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

Generator-based Fuzzing with Input Features

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Roman Kraus
geboren am: 25.11.1993
geboren in: Moskau

Gutachter/innen: Prof. Dr. Lars Grunske
Dr. Marcel Böhme

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

Contents

1. Introduction

1.1. Motivation

Fuzzing is a testing technique where a system under test (SUT) is repeatedly run with generated inputs. The idea is to find bugs by exposing the system to inputs which it might not expect [MHH+21]. One method to generate inputs is to randomly mutate (i.e., change) the bytes of example inputs. For instance, to randomly overwrite the bytes which constitute the characters of a given XML document. Such approaches can be called *mutation-based* [MHH+21] [LZZ18]. Generating inputs in such a way however can make it difficult to trigger functionalities which relate to valid processing. The reason is that randomly mutating the content bytes of an input has a high likelihood of breaking the syntactic or semantic structure of the input [PLS+19c]. Consider for instance XML (e.g., the string `"<a></a>"`) where deleting just a single angled bracket could already invalidate the entire input.

Generator-based fuzzing is a technique which could potentially help with that. This approach employs *generators* to create its inputs. Generators are programs which generate inputs of a certain type [PLS+19c]. This could be e.g., a generator which creates randomized, but always syntactically valid XML documents. Employing generators for the input creation can be beneficial, as the generator could give certain guarantees. For example, that the inputs will always be syntactically valid. This increases the chance of triggering valid functionalities, even if other aspects of the input are randomized.

Recent research suggests that generating inputs with certain qualities can be beneficial for increasing the coverage during fuzz testing [LS18] [CLO18] [HKZ22]. This is because certain functionalities might only be triggered by inputs with specific characteristics. For example, XML inputs which contain specific tags. One open question however is how to realize such a targeted feature generation with generator-based fuzzing. Namely, it is unclear how one could *represent*, *regenerate* and *identify* input features. This is a downside as generating inputs with certain qualities could have additional benefits to potential coverage increases. For example, it could be used for verifying debugging hypotheses by exposing the SUT to inputs with features that are considered suspicious. Moreover, it could be used to test the quality of bug fixes by producing inputs with properties that were earlier crash inducing [KHSZ20].

The goal of this thesis is therefore to develop an approach for targeted feature generation with generator-based fuzzing. We will employ this approach to investigate whether we can *identify* and *re-generate* features which are necessary to reach rarely visited areas of an SUT. This could increase the achieved coverage and could thus improve the effectiveness of fuzzing campaigns [LS18].

To achieve our goal, we will focus on inputs which have a tree-based structure (namely, XML and JavaScript). Our core idea is to represent features as *sub-trees* of an input and to re-generate the features by *splicing* (i.e., inserting) specified sub-trees

into the tree structure of randomly generated inputs. To identify critical features (e.g., ones which trigger a rarely visited SUT area), our approach performs *pattern mining* on the tree structure of grouped inputs. This allows us to identify shared tree-paths of inputs that fulfil a certain target (e.g., hit a certain SUT area).

To implement our approach we will extend the generator-based fuzzer *JQF* [PLS19b] and its flagship algorithm *Zest* [PLS+19c]. The Zest algorithm has been specifically designed to increase the valid coverage achieved during fuzzing. Extending Zest might thus yield a particular chance to increase the achieved valid coverage even more.

1.2. Research Questions

To investigate the viability and effectiveness of our approach we plan to answer the following research questions:

- **RQ1** Can our approach hit rarely visited areas more often and with more varied path traces compared to Zest?
- **RQ2** Can our approach increase the *overall* coverage compared to Zest?
- **RQ3** How effective is our approach with regards to feature learning and re-generation?

1.3. Outline

The remainder of this thesis is structured as follows. In chapter 2 we will discuss the background of our approach. This will include a discussion of Fuzzing in general, the specifics of Zest's generator-based approach and a discussion of pattern mining.

In chapter 3, we will present work which is related to ours. This will include previous fuzzing approaches that employ input features for fuzzing. Furthermore will discuss approaches which attempt to direct a Fuzzer into particular areas of an SUT.

Chapter 4 presents our approach. We will discuss the motivation behind it, show its details and present the implementation.

Finally, we will present the evaluation of our approach in chapter 5 and provide a conclusion and an outlook on further work in chapter 6.

2. Background

In the following chapter we will provide an introduction to foundational topics that are relevant to our project. We will start with a formal explanation of fuzzing and typical categories of modern fuzzers. Afterwards, we will discuss specifics of generator-based fuzzing. In particular, we will focus on the generator-based fuzzer JQF [PLS19b] and its algorithm Zest [PLS+19c]. Subsequently we will discuss pattern mining, especially sequential (and graph-based) pattern mining.

2.1. Fuzzing

2.1.1. Introduction to Fuzzing

Fuzzing can be defined as the process of running an SUT with inputs which can go outside of its expected input space [MHH+21]. In the case of XML this might be running an XML parser with XML documents which are randomly mutated. For example, where the contents of XML tags are randomly mangled. We will use the term "fuzzer" to denote the programs which perform the fuzzing. Our terminology will be based on Manes et al. [MHH+21] who provide reference definitions.

The goal of fuzzing is typically to discover bugs [God20] [BCR21]. To identify whether an execution resulted in a bug, a so-called "bug oracle" is used. In its simplest form this might be checking whether the run resulted in a crash. More intricate methods may e.g., monitor the internal state of SUTs. For example, by using sanitizers to detect memory issues which do not necessarily result in a crash (like e.g., buffer overflows) [MHH+21] [Pay19].

Fuzzing is usually performed iteratively. Even though there exists a broad range of fuzzers, Manes et al. [MHH+21] break the process down to a 7-step algorithm. We will use that algorithm to explain the general steps of fuzzing. The algorithm is depicted in Figure 1.It takes two inputs. First you have a set of configurations, $\mathbb{C}$. A configuration ($c \in \mathbb{C}$) encapsulates the parameters a fuzzer needs to generate a new input and to perform an execution. This includes at least the SUT. Other components might be seed inputs[1] or parameters for the randomization (e.g. where and "how much" to mangle a file). Manes et al. [MHH+21] leave the configuration contents deliberately open as each fuzzer might need different parameters depending on its algorithm.

The second input is the time limit, t_{limit}. This describes the maximum amount of time a fuzzer shall operate on an SUT. The entirety of executions/iterations a fuzzer performs before it stops can be called a "fuzz campaign" [MHH+21]. We will use the terms "fuzz run" or "fuzz execution" to describe a single execution of the SUT with the fuzzer (i.e., one iteration). The result of a fuzzing campaign is a set of discovered bugs, $\mathbb{B}$.

[1] Seed inputs are provided inputs, which are used for bootstrapping.

```
   Input: ℂ, t_limit
   Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅;
2  ℂ ← Preprocess (ℂ);
3  while t_elapsed < t_limit ∧ Continue (ℂ) do
4     conf ← Schedule (ℂ, t_elapsed, t_limit)
5     tcs ← InputGen (conf)
       // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← InputEval (conf, tcs, O_bug);
7     ℂ ← ConfUpdate (ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

Figure 1: The fuzzing loop (taken from [MHH+21])

Now that we know the inputs of the algorithm, let us discuss the processing steps. In the first step, the initial set of configurations is replaced by a new, preprocessed one (line 2)[2]. The preprocessing step is there to do some preparations before the first fuzz run. This can include instrumenting the SUT, preparing driver applications or removing redundant configurations [MHH+21]. Instrumenting means, that additional code is added to the SUT in order to monitor (or control) its operation [BNK16]. In the context of fuzzing this often means code which tracks code coverage during execution.

The next step is the while-condition of the fuzzing loop (line 3). This controls how often the SUT is executed with a new input. The while condition performs two checks. First, it examines if the time limit is met ($t_{elapsed} < t_{limit}$). Secondly, it checks whether there are any custom reasons why the fuzzer might stop. These custom reasons are implemented individually per fuzzer in the **Continue(ℂ)** function. For example, it might check whether the fuzzer traversed all possible paths of the program. If so, it could stop the operation. This can be detected by fuzzers which closely analyze and track the source code of the SUT [MHH+21]

The third step is the selection of the next fuzz configuration (line 4). This selection might be e.g., influenced by which configurations have recently performed well in terms of code coverage or number of found bugs.

In the fourth step, a test case (= input) is generated (line 5). The details of the generation are once again fuzzer dependent. Two typical approaches are to either mutate contents of a given seed file or to randomly generate inputs from a model (e.g. a grammar).

Finally, the test case is used as an input to the SUT (line 6). A given bug oracle (O_{bug}) is used to classify the outcome as valid or invalid. This result is stored in the so called "execution infos" ($execinfos$). These infos can also include additional data, like achieved code coverage.

These gained pieces of information are then used to update the configurations (line

[2]We leave out the initialization of the bug-set (line 1), as it is an obvious step

7). For instance to add, remove or modify certain ones.

The last step is then to update the set of bugs (line 8) and to prepare the next iteration of the fuzzer.

2.1.2. Fuzzer Categorization

There are many dimensions according to which one can distinguish different types of fuzzers. In the following we will focus on two aspects: Source code awareness and input generation.

Black- White- and Greybox-Fuzzer The first major distinction one can do between fuzzers is according to their awareness of the SUT source code. Typically one can identify three different types: black-, white- and greybox fuzzers [MHH+21], [LZZ18]. The source code awareness usually influences the decisions of the fuzzers (typically input generation).

Blackbox fuzzers constitute the most traditional form [MHH+21]. They refer to fuzzers which do not inspect the internals of the SUT at all. Instead, they make their fuzzing decisions only depending on the directly observable output of the SUT [MHH+21]. For example, the processing time or data on the standard-out stream. The internals of the SUT constitute a blackbox for these fuzzers. This approach can be very fast, as we have no instrumentation and likely process fewer information. However it might explore the program less thoroughly, as it has no information on the coverage achieved per input. Thus, it can e.g., not favor inputs which hit rarely visited program areas.

At the other extreme there are whitebox fuzzers. These have full access to the SUT source code and usually perform a thorough analysis of it. For example, they might attempt to identify important values or possibly vulnerable locations in the code [MHH+21]. Whitebox fuzzing is closely related to dynamic symbolic execution (DSE) [MHH+21], [God20]. This is an adaptation of "normal" symbolic execution. In ("normal") symbolic execution, the SUT is executed with abstract, symbolic inputs instead of concrete ones [BCD+18]. This allows one to directly explore each possible program path, as you only have to update abstract constraints on the variables and not provide real values beforehand. Furthermore, one can use obtained constraints to later generate concrete inputs with a constraint solver. DSE augments this procedure such that symbolic execution only keeps track alongside concrete executions. This can have performance benefits, as we run the system with concrete and not abstract values. Furthermore, DSE can also handle calls to libraries for which no source code is available. Normal symbolic execution would usually struggle with that [FR19].

DSE has some relation to fuzzing, as you also repeatedly run a SUT with new inputs to increase code coverage. Nevertheless, not all authors use the term fuzzing when discussing their DSE works [MHH+21]. One reason might be that the program exploration in DSE is more systematic than in traditional fuzzers. Still, it is a noteworthy

relation which is also relevant for this thesis.

Finally, we have greybox-fuzzers. These constitute a mix between blackbox- and whitebox-fuzzers. This means that they employ some instrumentation and code analysis, but usually keep it more constrained compared to whitebox-fuzzing [MHH+21]. The reasoning is to profit from source code awareness, while limiting the overhead. A typical metric collected by greybox-fuzzers is the code-coverage achieved by inputs. Measuring this induces arguably less overhead than e.g., tracking branch conditions and using solvers to generate inputs (as in DSE). Nonetheless, analyzing code-coverage can already be a very powerful metric to better explore certain program areas. Greybox-fuzzing is therefore an approach which is commonly employed in modern fuzzers. One prominent example is the fuzzer American Fuzzy Lop (AFL) [Zal14]. This fuzzer is "coverage guided", because it uses the coverage achieved by inputs to determine which inputs to focus on. Namely, it keeps a queue of inputs which from which it selects further inputs to mutate (see below). Generated inputs are added to the queue if they achieved new coverage [LS18].

Input generation A second important dimension according to which fuzzers can be classified is their input generation method. Currently there are two major categories: Model-Based and mutation-based (or "model-less") input generation [MHH+21], [LZZ18] [Pay19].

Model-Based Input Generation Model-Based input generation means that the fuzzers use a description of the input format to generate their test-cases [MHH+21]. Typical examples include grammars, API-descriptions or implicit models embedded in generators. One example for a grammar based fuzzer is Nautilus [AFH+19]. The fuzzer JQF [PLS19b] on the other hand is an example where the model is embedded implicitly. This fuzzer uses generator programs which have custom routines to create new inputs for the SUT.

Another approach in model-based input generation is not to rely on predefined models, but to instead infer them during execution [MHH+21]. For instance, a fuzzer could perform static or dynamic analysis to identify constants which might be relevant to pass certain constraints [LZZ18]. Furthermore, fuzzers could attempt to infer input grammars based on a given set of valid files. Even though model inference could yield important improvements, it seems that so far not much research has been done in this direction [MHH+21]. Nonetheless, it is a potentially promising approach which could simplify model generation.

Mutation-Based Input Generation The other major technique for input generation is mutation-based. In contrast to the model-based approach, this principle does not use any model of the inputs. Instead, it requires seed inputs. These are well formed inputs which are used for bootstrapping. Fuzzers in this category would take those inputs and randomly mutate parts of them to generate new inputs. For instance, by flipping

certain bits, by exchanging byte blocks or by setting parts to interesting values (e.g. 0 or -1 when mutating integers) [MHH+21]. The fuzzer AFL [Zal14] is a prominent example of this approach.

2.2. Generator-based Fuzzing

Generator-based fuzzing constitutes a sub-category of model-based fuzzing. The idea is to use generator programs which create inputs for an SUT. Thus, inputs are not derived from seed inputs via random content mutation, but are instead generated from the ground up by dedicated routines. The advantage of this approach is that generators could give certain guarantees. For example, that inputs will always be syntactically valid. This increases the likelihood of reaching valid processing stages with fuzz inputs compared to mutation-based approaches. Because mutation-based approaches have a high probability of breaking syntactic or semantic input properties due to their random mutations which ignore the input model.

In the following we will briefly trace the origins of generator-based fuzzing before describing the state of the art. Namely, the fuzzer JQF [PLS19b] and its algorithm Zest [PLS+19c].

2.2.1. Property-based Testing

Property-based testing (PBT) as introduced by Claessen and Hughes [CH00] can be seen as the origin of modern generator-based fuzzing[3]. It is a black-box testing technique where SUT properties are checked by confronting the SUT with randomly created inputs provided by generators. For instance, imagine that you have an SUT which can reverse a list of integers. One property could be that a list should be in its original order if you call the reverse-function twice on it. To test this with PBT, you would specify this property and then let generators create random inputs. For each such test case, the SUT is run and specified properties are checked. If an input violates the property, this might be an indicator for a bug in the SUT [LS17]. Thus, in its most traditional form PBT might be loosely defined as black-box fuzzing with generators and property-based bug oracles.

Popular instances of PBT are the seminal tool QuickCheck [CH00] and its Java implementation `junit-quickcheck` [Hol]. The modern generator-based fuzzer JQF is in fact based on `junit-quickcheck` and also describes its process as property-based testing [PLS19b]. The notion of generator-based fuzzing seems to be a relatively recent one which appears to have become more popular (or possibly even originated) with the advance of JQF and Zest [PLS+19a, Lem21, NG22].

[3]Please note that Claessen and Hughes [CH00] do not use the term "property-based testing" in their paper. However, they are credited as being the pioneers of that field [LS17].

In the following we will use the term PBT rather when speaking of more traditional, black-box, generator-based fuzzers in the lineage of QuickCheck. The term generator-based fuzzing will be used as the more general term and especially when referring to its more modern instantiations in the form of JQF and Zest. Please mind however, that the term PBT and generator-based fuzzing have also been used synonymously by one of the authors of JQF and Zest [Lem21].

2.2.2. JQF

Now that we have presented the origins of generator-based fuzzing, we will discuss the state of the art. Namely, the fuzzer JQF [PLS19b] and its algorithm Zest [PLS+19c].

JQF Motivation JQF is a generator-based greybox-fuzzer which has been implemented in Java. Its core strength is that it augments black-box PBT with coverage information. We have previously seen that it can be difficult for *mutation-based* fuzzers to cover valid functionalities of an SUT. This is because random mutations of the raw input content have a high probability of breaking syntactic or semantic constraints. Let us be more specific with what we mean by that. Many programs which take highly structured inputs (e.g., XML documents) process their inputs in two phases:

1. A syntactic analysis
2. A semantic analysis

The first check ensures that inputs fulfill the general syntactic input constraints. For instance that they are valid XML documents. The second check investigates whether the input also fulfills content constraints specific to the SUT. For example, that a XML file only contains tags which are defined in the SUT's standard and appear in sequences which match the SUT's schema [PLS+19c].

Based on that we make the following definitions: If an input passes the first stage, we will call it *syntactically valid*. If an input passes the first and the second stage, we will call it *semantically valid* (or simply *valid*). Syntactic and semantic *invalidity* refers to failing at *exactly* the corresponding stages. Simple invalidity (i.e., without specifying the stage) may refer to either [PLS+19c].

To make it more concrete, imagine an SUT which can only process XML documents which contain tags with the name "a". The input `"<a></a>"` would thus be syntactically *and* semantically valid because it constitutes syntactically valid XML and because it only contains tags with the name "a". If we would now replace an angled bracket with an underscore, we would destroy the syntactic XML structure of the input. Therefore, the mutated input `"<a_</a>"` would be *syntactically* invalid. If, on the other hand, we would replace both "a"s with "b"s, we would still produce syntactically valid XML. However, remember that our SUT can only process tags with "a" as their name. The input `"<b></b>"` would thus be *syntactically valid* but *semantically invalid*.

Fulfilling syntactic and semantic input requirements can be a difficult task for randomized input generation. Especially, if the inputs are generated by random mutation of content bytes, as is the case for fuzzers like AFL [PLS+19c]. Generator-based testing approaches (like PBT) can have an advantage because generators can e.g., give the *guarantee* of syntactic validity, even if other aspects of the input are randomized (for instance, the number of tags and their content in the case of XML). Nevertheless, black-box PBT approaches can still struggle at generating inputs which are semantically valid *and* which explore varying parts of the valid processing stages. One factor is that they have no information on which SUT branches have been executed by an input [PLS+19c]. Therefore, they could e.g., not attempt to re-generate new inputs based on ones which recently uncovered new areas. This is a downside because such a *coverage-guided* search strategy has proven itself to be quite successful with mutation-based fuzzers and can be considered to be the de facto standard in current fuzzing research [Pay19]. JQF attempts to bridge this gap by augmenting black-box PBT with coverage information. The result is our modern notion of generator-based fuzzing (at least, if we exclude Zest for a moment).

Technical details of JQF JQF extends the PBT framework `junit-quickcheck` [Hol] and instruments SUTs to collect coverage information on executed inputs. However, JQF does not ship with a mandatory, hard-coded fuzz algorithm which processes that information. Instead, it is built by design to be an extensible framework which comes with several fuzz algorithms and which allows for the simple implementation of custom ones. JQF realizes that with a concept it calls Guidances [PLS19b].

A Guidance is a Java class which represents a fuzz algorithm in JQF's framework. JQF itself provides the generic fuzzing loop. It delegates however certain functionalities within that fuzzing loop to a Guidance. In terms of Manes et al. [MHH+21] (see section 2.1.1) the tasks of a Guidance include the scheduling (and updating) of configurations as well as the decision when to stop a fuzzing campaign.

This principle is presesented in the pseudo-code of JQF's fuzzing loop in Listing 1. The loop continues as long as the Guidance can (or chooses to) provide another input (line 1)[4]. If an input is available, JQF obtains an InputStream (i.e., a byte stream) which encodes the input (line 2). That InputStream is then used to instantiate the input objects by calling the generators (line 3). This means that the generic byte stream is converted to objects which e.g., represent XML documents. Finally, the SUT is run with the generated inputs and the Guidance is informed about the outcome of the execution (lines 4-5).

[4]What is not mentioned here is that JQF also considers a configurable max runtime.

