject to ease comparability.

to run RLCheck and filter the results for each subject. In our experiments, RLCheck is always run for one minute. Therefore, the time spent for filtering is the value in column three subtracted by one minute. COV is filtering during the generation process, which results in the small durations in column three. The other filtering techniques (CHO, RND and PAR) don't differ substantially in their filtering durations. We note that the standard deviation (column four) is small for every technique, suggesting that our filtering techniques can consistently generate seed corpora in the time presented in column three. Looking at the valid coverage achieved by our new techniques after one hour, we observe that for Ant, Closure and Rhino none of our approaches archives statically significantly different coverage than any of the other filtering techniques. However, for Closure, PAR

archives statistically significantly more valid coverage after one hour than CHO (p-value $\approx$ 0.007) and COV (p-value $\approx$ 0.008). Looking at table 2 we can also observe that the seeds generated by PAR archive the highest levels of valid coverage for Closure. Those observations cement PAR's advantage for Closure. Further analysing table 2, we observe that seeds created by COV archive less coverage than the other approaches for all benchmarks except Rhino. We reiterate the argument that Rhino has the loosest validity constraints, which allows any input to archive high coverage. The disadvantage of COV in terms of seed coverage is particularly striking for Ant where it can cover around 550 branches while all other approaches can cover more than 4000 branches. We believe that this observation can be explained with the strict validity constraints of Ant. Moreover, we argue that COV generates too few seeds to solve many of Ant's validity constraints, while our other approaches can fulfil the constraints due to the sheer number of seeds. Another observation from table 2 and figure 5 is that CHO and RND tend to perform similarly. Moreover, their coverage after one hour is never significantly different, and their seeds produce similar coverage. This indicates that CHO's selection criterion for seeds does not substantially differ from random selection. We believe that this can be explained with the fixed number of previous seeds considered for similarity in CHO. Furthermore, we note that RND is competitive with the other approaches in terms of coverage, which indicates that the sheer number of seeds is relevant for boosting coverage discovery.
Looking at the reliability of finding unique failures (i.e. table 3), no filtering technique can significantly outperform the other techniques. However, we note that PAR performs worse compared to other approaches in terms of finding the rare failures (i.e. rhino_3, rhino_4 and rhino_5). Moreover, all these rare failures are inside the Rhino subject, where PAR can't record advantages in coverage.
Comparing our new approaches in terms of mean time-to-error (with table 4) leads to a mixed picture. CHO can archive the smallest time-to-error for ant_0 and rhino_2 where it also archives a statistical difference to all other approaches except for RND. This observation cements our claim that CHO and RND perform similarly. However, CHO always archives smaller time-to-error than RND except for rhino_5 (which has a low reliability for all approaches). Therefore, we argue that while RND and COV behave similar for coverage, CHO still outperforms RND in terms of efficiency of failure discovery. Another observation that confirms PAR's advantage on Closure is the mean time-to-error of closure_0. Concretely, PAR archives the lowest time-to-error with a statistically significant difference to CHO (p-value $\approx$ 0.008) and COV (p-value $\approx$ 0.0001). PAR also archives the smallest time-to-error for rhino_0 but without any statistically significant difference to any other filtering technique and also tiny differences overall. Failure rhino_1 is an interesting case. Concretely, this is the only subject where COV outperforms every other technique. Moreover, COV archives statistically significant differences to every other technique. Therefore, we argue that for some failures, COV can be a good choice. Concretely, failures linked to unique generator coverage might benefit from seeds generated by COV. For Failures rhino_3, rhino_4 and rhino_5 we reiterate that these failures archived low reliability, so their analysis would require longer runs for better soundness. Also, for these three failures we report that no statistically

significant differences exist between any of the techniques, which we contribute to the lack of data. However, we note that on rhino_3 the two runs of CHO (i.e. table 3) can find the failure much faster than any other approach. The same applies to rhino_5 and the single run of RND.
Wrapping up our observations, we can't give a clear answer to RQ3. We observe that PAR is a particularly good fit for Closure. Moreover, CHO and RND fair similarly, but for efficiency of failure discover we see small advantage of CHO. COV seems to be a worse choice than the other techniques, especially for Ant. However, we also discover that COV can outperform other approaches if the failures are linked to unique generator coverage.