```
1       while(guidance.hasInput()) {
2           InputStream inputStream = guidance.getInput();
3           Object[] args = generators.generate(inputStream);
4           Result result = runSUT(args);
5           guidance.handleResult(result);
6       }
```

Listing 1: Pseudo-Code of JQF's Fuzzing Loop

An important aspect is that the InputStreams which are returned by Guidances do *not directly* represent the inputs that are later generated (i.e., in line 3). Instead they constitute the byte stream that is consulted by generators when making random decisions. Since these byte streams control the decisions of the generators, they also decide how the input will look like. The fact that Guidances control the generator decisions (and not directly the content) is a crucial design choice which also plays an integral role for the Zest algorithm. We will thus discuss it more thoroughly, when presenting Zest.

2.2.3. The Zest Algorithm

The Motivation Behind Zest Now that we have discussed the basics of JQF let us move on to its arguably most important Guidance: The Zest algorithm [PLS+19c].

The idea of Zest is to increase the likelihood of generating syntactically and semantically valid inputs while fuzzing. We have previously seen, that many SUTs process inputs in two steps: First, a general syntactic check and second a semantic check. We have also seen, that it is quite difficult to fulfil both constraints. Especially, if you generate inputs via direct content mutation and do not pay attention to the underlying model. You can remediate this in some way if you use generators. Because generators could give certain guarantees. For instance, that all created inputs will be syntactically valid. Such inputs would pass the syntactic checks of the SUT. However, they will again likely *not* meet the semantic constraints, as the contents are fully random (i.e., they ignore specifics of the SUT). One solution could be to develop generators which are attuned to the SUT requirements. But this requires additional effort and knowledge which might be difficult to acquire (e.g., because it is extensive or not well documented). Thus it can be difficult to easily produce inputs, which explore semantic areas of SUTs. Zest improves upon this by *automaticallly* guiding off-the-shelf generators towards syntactic *and* semantic validity [PLS+19c].

Parametric Generators An important concept that Zest builds on is that of *parametric generators*. That is why we will first introduce this, before moving on to Zest itself.

We have already discussed generators in general before. We define them as programs which produce randomized, but syntactically valid inputs of a given type. In order to randomize the contents of their results (e.g., the number of XML tags), generators in

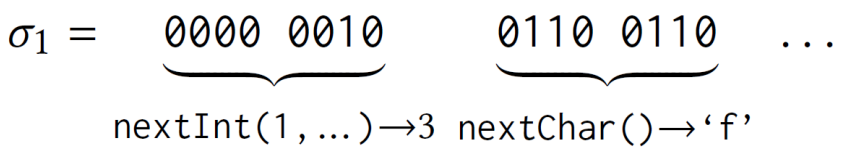


Figure 2: A Parameter-Stream (taken from [PLS+19c])

JQF employ a so-called SourceOfRandomness. This is a (potentially infinite) sequence of pseudo-random bytes. It can be used by randomizer-functions to derive complex, pseudo-random objects. For instance, if a function needs to produce a randomized integer, it might read the next 4 bytes from that stream and interpret that as the integer. Conversely, if a function e.g., provides a random character, it might just read one byte and use that as the character. This idea is illustrated in Figure 2. It shows how a generator calls randomizer-functions and how those functions use the SourceOfRandomness. First, the generator queries for an integer. This might e.g., be for the length of a random string. To obtain this value, the generator calls a nextInt(...) randomizer-function. This function reads one byte from the stream and derives the integer 3 from that pattern[5]. Afterwards, the generator asks for a random character. This might be e.g., the first character of the random string. Again, a randomizer-function reads from the SourceOfRandomness and returns a derived value (this time 'f'). For the sake of simplicity, both randomizer-functions read one byte in this example. In reality, each operation can as read as many bytes as it wants and use them in any way they need.

Padhye et al. [PLS+19c] call each bit of a pseudo-random bitstream a *parameter*. Each parameter is originally *untyped*. It only assumes a type, once it is read and interpreted in some way (e.g., as a component of an integer). A *parametric generator* might therefore be defined as a generator which uses a parameter stream to make (all) its randomized choices.

An important insight is the following: If you mutate a parametric byte stream (e.g., by flipping random bits) this can induce high level changes in the produced output. For example, imagine that we use the stream from Figure 2 to generate a XML tag pair (<random_string></random_string>). Let the obtained integer (3) refer to the length of a randomly generated tag name. Let the obtained character ('f') denote the first character of the string. Let the remaining two characters be an 'o' (without loss of generality). This would produce an output like this: <foo></foo>.

Let us now further assume, that we mutate some bits in the stream of 2. This is illustrated in 3. As you can see, the mutation would change the character "f" to "W". If we would now generate a XML document from that stream, we would obtain the following output: <Woo></Woo>.

Even though we performed a random mutation on the bit-level, we still obtained

[5]If you are wondering, why the function returns a "3" even though there is a "2" in binary, then please mind that the smallest number the function is asked for is a "1".

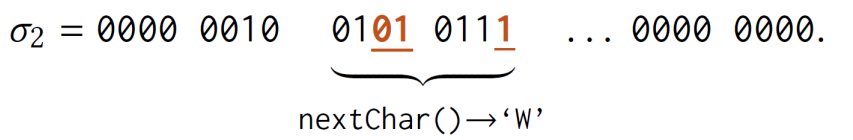


Figure 3: A Mutated Parameter-Stream (taken from [PLS+19c])

syntactically valid XML tags. This is because we affected the *choices* the generator makes and not just randomly flipped bits in the produced content. Our mutations thus caused changes on a conceptual XML level. This is what is meant by high level changes. Even though we performed the mutation at a low level (i.e., change bits in a stream), the impact occurs at a high level as it influences the course of the generator. And since generators always produce syntactically valid outputs, so are the new ones.

This insight is presented in the Zest paper [PLS+19c] and yields tremendous consequences. Because this provides a tool to perform low level mutations while still producing syntactically valid files. It is a key component of Zest's algorithm.

The Zest Algorithm We will now present how Zest works in detail. The core of Zest rests on three principles.

The first principle is that Zest uses generators to create its inputs. This ensures that the inputs should always be syntactically valid[6].

The second principle is that Zest derives new inputs by mutating the SourcesOfRandomness (or "parameter stream") fed to the generators. This guarantees that the new inputs will be again syntactically valid (as discussed before). Because, we are now influencing the high level decisions of the generators (e.g., "produce 5 instead of 3 XML tags") and not just randomly flip bits in the content.

The third principle is that new inputs are always based upon previous SourcesOfRandomness, which have either

- produced *semantically* valid inputs or
- uncovered new program areas.

First, this increases the chances of deriving new semantically valid inputs. Because we are now basing our new inputs on SourcesOfRandomness of previously semantically valid inputs. If we mutate these, the chance of new semantically valid inputs is arguably higher than from random ones. Secondly, we *also* have a higher chance of extending the coverage. Because we also derive new inputs from SourcesOfRandomness which have discovered new program areas. Such inputs are more likely to visit related but yet undiscovered areas, thus potentially increasing the coverage.

[6]Provided that the generator gives such a guarantee. For simplicity's sake we will assume that generators always produce syntactically valid inputs if not stated otherwise.

Initially Zest starts with randomly generated SourcesOfRandomness. However, as soon as a SourceOfRandomness fulfills one of the criteria mentioned above, Zest saves it to a queue and from then on only generates new inputs by mutating SourcesOfRandomness from that queue.

You can thus summarize Zest as a combination of coverage-guided fuzzing with parameter stream mutation. By following this approach, Zest is potentially able to better explore the semantic areas of SUTs compared to more traditional techniques before. Because Zest can concentrate on generating inputs which are semantically valid without risking to lose syntactic validity through mutation. This prediction has been validated by comparing Zest against AFL and junit-quickcheck. The tested input types included (among other things) XML configuration files for build systems (namely Apache Maven [Mav] and Ant [Ant]) as well as JavaScript-code fed to compilers and optimizers (i.e., Mozilla Rhino [Rhi] and Google Closure [Clo]). The JQF generators have been manually developed by the authors in less than two hours each and are kept rather small (all <= 500 LoC) and generic. Zest has been better at exploring semantic code in each benchmark, exercising up to 2.81 times as many semantic branches as the best other technique. Furthermore, Zest did discover semantic bugs faster and more reliably than the other approaches. Therefore Zest appears indeed to be a good method to better explore semantic code areas with fuzzing. It should be noted however that AFL obtained up to 1.6 times more syntactic coverage and also discovered more bugs there. Thus, Zest is not overall better, but still very effective at semantic fuzzing. This is something which Zest's authors mention as well [PLS+19c]. For them, Zest constitutes a technique which targets a different area than more traditional fuzzers like AFL. Namely, Zest focuses on fuzzing semantic processing stages of an SUT. It is thus a complementary approach and no direct competitor.

Determining validity and further Guidances Before we close this subsection we want to briefly discuss two more aspects. Specifically, how JQF/Zest determine semantic validity of an input and what other Guidances are available next to Zest.

First, let us discuss semantic validity. JQF distinguishes between three types of execution results:

1. SUCCESS
2. INVALID
3. FAILURE

The first outcome (SUCCESS) means, that the SUT ran without crashing. Thus, the fuzz run returned normally and no un-catched exception occurred. The second outcome (INVALID) indicates, that an assumption on the input has been violated. Assumptions can be put on inputs to e.g., filter ones out which do not fulfill a pre-condition (e.g., when testing division that the second operator is unequal to zero). The third outcome

(FAILURE) signals that the SUT reported an exception or assertion violation. This case can indicate potential bugs in the SUT [PLS19b].

Semantic validity for Zest means that JQF reports a SUCCESS for a given run. Therefore, it is up to the test- or SUT-developer to ensure that syntactic or semantic invalidity results in an error which can be picked up by JQF [PLS+19c].

In addition to Zest there also exist other Guidances. For example, the Zest paper mentions the AFLGuidance which allows to employ the popular fuzzer AFL for the input generation with JQF [PLS+19c]. Furthermore one could mention RLCheck which proposes a black-box technique for valid input generation with reinforcement learning [RLPS20]. However, we consider it valid to say that Zest is JQF's flagship (and default) Guidance.

2.3. Pattern Mining

In the next subsection we will discuss concepts of pattern mining which are related to and employed in this thesis. Particularly, we will focus on *sequential pattern mining* and *graph pattern mining*. Both topics are relevant to us, because we employed them to identify patterns within inputs that hit a targeted region. Therefore, a certain background is necessary to understand e.g., chosen parameters and implementation aspects.

2.3.1. Sequential Pattern Mining

Pattern mining may be defined as the process of identifying interesting, useful or unexpected patterns in a set of data [FVLK+17]. There exist several forms of pattern mining, depending on the type of input data and the restrictions that discovered patterns must fulfil. For instance, in *frequent itemset mining* the goal is to identify recurring patterns in a list of (unordered) sets of items. An adaptation of that is *association rule mining* which tries to identify implications between itemsets [Han12]. Sequential pattern mining is one particular type of pattern mining.

Sequential pattern mining attempts to find interesting subsequences in a given list of sequences. A sequence is an *ordered* list of itemsets ($s = \langle I_1, I_2, ..., I_n \rangle$). An itemset is a set of symbols (e.g., $\{a, b, c\}$). A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is said to be *contained* in another sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$ if each itemset of s_a is contained in an itemset of s_b such that the order of containing itemsets reflects the original order in s_a. More formally, the sequence s_a is contained in the sequence s_b if there exist integers $1 \leq i_1 < i_2 ... < i_n \leq m$ such that $A_1 \subseteq B_{i1}$, $A_2 \subseteq B_{i2}$, ..., $A_n \subseteq B_{in}$. If a sequence s_a is contained in a sequence s_b we say that s_a is a *subsequence* of s_b. It can be written as $s_a \sqsubseteq s_b$ [FVLK+17, Agg15].

For example, the sequence $s_1 = \langle \{b\}, \{f, g\} \rangle$ is contained in the sequence $s_2 = \langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$, because the first itemset of s_1 is contained in the first

itemset of s_2 and the second itemset of s_1 is contained in the third itemset of s_2. The integers would thus be $i_1 = 1$ and $i_2 = 3$, which is fine because $i_1 < i_2$ holds. On the other hand, the itemset $s_3 = \langle\{b\}, \{g\}, \{f\}\rangle$ is not contained in s_2. The first two itemsets of s_3 could mapped to the first and thid itemset of s_2 respectively. However, for the third itemset of s_3, there is no containing itemset in s_2 except for its third one. However, we can not re-use the third itemset of s_2 because it is already mapped to the second itemset of s_3. Otherwise we would violate the strict ordering which is required by the subsequence-relationship. [FVLK+17].

The support measure Sequential pattern mining attempts to find *interesting* patterns. There are different measures which could be used to define interestingness. One common metric is the *support measure*. Let $SDB = \langle s_1, s_2, ..., s_p\rangle$ be a list (or "database") of sequences. The support of a sequence s_p is the number of sequences of SDB which contain s_p as a subsequence. More formally: $sup(s_a) = |\{s : s_a \sqsubseteq s \wedge s \in SDB\}|$[7]. Counting the number of sequences which contain the subsequence is called the *absolute support*. Alternatively, you might analyze the fraction of sequences which contain the subsequence. This is called the *relative support*, defined as $relSup(s_a) = \frac{sup(s_a)}{|SBD|}$ [FVLK+17, Agg15].

Sequential pattern mining can now be defined as the process of identifying all *frequent subsequences* in a sequence database (i.e., a list of sequences). A sequence s is called *frequent* if $sup(s) \geq minsup$. The value *minsup* is a user-defined *minimum support threshold* [FVLK+17]. Again, the minimum support threshold could be absolute or relative (which would correspondingly affect our definition) [Agg15].

For instance, imagine that we have a relative support value of 0.5 and mine for patterns in the database presented at Table 1. In this case the sequence $\langle\{a\}, \{b\}\rangle$ would be a frequent pattern as it appears in at least half of the entries. However, if we increase the relative support value to 1.0 then only $\langle\{a\}\rangle$ would be returned as pattern. This is because this is the only sequence which appears as a subsequence in *all* entries.

Closed and maximal patterns Closed and maximal patterns are frequent sequential patterns with specific properties. Focusing on these can help reduce the set of patterns that is produced as a result.

Closed patterns are frequent patterns which are not contained in any other frequent pattern that has the same support [FVLK+17, MdCH18]. Let FS describe the set of frequent patterns for a sequence database. Then you can define the set of closed sequences as $CS = \{s_a : s_a \in FS \wedge \nexists s_b \in FS$ such that $s_a \sqsubset s_b \wedge sup(s_a) = sup(s_b)\}$. The notation $s_a \sqsubset s_b$ means that s_a is a subsequence of s_b but that is not identical

[7]Please note that Fournier-Viger et al. [FVLK+17] write $s \sqsubseteq s_a$. We consider this to be a mistake.

to s_b[8]. Focusing on closed patterns can be beneficial because they are *lossless*. This means that you can reconstruct all other frequent patterns (and their support) from this set because they would be subsequences of the closed patterns. Therefore you can get a more compact result list without losing any information [FVLK$^+$17].

To illustrate this, consider again the sequence database given in Table 1. If we set an absolute *minsup* of 2, then we get the frequent patterns which are listed in Table 2. You can see that the pattern $\langle\{a\}\rangle$ is closed because there is no other frequent pattern which contains $\langle\{a\}\rangle$ and which has a support of 3. Particularly, it does not matter that the pattern $\langle\{a\}\rangle$ is included in the pattern $\langle\{a\},\{b\}\rangle$ because they have different support values (namely, 3 versus 2). On the other hand, the pattern $\langle\{b\}\rangle$ is frequent but not closed. This is because this pattern is included in the pattern $\langle\{a\},\{b\}\rangle$ which has the same support of 2.

SID	Sequence
1	$\langle\{a\}\rangle$
2	$\langle\{a\},\{b\}\rangle$
3	$\langle\{a\},\{b,c\}\rangle$

Table 1: A sequence database

Frequent pattern	Support	Closed?	Maximal?
$\langle\{a\}\rangle$	3	yes	no
$\langle\{b\}\rangle$	2	no	no
$\langle\{a\},\{b\}\rangle$	2	yes	yes

Table 2: Frequent patterns of Table 1 (absolute $minsup = 2$)

A related concept is that of maximal patterns. A maximal pattern is a frequent pattern which is not included in any other frequent pattern [FVLK$^+$17, MdCH18]. Let MS describe the set of maximal patterns, then you could define it as $MS = \{s_a : s_a \in FS \ \wedge \ \nexists s_b \in FS$ such that $s_a \sqsubset s_b\}$. You can see that its definition is almost identical to the one of closed patterns, except for the missing support criterion. Nevertheless, both sets are not necessarily the same. For instance, in Table 2 the pattern $\langle\{a\}\rangle$ is closed but not maximal because it is contained in the pattern $\langle\{a\},\{b\}\rangle$. Maximal patterns can also be used to reconstruct all underlying patterns. However, you can not reconstruct the support values for the embedded patterns. For instance, if we would reduce Table 2 to the entry $\langle\{a\},\{b\}\rangle$ then we could conclude that $\langle\{a\}\rangle$ must also be a frequent pattern. However, we could not obtain from that entry that its support is 3 (and not 2). Therefore maximal patterns are not (fully) lossless [FVLK$^+$17].

[8]Fournier-Viger et al. [FVLK$^+$17] are not as explicit when introducing this notation, but we deduce its meaning from the context

Gap constraints We have seen in our definition of the subsequence relationship that the containing itemsets do not have to be contiguous. Therefore, the sequence $s_a = \langle\{a\}, \{c\}\rangle$ is contained in the sequence $s_b = \langle\{a\}, \{b\}, \{c\}\rangle$, even though the itemsets $\{a\}$ and $\{c\}$ do not directly follow one another in s_b. There could be however instances where you want that containing itemsets only have a maximum distance between one another. For instance, imagine that these sequences describe the nodes of two tree paths. Furthermore imagine that you want to find shared subpaths between those paths. In this case you would require that the components of a pattern appear strictly contiguous in the original inputs. Otherwise they patterns would not necessarily describe contiguous sequences in the underlying paths.

Such requirements can be specified with a so-called *maxgap* constraint [Agg15]. For instance, if we require that $maxgap = 0$, then s_a would *not* be a valid subsequence of s_b, because the gap between $\{a\}$ and $\{c\}$ is 1. On the other hand, it would become a valid subsequence if we could set $maxgap = 1$[9]. Provided that the algorithm and its implementation support such constraints you could thus control the maximum distance between containing itemsets. The *mingap* constraint would be the analogous way to require a minimum distance between containing itemsets [Agg15].

2.3.2. Graph-based Pattern Mining

Finally, we also want to briefly discuss graph-based pattern mining. We will focus on mining frequent subgraphs from a set of (static) graphs as this is also the approach that we experimented with. Please mind that there also exist other graph mining approaches which e.g., try to find interesting patterns in individual graphs (i.e., not across a list of graphs) or in graphs which change over time [FVHC+20].

Graph definition A graph may be defined as a tuple $G = (V, E)$ where V is a vertex set and E is an edge set. A vertex $v \in V$ constitutes a "node" in a graph. The set of edges E defines the connections betweens vertices. A graph can be directed or undirected. If the graph is directed, then each edge is an *ordered* pair of vertices [FVHC+20, Kan19]. An edge $e = (v_1, v_2)$ would then mean that there is a direct connection from vertex v_1 to vertex v_2 (with $v_1, v_2 \in V$ and $e \in E$). A reverse connection from v_2 to v_1 would be represented with an individual edge $(v_2, v_1) \in E$. For directed graphs, the set of edges thus constitutes a subset of ordered vertex pairs (i.e., $E \subseteq V \times V$) [FVHC+20]. If the graph is undirected, then edges do not carry an ordering[FVHC+20, Kan19]. An edge could thus be represented by a set $\{v_1, v_2\}$ instead of a tuple [FVHC+20].

The vertices and edges of a graph may also have labels. These can provide additional information on these elements. A label could e.g., be a number or character string which represents the type of a vertex. Our original graph definition would then be extended by functions which map vertices and edges to their corresponding labels

[9]Please note that we use the *maxgap* definition as employed by Aggarwal [Agg15] where no gap between itemsets means $maxgap = 0$. In contrast, in the SPMF [FVLG+16] implementation of the CM-SPAM [FVGCT14] algorithm no gap between itemsets would be equal to $maxgap = 1$ [FV22]. SPMF is a pattern mining framework.

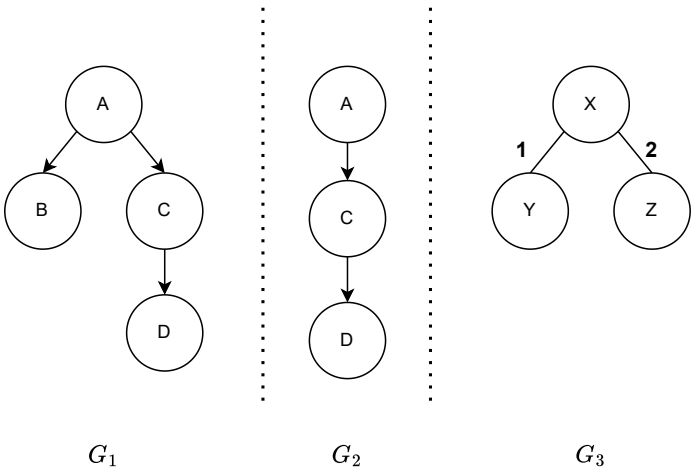


Figure 4: Examples of various graphs

[FVHC+20].

We have provided visual representations of example graphs in Figure 4. The graphs G_1 and G_2 represent directed graphs. The characters in the middle of the circles represent vertex labels. The graph G_3 constitutes an undirected graph which has vertex labels *and* edge labels. You can see that vertices can be nicely represented as circles and that their edges can be depicted by lines connecting these circles. Directed edges can be highlighted with an arrow [FVHC+20].

Frequent subgraph mining Similar to the search for frequent patterns in sequences, one can attempt to find recurring patterns in a list of graphs. More precisely, one can attempt to identify frequently shared *subgraphs* [FVHC+20, Kan19]. A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ [Kan19]. The graph G' thus constitutes a substructure of graph G. For instance, in Figure 4, the graph G_2 could be seen as a subgraph of graph G_1. Given a list (or "database") of graphs $GDB = \{G_1, G_2, ..., G_n\}$ one can attempt to find subgraphs which appear in a minimum amount of graphs of GDB. This process can be called *frequent subgraph mining*[10] [FVHC+20]. Again, the *support* of a subgraph G' means how many graphs of a graph database GDB contain G'. Furthermore, the user-defined *minsup* threshold again defines how many graphs of a database must *at least* contain G' so that G' is categorized as "frequent" and returned as a result [FVHC+20, Kan19].

Searching for frequent subgraphs instead of e.g., sequential patterns can be beneficial if your base data can be represented with a graph structure (as is e.g., the case for XML). This is because the returned graph patterns could encode structural properties

[10]Our definition of frequent subgraph mining is a bit simplified compared to Fournier-Viger et al. [FVHC+20]. They define it using subgraph ismorphisms. We omit that so that we do not have to introduce more graph terminology which is not necessary to understand the core idea.

which might get lost (or could be difficult to reconstruct) with other forms of mining. We became aware of subgraph mining only some time into the work on this thesis. Therefore, we experimented with its integration but eventually dropped it due to time constraints. We will come back to the potential of graph-based mining when discussing our approach in more detail.

3. Related Work

In the following, we will present research which is related to our approach and which was partially highly influential for its design. We will begin by discussing fuzzing approaches which incorporate input feature learning to achieve coverage improvements. Afterwards we will review approaches which attempt to target specific SUT areas.

3.1. Fuzzing with Input Features

The upcoming section discusses selected approaches which incorporate input feature learning or targeted feature generation to achieve coverage improvements during fuzzing. Especially the first two approaches (i.e., FairFuzz [LS18] and TGCT [CLO18]) have been highly influential for the design of our technique. But before we begin their discussion, let us clarify what we mean by "input features".

Input Feature Definition We define an input feature to be a quality of an input which can be directly observed on an input *without* having to first run it on an SUT. This could refer to quite primitive qualities, but also more complex ones. For instance, the existence of the substring `"a"` in the XML tag pair `"<a></a>"` could be counted as an input feature. However, features might also be more abstract and could be placed on the model level. For example, a more abstract feature could be that an XML document must include a pair of embedded tags (without being more specific). Therefore, the term "input feature" is rather broad. However, our definition excludes any qualities which can only be ascertained after the input has been run on an SUT. This means that qualities like e.g., input validity or SUT coverage do *not* meet our definition of input features even though they might be counted as "features of an input" in other contexts.

3.1.1. FairFuzz

Background of FairFuzz FairFuzz [LS18] is an adaptation of the fuzzer AFL [Zal14]. Its main objective is to improve the exploration of code which is only rarely visited by normal AFL.

We have mentioned in section 2.1 that AFL generates inputs by *directly* mutating the contents of previous inputs. Especially important is that AFL's mutations can change *any* part of an input without considering its underlying structure. This creates a great risk of generating inputs which are syntactically and semantically invalid. Accordingly, the authors of FairFuzz found that AFL often struggles to cover important areas of SUTs. They illustrate this on xmllint (an XML parser). When fuzzing this SUT, the authors discovered that AFL managed to produce the input <!ATTLIST BD. This input is very similar an input which guards new code, namley <!ATTLIST ID. AFL would only need to modify one character to generate the necessary input for uncovering new code (i.e. B to I). However, since AFL is as likely to mutate *any* of the characters,

quick success is rather improbable. Quite on the contrary. In this example *none* of the other characters should be mutated as their current form is a precondition to even come close to the relevant code. A random mutation would thus likely lead away from the currently interesting part.

The FairFuzz Algorithm The idea of FairFuzz is to find out which parts of an input should (or should not) be mutated to increase the probability of new code coverage. FairFuzz achieves this by employing three major ideas:

1. By focusing the fuzzing on specific program areas
2. By using a *mutation mask* which defines which bytes of an input are allowed to be mutated (and how).
3. By using *mutation experiments* to heuristically calculate this mask.

FairFuzz's Mutation Mask The mutation mask is an object which is individually calculated per input. It returns for each byte of an input whether FairFuzz allows it to be mutated (and how). The mutation mask only allows the mutation of a certain byte if mutating the byte would produce an input which likely again reaches a currently targeted program area. FairFuzz distinguishes three mutation types:

1. Overwriting a byte with a different value
2. Inserting new bytes at a given position
3. Deleting bytes starting at a given position

When generating a new input, FairFuzz uses the mutation mask for that input to decide which mutations it can apply where. The mutations are still random, however they are restricted to what the mutation mask allows. FairFuzz's main goal is to block input parts from mutation which should be kept intact to reach currently targeted program areas. Therefore FairFuzz *only* generates inputs from ones which have hit currently targeted areas. In addition to that, it calculates their mutation mask to know which parts it can mutate. By this, FairFuzz attempts to increase the coverage in targeted regions.

In order to calculate the mutation mask of an input, FairFuzz iterates over each byte of the input. It then mutates the byte and checks whether the input still reaches a desired program location. If so, FairFuzz allows the applied mutation type for this byte (e.g., an overwrite). Otherwise, it forbids it. This is done once for each byte and mutation type.

Please mind that this procedure still only calculates a *heuristic* mutation mask. Heuristic means that FairFuzz does *not* perform every possible mutation for each byte (e.g., every possible overwrite). Instead it only does *one* mutation per byte and

mutation type (thus, 3 per byte). FairFuzz does so for performance reasons, as it would be too exhaustive to explore each mutation possibility. Furthermore, performing every possible mutation would defeat the purpose of the idea as you would have already generated every possible mutant. Thus, you would have no need for further mutations.

Rare Branches What has been kept vague so far is what program regions FairFuzz targets. This is represented by a concept which FairFuzz calls *rare branches*.

A rare branch is defined as a code branch which has been visited fewer times than a certain threshold (over all runs). This threshold is called *rarity cutoff*. Any branch which fulfills $1 \geq$ visits $\leq$ *rarity cutoff* is considered a rare branch. FairFuzz only mutates inputs which hit reach rare branches. The aim is to thus increase the coverage in areas which so far have been barely explored.

To increase the likelihood of passing constraints imposed by rare branches, FairFuzz calculates the mutation mask for corresponding inputs. This limits the mutation to those input parts which can be changed while still (likely) hitting the rare branch.

The value for the rarity cutoff is determined dynamically per fuzz campaign. This means that it is *dynamically* obtained and not set beforehand. That approach has advantages to manually setting a constant cutoff value. It also has advantages to techniques which do not use cutoff values to determine rarity. First, if you manually set the rarity cutoff value, then this value has to be updated individually for each SUT and might not well reflect "rarity" in different fuzz campaigns of the same SUT (e.g., sometimes 100 visits could be "rare", while other times 100,000). On the other hand, you might consider not using cutoff values at all. Instead you could e.g., simply define the n least visited branches as "rare". However, this could also lead to a high variability regarding what number of hits constitutes rarity.

FairFuzz therefore monitors the execution and defines the rarity cutoff as the smallest 2^i greater than the smallest number of branch visits > 0. More formally:

$$visits_{min} := min(\{visits(b) \mid b \in Branches \wedge visits(b) > 0\})$$
$$rarity\_cutoff := 2^i \text{ so that } 2^{i-1} < visits_{min} \leq 2^i$$

By doing this, FairFuzz creates a dynamic rarity threshold which is automatically attuned per fuzz campaign.

FairFuzz Evaluation Fairfuzz has been compared to other versions of AFL on nine benchmarks (including e.g., xmllint, tcpdump, readpng). The results yield that FairFuzz achieved the highest coverage compared to other approaches in most SUTs (namely 8/9) and that its coverage increase was always the most rapid one. Furthermore, FairFuzz's coverage advantage was statistically significant in two cases. These results indicate that the approach is indeed viable and that it can lead to better code exploration.

3.1.2. Template-Guided Concolic Testing

Next, we will discuss another approach which also has been influential for this thesis. Namely, Template-Guided Concolic Testing (TGCT)[CLO18].

Background and Idea of TGCT

Dynamic Symbolic Execution and Path Explosion TGCT is a technique for dynamic symbolic execution (DSE). It has been developed to improve upon some common performance issues, especially the problem of so-called *path explosion*. We have already seen in section 2.1.2 that DSE can be referred to as "Whitebox Fuzzing". So, there is a certain relation between TGCT and fuzz testing. Nonetheless, please mind, that the authors do not refer to their approach as fuzzing.

Before we will discuss TGCT in depth, let us revise some key points about DSE. In DSE, a SUT is executed concurrently in two ways: First, there is a normal execution which provides the SUT with concrete inputs and runs it. In parallel, the DSE engine maintains a *symbolic* version of each input variable. This is an abstract version of those variables. During execution, the DSE engine keeps track of all conditions laid upon the symbolic input variables (e.g., $x > 0$ for a certain branch). This yields a formula, which describes the necessary conditions to follow this execution path. The DSE engine then negates one of the conditions of the formula. That is equivalent to following a different branch on the execution path. This formula would be then fed to a solver to obtain *concrete* inputs which fulfill these conditions. The SUT would be then run with those inputs and the whole procedure would repeat [BCD+18].

By following this approach, DSE is able to potentially explore all possible paths of a program. Because it can theoretically negate each condition it discovers and thus investigate each possible alternative. However, one common issue is that of *path explosion* [BCD+18]. This refers to the fact, that the number of paths can easily grow exponentially in DSE. Consider for example while loops. If a variable in its condition is symbolically tracked, each loop execution results in two paths: One for the *true*- and one for the *false*-case. Each of the *true*-paths would in turn lead to two new paths. So, if you were to continue this, you can easily see that the number of paths would grow exponentially to the base of two. This problem can become even more troublesome if you e.g., consider nested loops. A similar situation can e.g., occur with recursions.

Path explosion has at least two negative impacts. First, the growing number of paths can cause high memory demands. This could e.g., cause resource competition with other components (for instance, the solver). Thus, it could reduce the performance. Secondly, the DSE engine might "get stuck" on mostly exploring paths which result from the same loops. This could limit the exploration of the overall SUT.

TGCT Template-Guided Concolic Testing [CLO18] has been developed to improve code coverage in DSE, even in the face of path explosion. It does not directly prevent path explosion, however it attempts to enhance the performance by only tracking *some*

variables symbolically. This could e.g., lead to fewer loop conditions being collected (if the conditions no longer involve symbolic variables). Therefore it can reduce the potential for exponential growth.

The variables which are no longer symbolically tracked are set to *concrete constants*. These constants are mined from previous inputs which *improved code coverage*. The intuition is that there might be shared patterns in those inputs which can be beneficial for further code exploration. Therefore we have a dual approach in TGCT: First, you limit symbolic tracking to reduce path explosion. Second, you use values of previously effective inputs to derive new inputs. This second aspect is somewhat reminiscent of FairFuzz. However, we will see that the constant values are obtained quite differently.

TGCT proceeds in a four step loop. First it performs normal DSE for some runs. This is meant to obtain a base set of inputs which have been effective for uncovering new code.

Secondly, these inputs are analyzed for recurring patterns (e.g., common sequences of characters). These patterns are ranked according to their frequency. Furthermore, they are ranked according to whether they contain "good" or "bad" substrings. The "good" substrings belong to patterns which previously performed well. The "bad" ones failed to do so. Initially both sets are empty. They are only initialized after the first TGCT loop.

From the ranked patterns, only a fixed number of top-patterns is considered in the third step. Here, each selected pattern is converted into a so-called *template*. The difference is the following: Patterns are only free-floating sequences. In a template however, each pattern component is given a *concrete* position in the input. The positions are calculated individually per pattern component, according to where they occurred most often in the original inputs. Each input position which is not set to a concrete value is left vague, i.e. it is tracked symbolically.

In the fourth step, the templates are then used to perform DSE. As just explained, patterns define which input variables should be symbolically tracked and which are fixed to constants. TGCT analyzes whether employing the templates uncovered new areas and updates the "good" and "bad" patterns sets from step two accordingly.

As you can see, TGCT has some resemblance to FairFuzz as it also tries to fix aspects of inputs to constants which could increase coverage. Both approaches attempt to learn those aspects dynamically. However, while FairFuzz calculates a mutation mask, TGCT employs pattern mining. Furthermore, while FairFuzz specifically targets rare branches, TGCT attempts to increase coverage more generally. So, there are certain similarities, but also differences. The key point is that both techniques learn characteristics (or "features") of inputs that could be important for future generations. And this aspect has been very influential for this thesis.

The authors of TGCT evaluated their approach on 5 C-programs, comparing it against other DSE search heuristics (e.g., Random branch search). TGCT achieved to cover more branches than the other approaches in all benchmarks. Furthermore,

TGCT discovered 3 unique bugs not found by any other technique. Therefore, the current results suggest that TGCT is a viable approach which can indeed help improve the coverage.

3.1.3. K-Paths

Finally, we want to discuss an approach which is quite similar to ours, namely the work on grammar-based fuzzing with *k-paths* by Havrikov et al. [HKZ22]. Their idea is also to investigate the impact of characteristic subtrees in the model of an input on the achieved coverage during fuzzing. However, different to our approach, they produce inputs by employing *grammars* and not input-specific generators. Therefore, their technique belongs into the domain of grammar-based fuzzing.

Grammar-based Fuzzing Grammar-based fuzzing is a form of model-based fuzzing where inputs are generated by deriving words from a given grammar. Specifically, Havrikov et al. [HKZ22] focus on context-free grammars. We assume that the reader is familiar with the concept of a context-free grammar. For completeness, we quickly revise it here.

A context-free grammar "G" is a 4-tuple $G = (V, T, P, S)$, where

- V is a finite set of non-terminals (also called variables)
- T is a finite set of terminals
- P is a finite set of production rules
- S is the start symbol ($S \in V$)

Non-terminals are symbols which can be replaced (or "expanded") by applying a production rule. Production rules take the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$. Therefore, a production rule $A \to \alpha$ defines that a non-terminal A can be replaced with the (possibly empty) concatenation α which can contain non-terminals and terminals. Terminals can not be expanded. They constitute the final characters that we want to produce [SN07]. Deriving a word means that we begin with the start symbol and apply production rules until only a string of terminals remains.

Grammars can be powerful tools to formally describe the structure of inputs for a given SUT. For example, Havrikov et al. [HKZ22] employ grammars to generate JSON, URL and JavaScript inputs (among other things). Provided that these grammars correctly describe the syntactic structure, grammar-based fuzzing can increase the likelihood of reaching semantic stages of an SUT compared to mutation-based input generation.

K-Paths The derivation structure of a word can be represented by a tree. Consider for example the following (simplified representation) of a grammar[11]:

Term → Addition | Multiplication | Number
Addition → Term " + " Term
Multiplication → Term " ∗ " Term
Number → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

The word "1 + 2" could be derived from the start symbol Term as is depicted in Figure 5. You can see that non-terminals can be represented as inner nodes of the tree, while terminal symbols constitute leaf-nodes. Expansions of individual non-terminals are represented by edges.

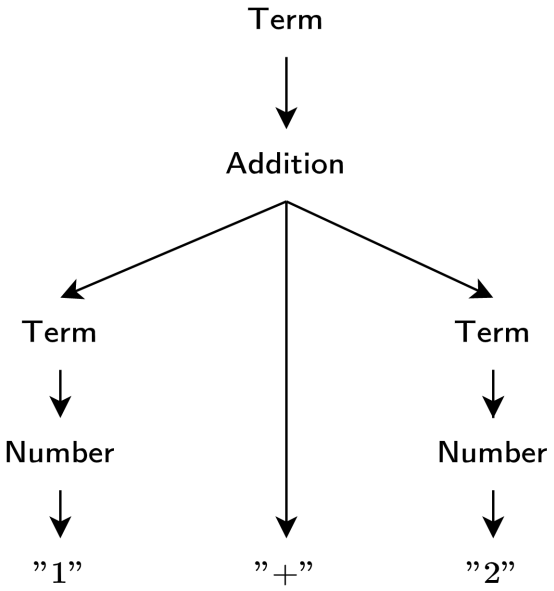


Figure 5: Derivation tree for the word "1+2"

An important observation is that the derivation tree consists of several paths which lead from the root to every leaf. For example, the path "Term → Addition → Term → Number → 1". This path has length 5, because it is a sequence of 5 derivation steps. Furthermore, each derivation tree can be analyzed in terms of its *sub-paths* of a fixed length. This leads to a notion of a k-path. A k-path is a derivation (sub-) path of exactly length "k" [HKZ22]. For example, some of the 2-paths contained in the

[11] We define all symbols which appear on the left side of a production to be a non-terminal. The pipe symbol (i.e., "|") separates different productions with the same non-terminal on the left side. I.e., $A \to \alpha \mid \beta$ represents $A \to \alpha$ and $A \to \beta$. Any character enclosed in double quotes represents a terminal.

derivation tree of Figure 5 are: "Number $\rightarrow$ 1", "Addition $\rightarrow$ Term" and "Addition $\rightarrow$ +".

A grammar could be also represented by a tree, similar to a derivation tree. Furthermore, you could also *enumerate* all unique k-paths that can be derived from a grammar for a given k. This information could be used to analyze a given set of inputs. A set of inputs which contains many of the possible k-paths is said to have high *k-path coverage*. The k-path coverage can be represented by a number within $[0, 1]$ and can be calculated as follows for a given set of inputs:

$$pathcov_k(S) = \frac{\left| \bigcup_{d \in D} paths_k(d) \right|}{|paths_k(G)|}$$

Where S is the set of inputs, D is the set of derivation trees for the inputs and G is the grammar. The operation $paths_k(d)$ returns the unique k-paths contained in a derivation tree d. Analogously, $paths_k(G)$ returns all unique k-paths of a Grammar G [HKZ22].

Intuitively, you would expect that a set of inputs with high k-path coverage should also achieve higher SUT coverage compared to inputs with low k-path coverage. This is because certain functionalities of an SUT could only be triggered by specific input sequences which are represented by specific k-paths. Therefore it could be beneficial to strive for high k-path coverage.

This is one of the aspects that Havrikov et al. [HKZ22] analyzed. They have developed an algorithm to enumerate all k-paths of a grammar (for a given k) and to systematically produce inputs which contain those k-paths. Furthermore, they compared this input generation approach to a purely random (but still graph-based) one. The input grammars that they tested included JSON, CSV, URL, Markdown (and later) JavaScript.

Havrikov et al. [HKZ22] have made several interesting observations. First of all, they have found that a systematic k-path generation resulted in significantly higher code coverage compared to random input generation. This suggests two things. First, that random input generation can have difficulties to explore a grammar as thoroughly as a more systematic approach (in a given frame of generated inputs). Secondly, that covering many k-paths of a grammar can indeed be beneficial for the SUT coverage. Particularly, Havrikov et al. [HKZ22] have found that k-path coverage strongly correlated with code coverage in their evaluation. Therefore, one can conclude that the level of k-path coverage can be a predictor for the achieved SUT coverage. Finally, they have used a decision-tree based learner[12] to learn *associations* between k-paths and specific code locations. The resulting decision trees were then used to produce inputs which target certain methods. Again Havrikov et al. [HKZ22] have found that their approach was quite effective, especially when compared to a random generation. On most SUTs

[12]Namely, an adapted version of Alhazen [KHSZ20].

they managed to cover more than 90% of the intended methods with at least one input[13].

These results suggest that a feature-guided input generation approach can be beneficial in several ways. First, that a systematic production of input features can lead to increased code coverage compared to random input generation. Second, that there can be associations between input features and code locations which could be leveraged (if successfully learned) to target those locations.

Both of these conclusions have strengthened our idea that targeted feature generation could be employed to achieve coverage gains compared to random input generation. However, we only became aware of this particular research some time after work on this thesis began. It therefore was not as influential in the design of our approach as the previously discussed techniques (i.e., FairFuzz and TGCT). Nonetheless, it provides (further) evidence for the potential of considering input features when fuzzing.

3.2. Directed Fuzzing

Directed Fuzzing is a technique where the goal is to automatically guide the fuzzer to *user-defined* areas of an SUT [BPNR17, CXL+18]. It is related to our approach because we also want to target certain code areas. However, our approach should not be classified as directed fuzzing because our targeted areas are *not* user-defined. Instead, our targeted areas are automatically selected with the hope to increase the *overall* coverage through a temporary concentration on (potentially) under-explored SUT regions. Thus our approach should rather be termed *coverage-guided*, even though we also target specific areas of an SUT.

AFLGo One of the first approaches (if not *the* first) of directed greybox fuzzing is the fuzzer *AFLGo* by Böhme et al. [BPNR17]. You may recall that AFL is a mutation-based fuzzer which employs coverage information to prioritize inputs for further mutation. Böhme et al. [BPNR17] extended AFL so that it is also able to explicitly target user-defined areas of an SUT. This is realized by a combination of two things. First, AFLGo performs a distance analysis at compile time. This analysis provides for each basic block[14] a value describing the distance to the target locations. AFLGo can then use this information to assign a distance value to an input, based on the basic blocks the input has visited. That distance value can then be employed to prioritize inputs which are "closer" to target locations. This constitutes the second major component of AFLGo's approach. More precisely, AFLGo gradually shifts from a rather explorative input generation to a more focussed one. In the more focussed stages inputs are prioritized which have a low target distance. Prioritization means here that more new inputs are generated from low distance inputs compared to other

[13] The coverage was not achieved by an individual input but as seen over all inputs.

[14] A basic block is a maximal set of instructions which can be executed in sequence without a branch [CT12].

ones. The distance values that AFLGo employs are based on analyses it performs on the call- and control-flow-graph(s) of an SUT[15].

The authors of AFLGo have found that their approach can reach targeted locations between 1.5 up to 11 times faster compared to baseline AFL. Furthermore AFLGo was able to discover more crashes, reproduce more crashes and reach more target locations compared to approaches based on symbolic execution [BPNR17]. This indicates the validity and effectiveness of AFLGo's directed greybox fuzzing.

Other directed fuzzing approaches Hawkeye [CXL+18] is a related directed greybox-fuzzer which extends AFLGo's approach by e.g., also considering function pointers. CAFL [LSL21] is another directed fuzzer which also considers constraints (and their ordering) necessary to reach targeted sites.

3.3. Further mentions

In this subsection we want to mention further work which is also related but which we will not discuss in more elaborate detail.

Tree based splicing Tree based input splicing is a concept which can be found in grammar-based fuzzing. The idea is to insert a particular subtree at a suitable position in the derivation tree of another input [AFH+19, DRRG14, LKBS21]. This could be a more effective mutation strategy compared to e.g., just randomly re-generating nodes of the tree. The reason is that the spliced subtrees could e.g., be already semantically valid. Inserting such subtrees arguably has a higher likelihood of adding new valid features to an input, compared to random subtree re-generation.

Driller Driller [SGS+16] is an approach which combines (mutation-based) fuzzing with selective symbolic execution. The idea is to identify when a fuzzer is "stuck". This means that it has problems uncovering new areas in the SUT. The reason could be that new paths are guarded by specific input constraints which are unlikely to be fulfilled by random mutations. If Driller considers the fuzzer to be stuck, it employs concolic execution to generate inputs which hopefully reach new paths. Afterwards Driller transitions back to fuzzing until the fuzzer again gets stuck. To identify whether the fuzzer is stuck, Driller employs a heuristic. Namely, it checks whether the fuzzer has performed a certain amount of mutations (proportional to the input length) without uncovering new basic block transitions. If this is the case, Driller performs concolic execution on inputs it finds interesting. These are inputs which first uncovered certain basic block transitions or which first reached a certain execution threshold for a loop in the SUT [SGS+16].

An experimental evaluation indicates that Driller is able to discover more bugs compared to normal fuzzing and symbolic execution. Furthermore, the results suggest

[15] A control-flow graph describes the connections between basic blocks (e.g., for a procedure). A call-graph describes the call-relationships between procedures [CT12].

that the concolic execution was able to uncover new code transitions [SGS+16]. It thus appears to be a promising idea to support the fuzzer with additional measures once it reaches a discovery plateau.

AFLFast AFLFast [BPR19] is an adaptation of the (mutation-based) greybox-fuzzer AFL [Zal14]. The authors of AFLFast observe that fuzzers like AFL tend to mostly execute a limited set of high-frequency paths in an SUT. This is a downside as less visited paths could guard new code which could be covered. To mitigate this, AFLFast models the fuzzing process as a markov chain[16]. Specifically, it models the probabilities to reach a certain SUT path after fuzzing a current input. Böhme et al. [BPR19] distinguish between high- and low-frequency regions based on the transition probabilities in the markov chains. They then adapt the number of generated mutants per input and the order of seed inputs[17] to increase the likelihood of reaching low-frequency paths. Effectively, inputs which execute low-frequency paths are mutated earlier and more often.

Böhme et al. [BPR19] have found that AFLFast was able to generate more unique crashes than AFL and also could produce certain crashes that normal AFL was not able to trigger. These results indicate (similarly to FairFuzz [LS18]) that a focus on rarely visited SUT areas can be beneficial for the performance of a fuzzer.

[16] A markov chain can be used to model a stochastic process. It consists of a finite of states and probabilities to move between those states [HMB17].

[17] With seed inputs we do not only mean inputs which are provided *before* the start of a fuzzing campaign, but also which could be generated *during* a campaign. We clarify it as the term "seed inputs" can also be just used for the former case.

4. Generator-based Fuzzing with Input Features

In this section we will discuss our approach for targeted input feature learning and re-generation with the generator-based fuzzer JQF/Zest. We will start by highlighting why targeted feature generation could be beneficial for a fuzzing campaign. Next, we will present our approach both intuitively and more formally. Finally, we will discuss the implementation.

4.1. Motivation

We have seen in chapter 3 that learning and re-generating particular input features can have beneficial effects on the performance of a fuzzing campaign. Namely, FairFuzz [LS18], Template-Guided Concolic Testing (TGCT) [CLO18] and fuzzing with k-paths [HKZ22] indicate that targeted feature generation can increase the achieved coverage during fuzzing. Furthermore, FairFuzz and AFLFast [BPR19] suggest that *rarely* visited SUT areas could be a promising target to boost the coverage of a fuzzer. Moreover, Fairfuzz and fuzzing with k-paths indicate that *input features* could be leveraged to reach targeted areas. Thus, it appears reasonable to integrate all of those aspects (as FairFuzz did) and to also employ it for generator-based fuzzing. Namely, to develop an approach for generator-based fuzzing which allows to *identify* critical input features of rarely visited SUT areas and to *regenerate* those features. This approach could hit rarely visited areas more often. This then could uncover new SUT code which could increase the coverage and might lead to the discovery of more bugs (compared to undirected fuzzing). The particular advantage of generator-based fuzzing is that the generators can give certain guarantees (like syntactic validity of inputs). The targeting of rarely visited areas could thus be even more effective compared to the mutation-based approach of FairFuzz.

However, to the best of our knowledge there has not yet been any published research which deals with targeted feature generation or directed fuzzing in the context of generator-based fuzzing. This is a downside as the state of the art generator-based fuzzer JQF [PLS19b] and its algorithm Zest[18] [PLS+19c] have been particularly capable at fuzzing valid processing stages of an SUT [PLS+19c]. Integrating targeted feature generation into Zest could thus yield a particular chance to test valid stages of an SUT even more thoroughly.

Furthermore, targeted feature identification and generation could also be used as an aid in the debugging process. Namely, assume that you can identify shared features of a group of inputs. For instance, a group of inputs which trigger a particular bug. The shared features could indicate what part of an input triggers the bug. Therefore, they could help forming a debugging hypothesis. Furthermore, you could use these features to test the robustness of fixes by constructing new inputs which contain the critical features [KHSZ20]. Alhazen [KHSZ20] appears to be a promising tool for that in the context of grammar-based fuzzing. It employs a decision tree to learn input features of

[18] In the following "Zest" will always refer to its JQF implementation.

inputs which cause a particular bug. However, to the best of our knowledge, there has not yet been any research particularly for generator-based fuzzing.

Next, we will take a step back and describe on a more intuitive level *why* certain input features can be relevant for the success of a fuzzing campaign and why they might be difficult to generate. Afterwards we will discuss the particular challenges which JQF/Zest pose to realize input feature identification and targeted re-generation.

The relevance of input features for fuzz coverage We use the term "input feature" to refer to qualities of an input which can be observed on an input without running it on the SUT. This can be rather low-level qualities, like e.g., individual characters of an XML string. However, it could refer to more higher-level characteristics, like e.g., parent-child relationships between tags of an XML document. It is important that definition excludes aspects which require an execution of the input. Thus, aspects like its coverage or run time do not qualify as "input features" in our sense. For a more thorough definition please refer back to section 3.1.

Input features can be important for the coverage achieved by a fuzzer. Imagine that you want to test a build system which takes as input an XML document. The XML document could describe various aspects of the project. For instance, the dependencies or build configurations. We have provided an example of a fictive build system XML in Listing 2.

```
1  <project>
2      <source_directory>/path/to/project</source_directory>
3
4      <dependency_list>
5          <dependency source="web">lib_a</dependency>
6          <dependency source="local">lib_b</dependency>
7      </dependency_list>
8
9      <build>
10         <run_tests>yes</run_tests>
11         <package>yes<package>
12     <build>
13 </project>
```

Listing 2: Example of a (fictive) build system XML

It is reasonable to assume that different tags of the XML file are processed by different procedures of the SUT (i.e., the build system). Therefore we *need* certain tags to trigger (and thus test) certain functionalities of the SUT. However, it can be quite unlikely to trigger these functionalities if inputs are randomly generated.

Assume that you have a *generator* which can generate syntactically valid XML. This generator could use a dictionary of say 100 keywords to populate the content of an XML

file (e.g., the tag- or attribute names). The keywords could have been scraped from a set of existing project XMLs[19]. Imagine that keywords are sampled in *depth-first* order during generation. To create the XML sequence `"<dependency_list><dependency source="web">"` (line 4 and 5 of Listing 2), we would thus need to subsequently sample the keywords `"dependency_list"`, `"dependency"`, `"source"` and `"web"`. If each keyword has the same selection probability, we have a likelihood of 100^{-4} to select these keywords in this order. Thus, you will on average need $\frac{1}{100^{-4}} = 100.000.000$ attempts to generate this keyword sequence. This can be quite a lot if your SUT only allows a limited number of executions per second. Imagine that you can run 1.000 inputs per second. Then you would on average need $\frac{100.000.000}{1.000} = 100.000$ seconds ≈ 27.7 hours[20]. This estimation should be taken with a grain of salt as the example is completely fictive and the estimation ignores optimizations of modern fuzzers (like limited mutation of previous inputs). However, it is meant to illustrate that on paper it could be quite difficult to test certain functionalities of an SUT *even* if a generator only creates syntactically valid inputs. That is because certain functionalities might only be triggered by specific input features which are unlikely to be randomly produced. This assumption is supported for mutation-based fuzzers by FairFuzz [LS18] and AFLFast [BPR19].

Another aspect that this example should illustrate is that particular input features might be necessary to *uncover* certain functionalities of an SUT [LS18, HKZ22]. For example, to fuzz functionalities related to build instructions (line 10 and 11 of Listing 2), we *need* to have a `"<build>"` tag in our inputs (line 9). Therefore, the `"<build>"` tag *guards* code of the SUT which can only be reached if this tag is present. Otherwise we do not have a chance to test functionalities related to the running of tests or packaging (line 10 and 11 of Listing 2). Thus, the generation of particular input features can also be *necessary* to explore deeper processing stages of an SUT.

Input Features with Zest We have seen in section 2.2.3 that Zest [PLS+19c] is a generator-based fuzzing approach which can be particularly effective at fuzzing semantic processing stages of an SUT. One of Zest's key concepts is that it mutates the *pseudo-random byte stream* which generators use to make random decisions. Zest calls each bit of such a byte stream a *parameter* [PLS+19c]. We will thus refer to these streams also as *parameter streams*.

Mutating a parameter stream influences the decisions of a generators during the input creation. It therefore impacts the form and the content of the input. However, it will *keep* all the guarantees of the generators intact. For instance, that generated inputs are always syntactically valid. This is because *each* parameter stream should be processed by a generator such that the resulting input fulfils the guarantees. This allows Zest to perform random mutations on a low level (i.e., mutating bytes

[19] As has been e.g., done in the Zest [PLS+19c] evaluation.

[20] For simplicity, this estimation ignores the probabilities required to build the underlying tag- and attribute structure

on the parameter stream) which still *always* will result in e.g., syntactically valid inputs.

However, one of the downsides of Zest's approach is that it can become very difficult to identify particular features within Zest's inputs. Zest represents its inputs with their parameter stream [PLS+19c]. Therefore, Zest does not store the inputs as they are passed to the SUT, but only the byte stream to generate them.

The bytes of the parameter stream influence the decisions of the generator. They therefore also *indirectly* encode the features of the input. To illustrate this, assume that we generate XML structures using the pseudocode of Listing 3. This generator first creates an empty XML node (line 2). Then it randomly selects a dictionary entry as the tag name of the node (line 4-5). Finally, it chooses a random number of child nodes (line 7) and recursively generates each child (line 9).