5.5 Threats to Validity

Internal Validity We try to avoid systematic errors by translating guidelines by Klees et al. [31] to practice. However, we ran shorted experiments than suggested by Klees et al., but argue that within the one-hour time limit we can observe gains in efficiency to support our claims. Additionally, we ran twenty repetitions, which might not be enough. However, we report small deviations in our data, suggesting that the number of runs was enough. Another possible threat to internal validity is the lack of a comparison to other seed corpora generation or minimization techniques. However, to the best of our knowledge, there is no other approach trying to generate seeds that are parameter sequences which are required for Zest. Another possible threat to internal validity could be the choice of the hyperparameters output size and time give to RLCheck. We note that we didn't tune these values, allowing for a fair comparison.

External Validity Generalization of our approach might assemble a threat to external validity as we only use four subjects for our evaluation. However, these four subjects are matured and widely-used real-world programs. Furthermore, they cover two widely used file formats (JavaScript and XML). Therefore, we argue that the subjects form a solid representation of real-world programs.

Construct Validity Our measurement of unique failures might be a threat to construct validity as it heuristically determines uniqueness based on four elements of the stack trace. However, we argue that the method has been proposed in [13] and used to classify unique failures by other researchers.

6 Conclusion and Future Work

In this paper, we presented a novel technique for the quick generation of seed corpora that addresses shortcomings of previous techniques. In particular, we extended the successful greybox fuzzer Zest [45] which uses parameter sequences, instead of inputs to the PUT, as seeds. Our technique quickly generates inputs using the blackbox fuzzer RLCheck [53], transforms these inputs into parameter sequences, and quickly filters these

sequences using information obtained from the generation process of RLCheck. We develop and compare four filtering techniques that exploit different types of information from RLCheck's generators. As a result of this process, we derive a small set of parameter sequences that can be used as a seed corpus for Zest. We evaluated our approach using four widely-used real-world programs that use two different types of file formats. The results of the evaluation suggest that our approach can significantly increase the efficiency of Zest to uncover failures and increase coverage. We argue that the increase of efficiency recommends our novel technique in situations with limit resources for fuzzing.
For future work, we want to investigate how the selection of the hyperparameters output size (i.e. number of seeds emitted by our technique) and time given to RLCheck influence the performance of our approach. Additionally, we want to improve choice sequence filtering by comparing candidates more efficiently using techniques like locality sensitive hashing (proposes in [41]). Moreover, we want to investigate if more fine-grained coverage measures can improve the performance of generator coverage filtering.

References

[1] Asm. `https://asm.ow2.io/index.html`. Accessed: 2022-03-21.

[2] Cert basic fuzzing framework. `https://www.cert.org/vulnerability-analysis/tools/bff.cfm`. Accessed: 2022-07-22.

[3] Cosine similarity. `https://www.sciencedirect.com/topics/computer-science/cosine-similarity`. Accessed: 2022-06-29.

[4] Junit 5. `https://junit.org/junit5/`. Accessed: 2022-07-23.

[5] junit-quickcheck. `https://pholser.github.io/junit-quickcheck/site/1.0/index.html`. Accessed: 2022-03-21.

[6] libfuzzer – a library for coverage-guided fuzz testing. `https://llvm.org/docs/index.html`. Accessed: 2022-03-21.

[7] Anand, S., Păsăreanu, C. S., and Visser, W. Jpf–se: A symbolic execution extension to java pathfinder. In *International conference on tools and algorithms for the construction and analysis of systems* (2007), Springer, pp. 134–138.

[8] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. Nautilus: Fishing for deep bugs with grammars. In *NDSS* (2019).

[9] Avgerinos, T., Rebert, A., Cha, S. K., and Brumley, D. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 1083–1094.

[10] Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR) 51*, 3 (2018), 1–39.

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

[12] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering 45*, 5 (2017), 489–506.

[13] 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.

[14] Cadar, C., Dunbar, D., Engler, D. R., et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[15] Chen, P., and Chen, H. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 711–725.

[16] Chen, P., Liu, J., and Chen, H. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 499–513.

[17] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (2000), pp. 268–279.

[18] Clarke, L. A. A program testing system. In *Proceedings of the 1976 annual conference* (1976), pp. 488–491.

[19] Feldt, R., and Poulding, S. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (2013), IEEE, pp. 350–359.