```
1  XmlNode generateNode() {
2      XmlNode node = new XmlNode();
3
4      int index = random.nextInt(0, dictionary.size()));
5      node.tag = dictionary.get(index);
6
7      int numChildren = random.nextInt();
8      for (int i = 0; i < numChildren; i++) {
9          XmlNode child = generateNode();
10         node.addChild(child);
11     }
12
13     return node;
14 }
```

Listing 3: Pseudo-Code of a simple XML generator

The XML input "`<project><build/></project>`" could thus be generated by the parameter stream depicted in Figure 6. For simplicity, assume that each call to "random" consumes one byte. The employed dictionary is depicted in Table 3. You can see how the different features of the input are encoded in the parameter stream. For instance, the first byte determines the name of the root tag (i.e., "`project`"). This is because the first byte impacts the first selected dictionary index. Similarly, the second byte determines the number of children of the root node. The following bytes influence the features of the child node (i.e., "`<build/>`").

Index	Entry
0	project
1	build
2	package

Table 3: A dictionary represented as a table

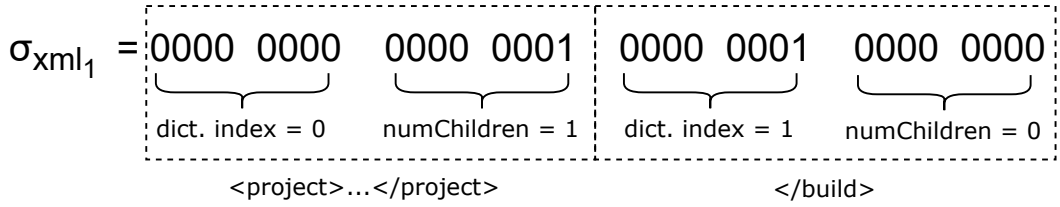


Figure 6: Parameter-Stream for the XML input "`<project><build/></project>`". Each number represents one bit.

The input features can thus be reconstructed from the parameter stream. However, it requires some effort. This is because you *have to* follow the code of the generator to identify and understand them. Compare this e.g., to a representation where the bytes *directly* represent individual ASCII characters of the input. In this case, you have a direct one-to-one correspondence between bytes and individual characters. This makes it easy to see which byte is "responsible" for which feature of the input.

Zest's input representation can thus make it difficult to understand how bytes of a parameter stream relate to different features of the input. In fact, it can even be more complicated than just presented. For example, *different* parameter streams could produce *the same* input. Namely, the parameter stream depicted in Figure 7 could produce the same XML as the original stream in Figure 6. For this assume, that the bounded `nextInt(...)` call (line 4, Listing 3) performs the bounding by calculating the next byte modulo the dictionary size. This guarantees that all returned integers will lie in the interval $[0, 2]$. However, it introduces *ambiguity* between the values on the parameter stream and how they are read as a result. This is because the parameter bytes for 0 and 9 produce the same result, since $0 \mod 3 = 0$ and $9 \mod 3 = 0$.

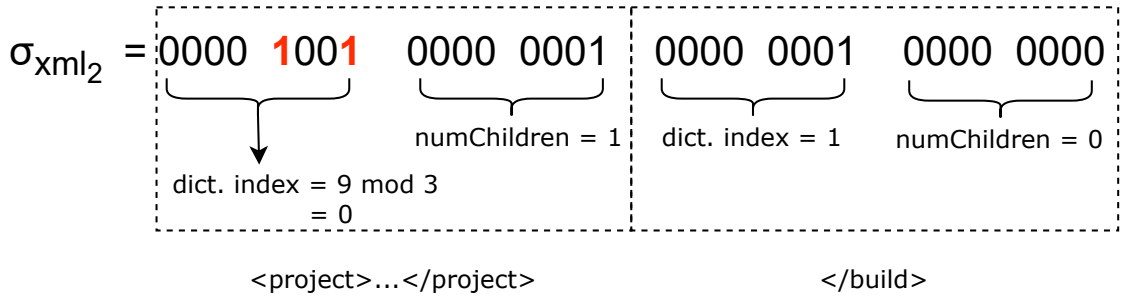


Figure 7: Mutated stream of Figure 6 which produces the same XML. Changed bits are highlighted red.

Another important aspect is that not all input features are explicitly represented in the parameter stream. For instance, we can derive from the parameter stream in Figure 6 that the tag `<build/>` is a direct child of the tag `<project>`. However, this information is not explicitly represented in the stream. Meaning, there is no

byte sequence in the stream which explicitly states that `<build/>` is a child tag of `<project>`. It can only be reconstructed if we run the generator on the stream and monitor the XML model during the creation. Thus, there can be input features that only indirectly follow from the parameter stream.

Finally, it is also relevant to mention that the *semantics* of parameter bytes can change depending on previous bytes. For this, consider a string generator which first samples the string length and then each individual character. A parameter stream for the string `"abc"` is given in Figure 8. Now take a look at the mutated stream in Figure 9. You can see how the mutation of the first byte influenced the semantic role of the third byte. Before, the third byte determined the second character of the string. After the mutation, the third byte now selects the length of the next string. Therefore, one can not generally assume that a certain position in the parameter stream always relates one particular input feature.

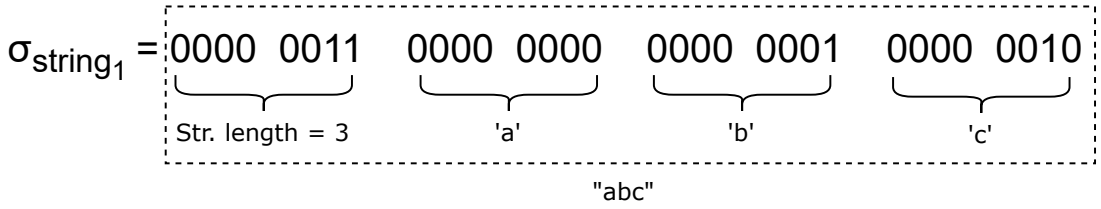


Figure 8: Parameter-Stream for the string `"abc"`.

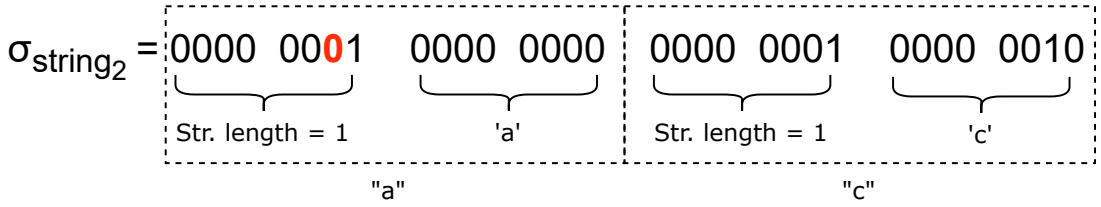


Figure 9: Parameter-Stream for the two strings `"a"` and `"b"`.

This entire discussion should make clear that it can be quite difficult to draw conclusions between a parameter stream and the features of the resulting input. Therefore it can also be difficult to apply the feature representation and identification methods we discussed in section 3.1. For instance, FairFuzz [LS18] employs a *mutation mask* to learn which bytes it can mutate while still hitting a targeted region. This preserves certain input features, e.g., characters of an XML tag, by blocking them from mutation. However, it can not be directly employed for Zest's parameter streams. FairFuzz learns its mutation mask by mutating each byte and observing its effects on the coverage trace. This can work fine with FairFuzz because it mutates the bytes which *directly* constitute the content of an input. The mutations of FairFuzz thus only have *local* effects. Namely, a single byte mutation only impacts that particular byte in the input. This would then e.g., also *only* impact the single character which is represented by this

byte. Therefore, FairFuzz can later use the result of each individual mutation to build a *combined* mutation mask as a union of the individual mutation experiments. This can be difficult with Zest. The reason is that the semantics of blocked bytes chan change if previous bytes are mutated. For example, imagine we require strings as inputs whose second character is a 'b'. In this case, the third byte of the parameter stream of Figure 8 would be blocked. The bytes before will likely not be blocked from mutation as they are either irrelevant or unlikely to change the interpretation of the third byte. However, there *are* cases where the interpretation of the third byte is changed by the mutation of previous bytes (see Figure 9). Therefore, one can not guarantee in Zest that a blocked byte will always have its intended impact on the input features. This situation can become even more complicated if you try to form one combined mutation mask which is the result of different isolated byte mutation experiments (as FairFuzz does).

Another challenge for FairFuzz on Zest is the ambiguity of Zest's parameter streams. We have previously highlighted that *different* bytes on a parameter stream can have the *same* impact on the resulting input (see Figure 6 and Figure 7). Therefore it is also not guaranteed that a mutation of a byte will also change the content of the input. This is a contrast to FairFuzz, where the mutated bytes always directly impact the content (because they *are* the content). It could thus happen in Zest that a byte will not be blocked from mutation because the mutation experiment happened to produce an equivalent byte value. This could then leave bytes open for mutation which in fact should best be blocked. FairFuzz's [LS18] mutation mask strategy can thus rather not be used as effectively for Zest as it has been for AFL.

The approach of Template-Guided Concolic Testing (TGCT) [CLO18] faces similar issues. TGCT employed pattern mining to identify character sequences in inputs which should be reproduced to increase the coverage. In theory, one could attempt a similar technique to learn shared bytes of Zest inputs which are important for e.g., valid coverage. However, a problem is again the ambiguity of Zest's input bytes. Therefore it is not guaranteed that bytes which produce the same input features will also have the same byte values. These bytes would then not be identified as a pattern even though they have the same effect on the input. Moreover, even if one could identify a pattern, then it is not quite clear *where* to insert this pattern. This is because the insertion position *must* be at a place in the parameter stream where the generator will read the pattern bytes in their intended sense. Identifying such positions however can be quite difficult, as they depend on the implementation and the course of a generator for a particular input.

Finally, one could also consider whether Havrikov et al.'s *k-paths* approach [HKZ22] might be a suitable technique to represent and identify input features with Zest. However, the problem here is that k-paths are a *grammar-based* concept relies on (sub-) paths in a derivation tree. Zest's parameter streams however are unstructured and have no notion of a path. Therefore, it is not really a concept which can be directly employed for Zest.

In summary, this discussion should make clear that Zest's parameter streams do encode the features of the resulting input. However, it can be very difficult to extract

those features without running the parameter stream on the corresponding generator. Furthermore, the parameter streams make it difficult to employ previous techniques which attempt to learn and reproduce particular features of an input. Important reasons are that that the position of a byte in the parameter stream is no guarantee for its semantic impact and that different byte values can have the same impact on the input. Furthermore, it can be difficult to know where to insert a sequence of bytes even if one should have identified bytes which are responsible for a certain feature.

4.2. The Approach

Representing features on the model level The issues that we have discussed above made it difficult for us to develop a technique which can learn and regenerate particular input features on the basis of Zest's parameter streams. To overcome this, we have decided to instead focus on the *model-level* of an input. Meaning, we analyze the features of an input *after* it has been generated from a parameter stream. The model of an input is an abstract representation of an input. However, it might also be materialized if the model actually describes the internal structure of an input. When studying the Zest evaluation [PLS+19c], we have noticed that most of Zest's test subjects take inputs which can be represented with a *tree-based model*. Namely, the build systems *Maven* and *Ant* take XML files. The subjects *Closure* and *Rhino* process JavaScript. When analyzing JQF's XML and JavaScript generator it becomes evident that the produced inputs inherently have a tree-based structure[21]. This means that their models form a tree of "nodes". To illustrate, consider the XML string represented in Listing 4. JQF's `XmlDocumentGenerator` would internally build a tree-based model for this input which is illustrated in Figure 10. This model is not necessarily what is later passed as an input to the SUT. Instead, it is an object-based representation of a structured input. This representation could be then converted to other ones which describe the information in another form (e.g., the XML string in Listing 4). However, the important aspect is that this model is *materialized*. Meaning that there are objects in main memory which represent the individual nodes and connections of an input. Thus we could also change the features of the input if we change aspects in the model.

```
1 <project>
2     <version>1.0.0</version>
3     </build name="out">
4 </project>
```

Listing 4: A simple XML string

Another important insight is that input features can be much more easily identified on a tree-based model compared to its parameter stream representation. First, the

[21] JQF's generators do not always "materialize" the trees as a data structure. Thee tree-based form of the inputs is nonetheless recognizable in the generators. We will discuss it in more detail when presenting our implementation.

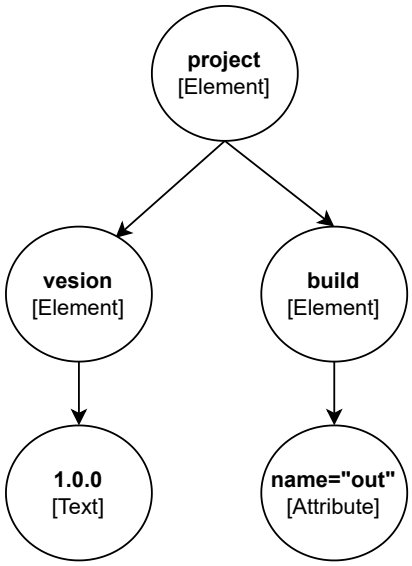


Figure 10: Tree model of the XML string of Listing 4

input models speak for themselves (once generated). This means that you do not need to follow the implementation of the generator to identify the features of an input. For example, the existence of a certain XML tag is represented in the model by the existence of a corresponding `Element` node. This can be immediately identified, without first running the input again through the generator and monitoring its execution.

Second, the input features are encoded less abstractly compared to Zest's parameter streams. For example, the fact that there exists a tag with the name "`project`" is represented by a logical entity (i.e., a node) which has the tag name as a readable string. This is more clear compared to Zest where the features would be encoded in a sequence of bytes.

Third, input features are also encoded more directly in contrast to parameter streams. For instance, the fact that the "`project`" tag contains a "`version`" tag is evident due to the parent-child relationship in the model. If you recall, such a tag relationship was not explicitly represented in the parameter streams of our example XML generator (Listing 3).

These aspects have lead us to the decision to analyze and reproduce input features on the *model level*. Particularly, we will focus on inputs which have a (materialized) *tree-based* model. This means that they are structured as a tree of node objects which describe the form and content of the input. While this might seem like a limitation, it allows us to employ our approach on relatively important input formats like XML, JavaScript or HTML. Furthermore, our ideas could possibly be transferred to other domains which deal with tree-based inputs. One example would be grammar-based fuzzing where inputs can be natively represented as derivation trees.

Features of tree-based inputs Input features can be represented relatively easily when dealing with tree-based models. There are two types of basic features. Either the *existence* of a particular node or the *connection* between two nodes. Nodes can furthermore have a type (e.g., `"Element"` in Figure 10]) and additional metadata (e.g., the name of an `Element` node) to qualify them. These four qualities however can suffice to describe the entire input structure and content. Especially when combined into more complex features. Particularly important for us is the concept of a *path* through a model. This notion refers to the paths which lead from the root of a tree to each of its leaves. For example, the tree model of Figure 10 has the paths depicted in Figure 11. Each bulletpoint indicates an individual path. The components of one path are separated by the symbol ">". We call each such path a *feature path* to highlight that it describes the features of an input. Each path component (or node in the tree) is accordingly referred to as a *feature node*.

- **project** [Element] > **version** [Element] > **1.0.0** [Text]
- **project** [Element] > **build** [Element] > **name="out"** [Attribute]

Figure 11: Tree paths of the model of Figure 10

Learning shared features of tree-based inputs Each tree-based input can be decomposed into its list of feature paths. The feature paths are thus are another representation of an input. Furthermore, each feature path can be modelled as a *sequence* of strings. This can be done by converting each feature node to a string containing the type and metadata of the node. We have seen this in Figure 11.

The sequential nature of feature paths allows us to employ *sequential pattern mining*. You can recall that sequential pattern mining finds shared sub-sequences in a list of sequences (see section 2.3.1). A feature path may be represented as a sequence of nodes (or a string which encodes this node). A sub-sequence would then correspond to a sub-path in a feature path. For readability we will use the symbol ">" to separate elements of a sequence. Furthermore, we will only analyze sequences whose itemsets all have exactly one item. We will thus omit the curly brackets ({}) when describing components of a sequence. For example the notation "**version** [Element] > **1.0.0** [Text]" describes the formal sequence ⟨{"**version** [Element]"}, {"**1.0.0** [Text]"}⟩. Furthermore, we will only analyze sub-sequences which are strictly contiguous in the underlying sequences.

The sequence of our notation example is a sub-sequence (and thus sub-path) of Figure 11's first path. Importantly, you can see how this sub-path still conveys information about the *features* of the underlying input because it is a *feature (sub-) path*. Namely, the features are that you have an `Element` node with the name "version" which contains a `Text` node with the content "1.0.0". Sequential pattern mining on feature paths could thus allow us to identify *shared features* of a set of inputs. To illustrate this,

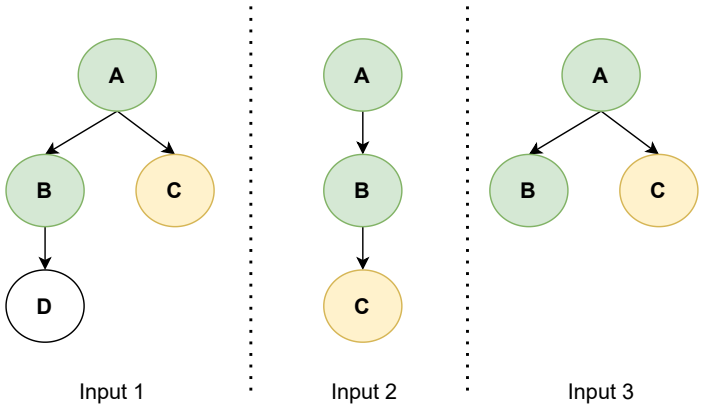


Figure 12: Three input trees with shared sub-paths. Shared sub-paths are highlighted in color.

consider Figure 12. This Figure shows the tree model of three inputs. You can see that they all share a certain set of sub-paths. These sub-paths are highlighted in matching colors. Furthermore, these shared sub-paths could be identified using sequential pattern mining.

Sequential pattern mining gets as input a list of sequences. It then returns sub-sequences which appear in at least *minsup* many input sequences. The *minsup* value may also be given as a fraction of the inputs (the so-called relative support) (see section 2.3.1). This will be our preferred method. For example, imagine that you have a list of the following sequences and a relative *minsup* of 0.5.

- $1 > 2 > 3$
- $2 > 3$
- $4 > 1 > 2$

In this case you would identify the subsequences "$1 > 2$" and "$2 > 3$" as frequent sub-sequences as they appear in at least half of the input sequences[22].

Sequential pattern mining could now be used to find patterns between tree-based inputs. For this consider Table 4. Here we have presented the feature paths of Figure 12 as a *sequence database*. This is the formal name for the sequence list a sequential pattern miner receives as input. The feature paths of each tree have been combined into one sequence per input. This has been done by concatenating each path. Each path is linked by an additional entry which contains the index (or "number") of the input. We will discuss its role later in more detail.

Now imagine that we perform sequential pattern mining on this database with a relative *minsup* of 1.0. The returned sub-sequences would be:

[22]In fact, you would also find the individual symbols "1", "2" and "3" as frequent patterns. However we focus here on *closed* patterns. More on that later.

Input	Sequence
1	$\langle\{A\},\{B\},\{D\},\{1\},\{A\},\{C\}\rangle$
2	$\langle\{A\},\{B\},\{C\}\rangle$
3	$\langle\{A\},\{B\},\{3\},\{A\},\{C\}\rangle$

Table 4: Sequence database for the paths of Figure 12

- $\langle\{A\},\{B\}\rangle$
- $\langle\{C\}\rangle$

These are exactly the shared sub-paths we have highlighted in Figure 12. Therefore sequential pattern mining appears like a valid option to identify shared sub-paths in a given list of trees.

To finalize our discussion, we want to mention some further aspects. Namely, we have not mentioned some hidden configuration for the pattern mining example above. First, we have only mined for *closed* patterns. Intuitively, this removes all sub-patterns which are contained in other ones. Otherwise, the sub-sequences $\langle\{A\}\rangle$ and $\langle\{B\}\rangle$ would also have been returned as patterns. They are however not as interesting because they are already contained in another pattern. Furthermore, these smaller patterns lose the information that the nodes A and B are actually connected in each tree. Returning such sub-patterns could thus also add confusion to the results. For more details on closed patterns see section 2.3.1.

Another aspect is that we have set the constraint that $maxgap = 1$. The *maxgap* value defines how many items may lay between items of a pattern. With $maxgap = 1$ we say that patterns must be *contiguous* sequences of items[23]. If we had not defined a *maxgap* constraint, then the sub-sequence $\langle\{A\},\{C\}\rangle$ would also have been returned as the items A and C follow upon each other in every input sequence. They however do not always follow contiguously on one another. Therefore, they do not always form sub-paths in the underlying tees and are thus not interesting to us. You may also have noticed that we have spoken of the distance between *items* when describing the *maxgap* constraint here. In fact, the distance refers to *itemsets*. However, as discussed before, our itemsets always consist of exactly one item. Therefore, the distance between itemsets and items is equivalent in our case. More information on *items* versus *itemsets* and the *maxgap* constraint can again be found in section 2.3.1.

Furthermore, we want to explain why we have added the itemsets $\{1\}$ and $\{3\}$ to the sequences of input one and three. These prevent that patterns can be found *between* paths. We call these components *path-end components* because they signal the end of every path (if we have multiple paths in an input). Each path-end component is a number corresponding to the index of the input. They are thus unique per input and can *not* be identified as a pattern between inputs. They form a pattern barrier between

[23]This interpretation of the *maxgap* value is in accordence with the interpretation of the pattern mining algorithm that we use. Namely, CM-SPAM [FV22].

paths of an input (in combination with $maxgap = 1$). The result is that patterns can only be found *along* a path. This is exactly what we want. Otherwise we could find patterns *between* the end of one path and the start of another one. These patterns are not useful for us because they do not constitute real sub-paths in the underlying inputs.

Finally, we want to mention that *graph-based pattern mining* might be an even more natural way to mine for patterns in trees. In fact, this is something we experimented with over the course of this thesis. However, we had some difficulty finding (peer-reviewed) implementations of graph-mining for *directed* graphs. Directedness however can be a crucial feature. For instance, in XML it is very important whether tag `<a>` contains tag `<b>` or the other way around. This information however would get lost when mining on *undirected* graphs. Meaning, the undirected graph pattern `a - b`[24] could mean either case. We thus adapted the input graphs so that we could *reconstruct* the directedness of patterns even when mining with formally *undirected* graphs[25]. However, it added considerable complexity and was thus not picked up again when implementing our final approach. Mining on graphs can nevertheless solve some issues with sequential pattern mining we will discuss in the next paragraph. It could thus be interesting for future work.

Reconstructing structures from sequences We have discussed in the motivation (section 4.1) why input features can be relevant to reach certain areas of an SUT. These features might be represented as e.g., sub-paths in a tree model. However, they also could form more complex structures. To illustrate this let us again pick up our fictive build system SUT. Imagine that there is a rarely visited branch in the SUT which is only activated if we package the project and also run the tests when building. The corresponding XML sub-structure is presented in Listing 5. It could trigger a specific branch because the processes might need coordination. The graph of the necessary XML sub-structure is depicted in Figure 13.