[20] Fioraldi, A., D'Elia, D. C., and Balzarotti, D. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2829–2846.

[21] Ganesh, V., Leek, T., and Rinard, M. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 474–484.

[22] Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., and Marinov, D. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1* (2010), pp. 225–234.

[23] Godefroid, P., Klarlund, N., and Sen, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 213–223.

[24] Godefroid, P., Peleg, H., and Singh, R. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), IEEE, pp. 50–59.

[25] Herrera, A., Gunadi, H., Hayes, L., Magrath, S., Friedlander, F., Sebastian, M., Norrish, M., and Hosking, A. L. Corpus distillation for effective fuzzing: A comparative evaluation. *arXiv preprint arXiv:1905.13055* (2019).

[26] 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* (2021), pp. 230–243.

[27] Hocevar, S. zzuf. `https://github.com/samhocevar/zzuf`. Accessed: 2022-07-22.

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

[29] Huang, R., Sun, W., Xu, Y., Chen, H., Towey, D., and Xia, X. A survey on adaptive random testing. *IEEE Transactions on Software Engineering 47*, 10 (2019), 2052–2083.

[30] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

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

[32] Kukucka, J., Pina, L., Ammann, P., and Bell, J. Confetti: Amplifying concolic guidance for fuzzers. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE* (2022), vol. 22.

[33] Lampropoulos, L., Hicks, M., and Pierce, B. C. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–29.

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

[35] Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., and Tiu, A. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 627–637.

[36] Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., and Liu, Y. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 533–544.

[37] Löscher, A., and Sagonas, K. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2017), pp. 46–56.

[38] Löscher, A., and Sagonas, K. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)* (2018), IEEE, pp. 70–80.

[39] 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 (2019), 2312–2331.

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

[41] Miranda, B., Cruciani, E., Verdecchia, R., and Bertolino, A. Fast approaches to scalable similarity-based test case prioritization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 222–232.

[42] Nguyen, H. L., and Grunske, L. Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing. *arXiv preprint arXiv:2202.13114* (2022).

[43] Noller, Y., Păsăreanu, C. S., Böhme, M., Sun, Y., Nguyen, H. L., and Grunske, L. Hydiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 1273–1285.

[44] 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* (2019), pp. 398–401.

[45] Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Le Traon, Y. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 329–340.

[46] Padhye, R., Lemieux, C., Sen, K., Simon, L., and Vijayakumar, H. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages 3*, OOPSLA (2019), 1–29.

[47] Pailoor, S., Aday, A., and Jana, S. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 729–743.

[48] Papadakis, M., and Sagonas, K. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang* (2011), pp. 39–50.

[49] Peng, H., Shoshitaishvili, Y., and Payer, M. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 697–710.

[50] Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., and Roychoudhury, A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering 47*, 9 (2019), 1980–1997.

[51] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS* (2017), vol. 17, pp. 1–14.

[52] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 861–875.

[53] Reddy, S., Lemieux, C., Padhye, R., and Sen, K. Quickly generating diverse valid test inputs with reinforcement learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), IEEE, pp. 1410–1421.

[54] Security, M. funfuzz. `https://github.com/MozillaSecurity/funfuzz`. Accessed: 2022-07-22.

[55] Serebryany, K. OSS-Fuzz - google's continuous fuzzing service for open source software. USENIX Association.

[56] Sirer, E. G., and Bershad, B. N. Using production grammars in software testing. *ACM SIGPLAN Notices 35*, 1 (1999), 1–13.

[57] Srivastava, P., and Payer, M. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021), pp. 244–256.

[58] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[59] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction.* MIT press, 2018.

[60] Visser, W., and Geldenhuys, J. Coastal: Combining concolic and fuzzing for java (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2020), Springer, pp. 373–377.

[61] Wang, J., Chen, B., Wei, L., and Liu, Y. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 579–594.

[62] Wang, J., Chen, B., Wei, L., and Liu, Y. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 724–735.

[63] Wang, J., Song, C., and Yin, H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing.

[64] Wang, T., Wei, T., Gu, G., and Zou, W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 497–512.

[65] You, W., Liu, X., Ma, S., Perry, D., Zhang, X., and Liang, B. Slf: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 712–723.

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

[67] Zalewski, M. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`. Accessed: 2022-03-21.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 02.08.2022 [illegible]