```
1 <build>
2     <run_tests>yes</run_tests>
3     <package>yes</package>
4 </build
```

Listing 5: XML substructure to trigger a specific rare branch

Imagine that we have several inputs which trigger the rare branch and perform sequential pattern mining on their paths as discussed above. We would then find the two paths as patterns shown in Figure 14.

[24]This means graph node "a" is conntected with node "b" but without any information on the direction.
[25]The details are omitted here for brevity. We however hand in our implementation. For this please have a look at the JavaScript mining of `jqf-feature`.

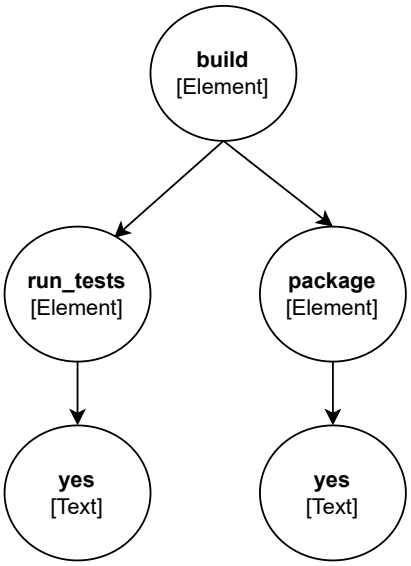


Figure 13: Graph of the XML of Listing 5

- **build** [Element] > **run_tests** [Element] > **yes** [Text]
- **build** [Element] > **package** [Element] > **yes** [Text]

Figure 14: Feature paths patterns for the rare branch corresponding to Figure 13. Each bulletpoint indicates one pattern (i.e., frequent sub-sequence).

These sequential patterns correctly describe the paths of the necessary structure. Yet, they do *not* convey that the paths should form a *shared structure*. Namely, that both paths should have *the same* root node (as depicted in Figure 13). This is a downside because if we would add these paths *individually* to an input, we will likely not trigger the rare branch. On the contrary, we would likely produce a semantically invalid input, because we would add *two* distinct `<build>` tags.

To resolve this issue, we employ a heuristic which *reconstructs* a structure from a set of mined paths. Namely, we assume that identical path-prefixes[26] form one *shared* sub-path in the tree. This allows us to correctly reconstruct the structure of Figure 13 from the patterns that we obtained. The reason is that both patterns start with the prefix "**build** [Element]". The node for each prefix would thus be merged into *one* by our heuristic. This would also work with more complex prefixes. Imagine that both patterns started with "**project** [Element] > **build** [Element]". Again, we would only create *one* node for each prefix node. The result with (and without) our reconstruction

[26] A path-prefix is a sub-sequence of path components beginning at the first component of a path. A component is one "item" or "node" of the path.

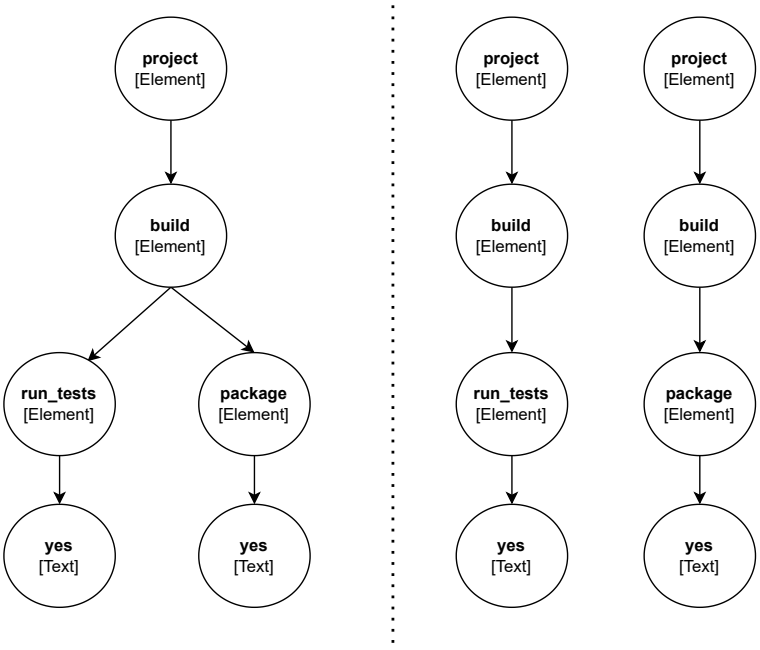


Figure 15: Example of our reconstruction heuristic. Left depicts the result with our heuristic. Right shows the result without.

heuristic is illustrated in Figure 15.

Our heuristic allows us to reconstruct structures from one-dimensional patterns. However it could also erroneously merge prefix nodes which in fact should be represented by distinct nodes. This is a trade-off we accept with our heuristic. Our intuition is that the regeneration of multi-dimensional structures can be important to trigger certain SUT functionalities. Thus, we have to identify them in some way. The presented heuristic offers an solution for sequential patterns. Nevertheless, it could produce imperfect or even wrong results.

Regeneration of features So far we have discussed how input features could be *represented* and *learned* with tree-based inputs. Now we will present our approach to *regenerate* them. The concept that we employ is *tree-based splicing*.

Tree-based splicing is a technique which can be found in grammar-based fuzzing [AFH+19, DRRG14, LKBS21]. The idea is to *add* a structure to a given tree input. Imagine that we have identified the subtree depicted in Figure 16 as a feature to regenerate. This could be a substructure which triggers a certain behavior in an SUT. To regenerate this feature we could take the tree model of another input and *add* our substructure. This process is shown in Figure 17. We will call this process *(tree-based) splicing*. Splicing will add the substructure to the model of the input. The generator could then process the model to produce a final input for the SUT containing the

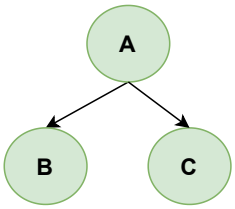


Figure 16: Example of a mined subtree

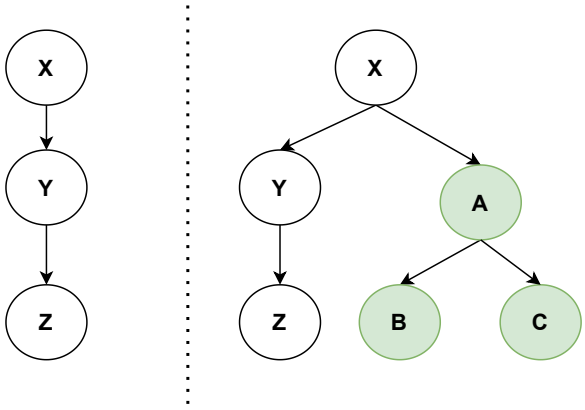


Figure 17: Example of simple splicing. Left shows the input before splicing. Right depicts the result afterwards.

spliced features. This should then trigger our wanted behavior.

However, the process of splicing can be more complicated than in our simple example. There are at least two concerns: Syntactic validity and semantic validity. Let us first discuss syntactic validity.

Our simple example ignores that tree nodes can have a *type*. For example, in XML we can distinguish between nodes which represent tags (here referred to as `Element` nodes) and nodes which represent tag `Attributes`. Imagine that we deal with a tree-based model where parent-child relationship must follow the hierarchical relationships in XML. An `Element` node could thus have an `Attribute` as a child node. This is because `Attributes` are aspects of `Elements` and not the other way around. However, `Attributes` can *not* have an `Element` as their child. This would relate to *syntactically invalid* XML. Even if a generator could try to make sense of such a structure, it would likely produce a result similar to Listing 6. Here the tag `<b>` has been attached as a child of the attribute `some_attribute`. The result is not syntactically valid XML.

```
1     </a some_attribute=</b>>
```

Listing 6: Attaching tag `</b>` to the attribute `some_attribute`

One thus has to be careful when choosing *where* a structure is spliced to. We call the node we attach a structure to the *splicing target*. To ensure the syntactic validity

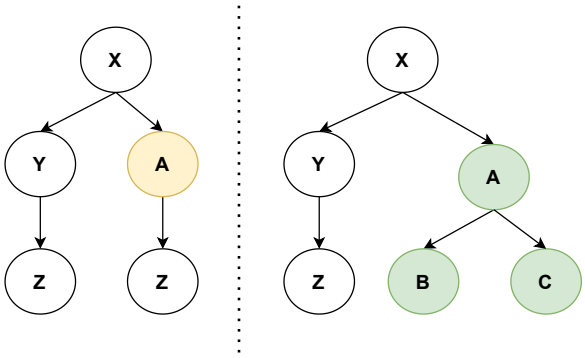


Figure 18: Realizing splicing as an overwrite. Left shows the input before the splicing. Right shows the splicing result.

of inputs, one has to pay attention that our spliced structure can be attached to our splicing target as a child. This is a check which has to be implemented individually per input format (e.g., individually for XML and JavaScript). That is because each input format can have different node types and different allowed connections[27]. We circumvent this issue. Namely, we do not actually add to a splicing target but actually *overwrite* it. Therefore, our splicing targets must have the same type as the root of our splicing structure[28]. For example, we can only overwrite `Element` nodes with other `Element` nodes or `Attributes` with other `Attributes`. The result is virtually the same as appending to a node. The difference is that we *delete* the substructure of our splicing target and replace it with our spliced structure. The process is depicted in Figure 18. The overwritten target node is highlighted yellow on the left side of the image.

Implementing splicing as an overwrite *requires* the existence of a type-identical target node when splicing. This potentially reduces our pool of available splicing targets. However, it arguably simplifies the implementation. Otherwise we would have to mind the acceptable parent-child relationships of around 30 different JavaScript node types. This could have been a sizeable effort and would have added complexity.

Another concern for the syntactic validity of inputs is the potential *incompleteness* of patterns. Imagine that we have a tree representation of XML attributes where the name and value of an attribute are represented as children of an `Attribute` node. This is illustrated on the left side of Figure 19 for the attribute `source="web"`. It could easily be that attributes with the name `"source"` always trigger a certain rare branch *regardless* of the attribute value. Pattern mining could then find an *incomplete* attribute substructure as shown on the right side of Figure 19. It is incomplete because XML attributes *require* both an attribute name and value. If we would splice this incomplete structure, we could thus not produce syntactically valid XML. To resolve this, we need to implement routines which check the completeness of patterns and

[27] Grammar-based fuzzing can be at an advantage here because the productions can define which non-terminal can be expanded into which symbols. This allows for the automatic extraction of such rules.

[28] With "splicing structure" we mean the structure we want to splice.

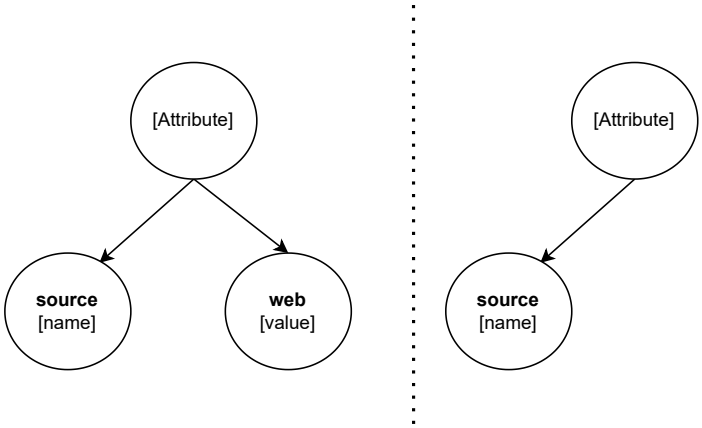


Figure 19: Example of a complete (left) and incomplete (right) attribute structure

perform a completion if necessary. We do this by letting the generators fill the missing parts in a random manner. In our case it is however only done for JavaScript because our XML nodes can not produce incomplete patterns. For instance, our `Attribute` nodes contain the attribute name *and* value directly within the `Attribute` node. This reduces the granularity of learnable patterns but also makes it impossible to mine incomplete attributes.

Finally, we want to discuss concerns regarding *semantic* validity. Imagine that we have learned the XML structure `<project></build></project>` as a pattern. Furthermore imagine that we want to generate new inputs off of that via splicing. For instance, by splicing it to inputs which were previously valid. Let us assume that *all* valid inputs have a `<project>` tag as their root. If we would add our structure as a *child* to an existing tag, the we would create an *additional* `<project>` tag which is likely semantically invalid. Our input would thus likely be discarded with an error.

On the other hand, if we would *overwrite* an existing node, then we would likely have to overwrite the root node as it should be the only `<project>` tag in valid inputs. Overwriting the root however would *delete* the entire previous structure below the root. The only thing which would remain is our added `</build>` tag. This input is likely too simple and would also be discarded as semantically invalid. Thus, there is a certain dilemma.

Our solution is to perform a *mix* between overwriting and appending. Our splicing implementation *overwrites* the metadata of the target node (e.g., the tag name). However it keeps *all* original child nodes of the target (if possible)[29]. It then *adds* all the children of our splicing structure root to the splicing target. This makes our splicing less destructive than a simple overwrite. Furthermore, our splicing implementation *favors* splicing targets which have the same metadata as the splicing root[30]. Therefore,

[29]Sometimes our generators can have limitations regarding the number of children of a node. In such cases we have to compromise by deleting (randomly selected) children of the splicing target or possibly even our splicing structure.

[30]Splicing root means the root node of the structure we splice

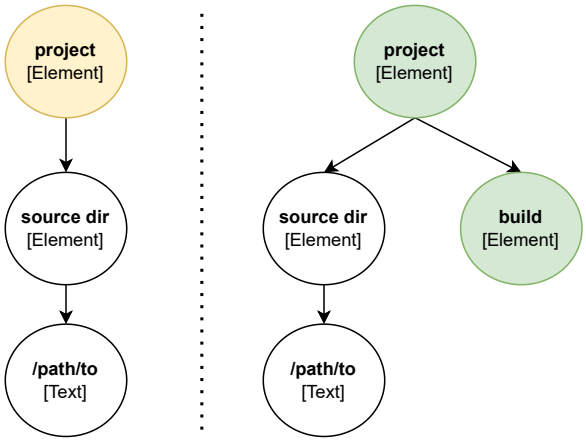


Figure 20: Splicing which combines a node overwrite with child appending. Left depicts the original input. Right shows the splicing result.

we would e.g., rather select nodes as a splicing target which have the same tag name as our splicing root. This would bias us to always select the `<project>` root as a target in our example. Moreover, our implementation would not categorically remove the original children, but (ideally) only add to it. This should increase the probability for creating a fully valid input compared to a complete subtree replacement. The entire idea is illustrated in Figure 20. The original `<project>` root is highlighted yellow to indicate that it would be the favored splicing target for the splicing structure `<project></build></project>`. You can see that we have overwritten the original root but simultaneously kept the original children.

Identification of rare branches We have mentioned at the beginning of this chapter that our goal is to better explore *rarely* visited branches. For this, we first have to define what we mean by rare. To define *rarity* we employ FairFuzz's [LS18] approach. FairFuzz defines a branch as rare if its number of hits is below a dynamically set threshold. FairFuzz calls this threshold the *rarity cutoff*. The details and formula of FairFuzz's approach can be found in section 3.1.1. We have chosen FairFuzz's approach because it allows us to *dynamically* identify which branches are rare and which not. This is helpful, because which branch is "rare" may change over the course of a fuzzing campaign. Furthermore, FairFuzz's approach allows us to automatically adjust the concept of rarity per SUT. This is also important, because the number of hits which constitute rarity can also change from SUT to SUT. For instance, in some SUTs 100 hits could be considered rare while there might be others where the number could be closer to $10,000$. Therefore, FairFuzz's technique seemed like a good solution.

However, when experimenting with our approach we have sometimes ran out of rare branches to target with FairFuzz's approach. The reason is because FairFuzz's cutoff can be quite restrictive and because we target each branch at max *once*. Thus, we need a "new" rare branch for each targeting. Furthermore, we also exclude branches which we consider related to previously targeted ones. Together with the fact that

FairFuzz's cutoff can be quite restrictive, we can run of branches to target during a fuzzing campaign.

These target exclusions are an optimization of our approach. The fact that a previously targeted branch is still rare indicates that its spliced pattern was *not successful*. Otherwise the branch would likely not be rare anymore. Furthermore, we only store a limited amount of inputs per branch (to control the memory overhead). Thus it is likely that we would mine on the same inputs as before and produce the same pattern which was *not effective* the last time. Repeatedly using ineffective patterns can be a waste of resources. To mitigate this, we prevent the re-targeting of previously targeted branches. Furthermore, we exclude "related" branches. These are branches which would use the same inputs for mining as previous targets. The idea is to exclude rare branches which are always triggered together with a previous target. For instance, imagine that you fuzz our example build SUT. There might be a rare branch which is only triggered when the project XML contains a `<build>` tag. However, it is likely that the `<build>` tag will trigger an entire list of related rare branches. This is because the `<build>` tag might activate a complex set of (rarely activated) SUT procedures. The corresponding rare branches would have different branch IDs but they would *always* be triggered together. Thus, they would be activated by the *same* inputs and would produce the same (in-/effective) patterns. For this reason, we also exclude branches which we consider related according to our heuristic.

Our exclusion of previous targets, together with FairFuzz's rather restrictive rarity cutoff can let our approach run out of further rare branches to target. To continue targeting, we switch to another rarity heuristic in such cases. This heuristic defines all branches with hits in the lowest 30% percentile as targets. That heuristic can be more inclusive than FairFuzz's one. For instance, we analyzed rarity on the SUT *rhino* over 20 campaigns at different points at time. Each campaign had a duration of 12 hours. Over these campaigns, only 4% of valid branches (241 absolute) were identified as rare. Thus we saw a potential to raise the threshold.

There could be also other methods to raise the threshold. If you recall, FairFuzz's rarity cutoff is always a 2^i. Thus one could alternatively increase the exponent i in incremental steps. We have decided against this option, since the increases could be considerably huge at some point. By focusing on the lowest 30% percentile we can ensure that rarity will always only include the low end of hit counts. Formally however we only define branches as rare according to FairFuzz's formula. Our alternative heuristic only serves as a fallback. The idea is to further support the SUT exploration by targeting comparatively rarely visited areas even if we have exhausted the formally rare branches.

Handling multiple pattern subtrees So far, we have only discussed the case when pattern mining returned *one* pattern subtree for a group of inputs. However, it could happen that we obtain multiple ones. The reason is that our subtree reconstruction algorithm could produce several distinct pattern trees. To illustrate that, let us go back to our example build system SUT. Imagine that we want to target a branch which is

- **dependency_list** [Element] > **dependency** [Element]
- **build** [Element] > **run_tests** [Element]

Figure 21: Example of feature paths which result in different pattern trees

related to handling dependencies. Let us assume that we have several inputs which trigger this functionality and that we mine on their model tree paths. The mining algorithm could return the frequent feature paths shown in Figure 21.

You can see that mining returned a subpath related to dependencies, as one would expect (the first pattern). However, the mining incidentally also found a pattern related to the running of tests during building (the second pattern). This could easily happen, because pattern mining returns *every* shared subpath it finds. This might also include ones which actually do not contribute to our target branch. This raises two issues.

First, we can *not* build a combined tree from the patterns in our example. This is because the obtained patterns have no shared path prefixes. We thus end up with *multiple* trees as a result. This is shown in Figure 22.

Second, the example highlights that we can incidentally find patterns which *do not contribute* to our target branch. These can make it in fact more difficult to trigger our target, because the more complex the pattern, the more likely it (arguably) is to violate input validity by splicing. Especially if we splice several trees at once.

Let us first discuss how one can deal with multiple pattern trees (as in Figure 22). One option would be to always splice *all* pattern trees that one has obtained. Meaning, if you have an input you want to splice to, then you would always splice *all* obtained patterns at once into that input. This will guarantee that you also always will splice the trees which actually contribute to the target branch. On the other hand, it could also increase the likelihood of creating invalid inputs because you splice *multiple* trees into one input. Each splicing is a (somewhat controlled but still) randomized manipulation of a potentially valid input. And each randomized manipulation has the potential to break validity.

Another idea would be to splice different *combinations* of obtained patterns. For instance, imagine that we have obtained the tree set $\{t_1, t_2, t_3\}$ to splice. Then one could first try to splice $\{t_1, t_2\}$ together, then $\{t_2, t_3\}$, afterwards only $\{t_1\}$ and so on. Each time one could monitor what subtree combinations perform best and finally settle on that combination which triggers the target branch most effectively. However, this approach could lead to a combinatorial explosion with a growing number of subtrees. It also would be particularly inefficient if most subtrees do not contribute to the targeting of the branch. We would try them multiple times with different combinations without actually triggering our target branch.

To avoid the above issues, we only splice *one* learned subtree. The subtree we splice is obtained by comparing the performance of each individual tree. This means that

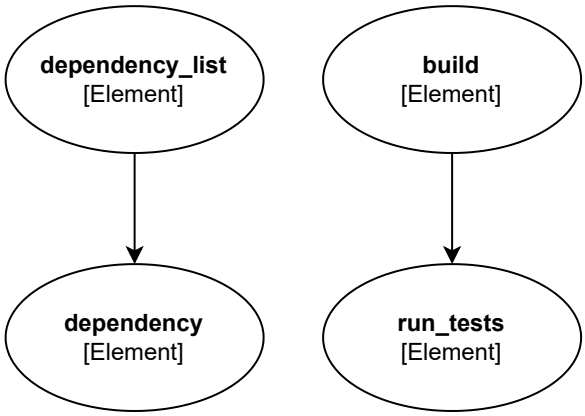


Figure 22: Reconstructed pattern trees of the paths of Figure 21

we splice each subtree individually and monitor how effective it is at triggering the target branch. Finally, we compare which tree was most effective and only keep that. Effectiveness is measured with the ratio $\frac{target\_hits_i}{splicing\_attempts_i}$ for each learned tree t_i. We perform a fixed number of splicing attempts per tree. The number of attempts was set to 100 in our experiments.

The advantage of our approach is that we do not have to splice multiple trees at once. This can be beneficial for the validity. On the other hand, our approach could *fail* to trigger rare branches which actually require the splicing of multiple disjunct subtrees. Thus, there is a certain tradeoff between keeping multiple trees and reducing them. Our approach should at least help us identify the subtree which is best at triggering the targeted branch if we only need tree. However, it remains that our approach will not fully identify the pattern if we need to splice multiple disjunct subtrees.

Targeting start and interval The goal of our overall approach is to support the fuzzing campaign by focusing on rarely visited areas. However, an important question is *when* and *how often* this targeting should happen. For this question, we found it difficult to find previous research that would serve as an effective basis for us. Our intuition was to start targeting once the performance of the fuzzer (e.g., the branch discovery) begins to plateau. This indicates that the fuzzer has exhausted the set of branches which it could find with rather simple mutations. It might thus make sense to now concentrate on rarely visited SUT areas. Because these areas might be difficult to trigger but still could guard code to cover.

This time we found it difficult to base ourselves on FairFuzz [LS18]. That is because FairFuzz *only* mutates inputs which hit rare branches. However, we found it difficult to understand when rarity is established by FairFuzz. It could be that FairFuzz updates the rarity cutoff with each new input it mutates. Another option is that it is only updated in certain intervals. In theory, it could also be that the rarity cutoff is set beforehand on the basis of previous fuzz campaigns. We find that the FairFuzz [LS18] paper is not very explicit regarding that. However, our intuition is that rarity in FairFuzz is established

dynamically during a fuzzing campaign[31]. Nevertheless, due to the uncertainty we found it difficult to use FairFuzz to decide when to start targeting ourselves. One could theoretically begin targeting immediately at the start of a fuzzing campaign. However, our approach pattern mining and splicing arguably incurs considerably more overhead compared to FairFuzz's mutation mask. Thus it seemed sensible to identify a targeting start time when the trade-off between additional overhead and performance gains would be best (or at least better).

Driller [SGS+16] is an approach which has a similar idea to us. This technique switches from mutation-based fuzzing to concolic execution once it detects that the fuzzer is "stuck", which means that it has difficulties finding more branches. But again, we found the described approach difficult to apply. Driller switches to concolic execution once it has performed a specified amount of mutations without uncovering new branches. The number of mutations it performs is proportional to the length of an input [SGS+16]. Still, we found this rather vague, because the description is not more explicit than here presented. Thus we do not know which exact mutation threshold has been employed by Driller. Furthermore, it is not quite clear *when* Driller switches to concolic execution. However, to the best of our understanding it does so, once it has checked *all* inputs which it has stored in its queue.

Due to these difficulties, we have developed a custom heuristic to identify when the performance of a fuzzer starts to degrade. It works by analyzing the number of paths that the fuzzer discovers over time. A path in Zest refers to a coverage trace which contains previously not seen branches or which managed to increase the hit counts of some branches. We analyze the path discovery ratio to estimate the inflection point in the performance. For this, we sample the current number of discovered paths every 1000 seconds. Let this number be $paths_i$ at a measurement interval i. The path discovery ratio at interval $i+1$ is then defined as $pdr_{i+1} := \frac{paths_{i+1} - paths_i}{paths_i}$. We have found that in practice that a path discovery ratio below 0.05 was a good indicator for the inflection of the performance or the initial stages of plateauing. With "performance" we refer to the discovery of new branches. Thus, we start targeting once the path discovery ratio drops below 0.05. If the path discovery ratio drops once below we target different rare branches in intervals of 10 minutes. This means that we do not analyze the path discovery ratio again once we heuristically determined a performance degradation. Instead, we now focus on targeting rare branches. If a rare branch is targeted, we first learn the pattern and then splice it in newly created inputs for at max 10 minutes. Afterwards, we select a new branch to target. However, we will discuss the details of targeting more in our description of the integrated approach.

Splicing and parametric inputs Before we present the integrated approach, let us discuss how we combine model-based splicing with the parametric inputs of Zest. You will recall that Zest represents its inputs by the parameter stream which generators use to create inputs. Due to the difficulties of identifying and regenerating features with a

[31]Because the FairFuzz paper also refers to the cutoff as a "dynamic" rarity cutoff [LS18].

parameter stream, we have to decided to splice on the model level. This means that we work on the models of inputs which are the *result* of generators. Splicing thus means in our context, that we have a tree of node objects in main memory which represent an input. We would then take a subtree of other nodes and adapt the connections in the original model to perform splicing. This should then we can create an input (model) which contains new features after the splicing. However, we lack the *parameter stream* which would generate our spliced input[32]. For this, we would need to map the model back to its parameter stream which can be quite difficult. The reason is that we would need to predict (based on the model) which feature decisions are made at which point in the generator and which bytes need to be written where to create our input. This is not trivial because generators can be highly recursive and also can represent certain input features only implicitly in a stream. At the same time it would be a loss if we would not have the parameter stream of a spliced input. Because with the parameter stream we could give Zest a chance to *further mutate* the spliced input. This could help further explore our targeted branch (or its surrounding region) without additional splicing.

Thus, we have developed a technique to obtain the parameter stream based on the model. It is based on the following observation. JQF's XML and JavaScript generators create their inputs by effectively *traversing* the tree of an input while they generate it. You can see this principle in the pseudocode of our simple XML generator in Listing 3 (page 37). The XML input is generated during a *depth-first* traversal of the input while it is being generated. We leverage this to obtain the parameter stream of a model. Namely, we perform a traversal of the tree with the generator, but instead of making random decisions, we force the generators to make ones based on the given model. For this, we implement the input generation *within* the model. Our generators work like this that they first create a bare instance of each node and then generate their contents by calling a `populate()` method on them. This idea is illustrated in Listing 7. Imagine that we first create a bare root node (similar to line 8 of Listing 7). Then we could call its `populate()` method to create the content of the node and its entire subtree.

[32]Spliced input refers to resulting input after the splicing.

```
1  void populate() {
2      int index = random.nextInt(0, dict.size());
3
4      this.tag = dict.get(index);
5
6      int numChildren = random.nextInt();
7      for (int i = 0; i < numChildren; i++) {
8          XmlNode child = new XmlNode();
9          child.populate();
10         this.addChild(child);
11     }
12 }
```

Listing 7: An XML gnerator which can track create a parameter stream from a model

Implementing the generation within a node object gives us the chance to access the attributes of the node during generation. Typically they would be uninitialized during the traversal. However, we could access the attributes if we would traverse the tree *after* the generation, because they would be set then. Our trick is to store the *decisions* of a generator as *attributes* of each node. This allows us to access the decisions for a node, if we would traverse its subtree again. Importantly, we can instruct the generator to use pre-existing decisions of a node, if there are any. This allows us to control the decisions of a generator from the model level. To achieve this, we always pass a *fallback* value each time we query for a random value. This fallback value is the attribute value of the corresponding decision. If our generators receive an *initialized* fallback value, they know not to read from a pseudo-random byte stream, but to instead *write* (and return) bytes corresponding to the given fallback value[33]. This creates a parameter stream which corresponds to model. You can find an example for this procedure in Listing 8. This example is an adapted snippet of Listing 7 (line 2). In our adapted example, we query for the index in a dictionary but pass an additional parameter. This parameter is the value of the index during the original generation of the node. During the original generation, it would be uninitialized. Accordingly, the generator would sample a random value. However, if we would traverse tree afterwards, then the attribute would be set (see line 2, Listing 8). We would thus pass an *initialized* value. The "random"-object would thus not read and return a pseudo-random value but instead write and return a value corresponding to the attribute. This allows us to create a parameter-stream from a given model. Because by manipulating the attributes of a node, we can now simultaneously influence the generator. To create the parameter stream for a changed model, we would simply have to re-run the traversal of the tree and track the parameter stream that the generator writes based on the model.

[33]In the implementation we in fact do not necessarily check whether the fallback value is initialized but also consider if other things are initialized. It is simplified here for brevity.

```
1    int index = random.nextInt(0, dict.size(), this.index);
2    this.index = index;
```

Listing 8: Example for model-based decision control during generation

This technique allows us to obtain the parameter stream for a spliced input. Because all that splicing really does is influence the attributes of affected nodes. We could thus afterwards *re-run* the generation with the current model and obtain the parameter stream to re-create it "from thin air" (i.e., without splicing but only with the parameter stream and generator).

While this technique is practical, it also has its caveats. First, we have to know which bytes a generator has to *write* for a given decision. This depends on how the generator interprets the bytes of a parameter stream. It can be straightforward. For instance, we have seen in Figure 6 (page 38) that integer decisions might be represented by their corresponding byte sequences. In this case, we would simply have to write the byte sequences for an integer to create the corresponding decision on the parameter stream. However, it can also be more complicated than that. For instance, if we deal with *bounded* integer sampling. For example, if we query for integers that all must lie within the interval $[1, 10]$ (i.e., random.nextInt(1,11)). In this case we would have to *know* how the generator converts pseudo-random bytes to values in that interval. Based on that we would have to write a *reversal routine* which returns the parameter bytes we have to write for a given value within $[1, 10]$. This can be rather complicated as generators might operate with non-trivial operations on bytes which might be difficult to understand and to reverse. However, it *can* be manageable. Especially, if the low-level operations are only based on unbounded integer sampling[34]. Nevertheless, we had some difficulties understanding and reversing the sampling of (double-precision) floating point numbers as it relied on querying integers of different bit sizes, partially bit-shifting them and then multiplying them. This made it difficult for us to write a reversal routine for such floating point numbers. To overcome this, we have *discretized* our generators. This means that they only can query for unbounded pseudo-random integers at the low-level. More complex queries (e.g., bounded integers or floating point numbers) have to be built on this integer sampling. This can work reasonably well, however it can limit the precision of floating point numbers. For example, to generate real values in the interval $(0.0, 1.0]$, we use the following sampling: $value = \frac{\mathsf{random.nextInt}(0, \mathsf{SAMPLING\_RANGE})}{\mathsf{SAMPLING\_RANGE}}$, where SAMPLING_RANGE is an integer greater zero. You can see how the precision of the values is influenced by the SAMPLING_RANGE. In our experiments, the SAMPLING_RANGE was set to 1000. This limited precision is definitely a downside compared to the normal sampling where we likely have more flexibility regarding that. On the other hand, it allows us more easily to identify which values we have to write on the parameter stream to create a given floating point number. That is because each floating point number can be mapped back to a corresponding integer.

[34]For example, producing integers within a bounded range might first query an unbounded integer and then use modulo and addition to move the unbounded integer into that range.

The second caveat of our model-based parameter stream generation is that splicing might produce features which actually can *not* be produced by the generator. Therefore, we also can *not* create a parameter stream for the model. For instance, imagine that you have an XML generator which allows created XML documents to have at max 100 different tag names (but potentially more tag instances). Now imagine, that you have an input which exhausted that limit and that you splice a structure to it which contains *new* tag names. This could create a model which contains more than 100 different tag names. That however is an input which could *not* be created by the generator. In such cases, our generators will still attempt to build an input which *can* be created by the generator. Namely, by replacing the new tag names with existing ones from the model. This will keep the number of unique tag names within the threshold[35]. However, it might *change* our original pattern. This is a trade-off that we accept. It might change certain features, but it still could keep other features of the spliced structure intact. Thus, we could nonetheless produce inputs which are closer to triggering our target branch compared to randomly mutated ones. A similar concern is that you often have to mind max tree depth constraints when splicing. This means that generators often only allow trees with a fixed maximum depth. That constraint however can often be checked more easily before the splicing so that we do not have to prune our spliced tree afterwards.

The integrated approach In the previous sections we have discussed several basic components of our approach. Now we will combine them and present a more formal, integrated description of our technique.

In essence, our approach begins as a normal Zest campaign. Only at a certain point does our approach begin to target rare branches via splicing. Namely, when it detects an (incoming) performance plateau with our path discovery ratio heuristic. If that is the case, we switch to the targeting stage. In this stage we individually select rare branches to target. The rarity of a branch is determined according to FairFuzz's [LS18] (or our updated) heuristic. If we have selected a branch, then we take a group of inputs which hit that branch and perform sequential pattern mining on their feature paths. The inputs we mine on are recorded throughout the entire duration of a fuzzing campaign. They are stored in memory and in their parameter stream representation[36]. We limit the number of inputs we store per branch to 5 to control the memory overhead.

After the mining, we reconstruct the pattern trees from the pattern sequences we obtained as a result. This might produce *several* distinct pattern trees. Each pattern is referred to as a *candidate*. We splice each candidate to 100 randomly selected (mutated) inputs of Zest's queue and observe how often they manage to trigger the target branch. In the end we only keep the pattern which has the best target hit to attempts ratio.

[35] In fact, this example is based on the restrictions that JQF's JavaScriptCodeGenerator has with regards to unique identifier names. We described it in the context of XML because most of our examples are based on XML.

[36] Before we mine on them, we create their models with the generators. We store their parameter stream representation as it arguably costs less memory than their generated model.

The resulting pattern is now spliced for at max 10 minutes (as seen from the targeting start). For this, we let Zest mutate the inputs it has stored in its queue and then splice to the mutated input. However, we do *not* splice to each input. Instead we have a splicing probability. This probability is based on the number of shared *characteristic branches* between the inputs we have mined on and the current parent input[37]. Let us (for now) refer to the inputs we have mined on for a single branch as its *mining inputs*. We define the characteristic branches of a pattern as branches which are shared by the mining inputs but which are not also shared by other inputs. For this, we first obtain the branches which are shared by the mining inputs and then obtain the branches shared by a random sample of other inputs. Then we remove the branches which are shared both by the mining inputs and the random sample. The remaining set of branches is referred to as the *characteristic branches* of our pattern. If we perform splicing, we first check how many of the characteristic branches are contained in the path trace of the current parent input. The ratio of contained branches (compared to all characteristic branches) gives us the probability with which we will we splice to the current input.

The intuition behind this is to make the splicing probability dependent on the similarity between a parent input and our mining inputs. The more similar the inputs are (concerning their branches), the higher is the splicing probability. Our goal is to focus the overhead of splicing on inputs which have a higher probability to trigger the target branch after the splicing. If we splice to inputs which are completely unrelated to our target, then we might have a lower chance to trigger the target branch and the trade-off between splicing attempts and target hits might be lower.

To illustrate our approach more compactly and to compare it against Zest, we provide the (simplified) pseudocode of the Zest in Algorithm 1 and our adapted version in Algorithm 2. The additions of our approach can be found in the lines 10 - 17 and 25 - 30 of Algorithm 2.

There are some other details one can mention. First, we currently only target *valid* branches. This is because Zest has been specifically developed to better explore valid processing stages of an SUT [PLS+19c]. Thus, we consider it best to focus on valid areas, just like Zest. Secondly, we stop targeting early if the number of target hits crosses a *non-rarity threshold*. We do this because some patterns increase the target hit counts of a target quite rapidly. If the hit counts are greater than the median of all valid hit counts, we consider the target branch not rare anymore. In this case, we immediately target the next branch so that we can hopefully explore more branches in a shorter amount of time.

Finally, we want to discuss some details about the pattern mining. The relative *minsup* value we require is 1.0. Therefore, the pattern mining algorithm will only return subsequences which are shared by *all* inputs. The reason is that we assume that input features for a certain branch must be shared by all inputs which hit that branch. Thus, it would not make sense that we e.g., also return patterns which are only present in 70% of inputs.

[37] A parent input is an input of Zest's queue which has been mutated to generate a new one.

Algorithm 1 The Zest Fuzzing Loop

```
 1: queue ← INITINPUTQUEUE()
 2: coverage ← INITCOVERAGE()
 3: generator ← GETGENERATOR()
 4:
 5: while ¬TIMEOUT() do
 6:     parameterStream ← queue.nextInput()
 7:     parameterStream ← parameterStream.mutate()
 8:
 9:     input ← generator.createInput(parameterStream)
10:     runCoverage ← RUNINPUT(input)
11:
12:     if coverage.doesNotContain(runCoverage) then
13:         coverage.update(runCoverage)
14:         queue.store(parameterStream)
15:     end if
16: end while
```

Crash splicing In addition to our approach of targeting rare branches with splicing, we have also developed an approach to investigate the validity of pattern mining for debugging purposes. This technique mines on inputs which trigger the same crash. The idea is to identify a feature pattern in those inputs and to employ the learned pattern. Either to understand the circumstances under which an error occurs or to produce more inputs which trigger that crash. The second application can be particularly useful to investigate the robustness of fixes for a crash. To uniquely identify a crash, we produce a hash of its stack trace. Our stack trace hashing is based on Zest's one but it is more restrictive. Zest hashes the *entire* stack trace. Our approach only hashes the top 3 elements of the stack trace and the exception class. We will thus likely discover fewer unique crashes because we consider less details when hashing. The reason is that Zest's hashing finds many allegedly unique crashes which in fact seem to be identical except for some details lower in the stack trace. For this reason, we deduplicate more restrictively to be more sure that unique crashes point indeed to unique bugs. This deduplication is also used in the analysis of found crashes for our targeting approach.

Other than that our approach is quite simple. We only perform pattern mining once, namely at the start of a campaign. Our algorithm performs the mining on inputs which previously triggered a particular crash. Once the mining is done, we reduce the obtained pattern trees to a single one, similar to our rare branch technique. Namely, we check which pattern tree is most effective at triggering the target bug and only keep that pattern. From then on, we have several variations. For instance, one implementation performs a Zest campaign and splices the pattern with a (fixed) probability to inputs created by Zest. Another implementation splices to *each* input. However, this time it does not also perform a Zest campaign, but only splices to given inputs from a previous fuzzing campaign. These implementations are purposefully simpler because we only

Algorithm 2 Zest with Targeted Splicing

```
1: queue ← INITINPUTQUEUE()
2: coverage ← INITCOVERAGE()
3: generator ← GETGENERATOR()
4:
5: while ¬TIMEOUT() do
6:     parameterStream ← queue.nextInput()
7:     parameterStream ← parameterStream.mutate()
8:
9:     input ← generator.createInput(parameterStream)
10:    if CURRENTLYTARGETING() then
11:        pattern ← GETCURRENTPATTERN()
12:        probability ← GETSPLICINGPROBABILITY(pattern, input)
13:        if RANDOMDOUBLE(0, 1) ≤ probability then
14:            input ← input.splice(pattern)
15:            parameterStream ← input.buildParameterStream()
16:        end if
17:    end if
18:    runCoverage ← RUNINPUT(input)
19:
20:    if coverage.doesNotContain(runCoverage) then
21:        coverage.update(runCoverage)
22:        queue.store(parameterStream)
23:    end if
24:
25:    if SHOULDTARGETNEWBRANCH() then
26:        rareBranch ← GETRAREBRANCH()
27:        inputList ← GETINPUTSFORBRANCH(rareBranch)
28:        pattern ← MINE(inputList)
29:        SETTARGETWITHPATTERN(rareBranch, pattern)
30:    end if
31: end while
```

want to get a rough idea of how much potential pattern mining and splicing can have for debugging purposes. Unfortunately we were not able to evaluate them due to time constraints.

4.3. Implementation

Our approach has been implemented in Java as an extension of JQF/Zest[JQF][38]. It is implemented within an IntelliJ IDEA project (version 2021.1.2 - Community Edition). We use Maven (version 3.8.1) as a build system. The development took place with the Java SDK on version 11. The development operating system was Windows 10. We make our implementation, experimental data and scripts available as a (Humboldt University internal) GitLab repository[39]. Access is granted upon request. We can also supply you with a replication package on request. This will not require access to the Humboldt University GitLab.

We have made two implementation attempts. The first attempt is located in a project called `jqf-feature`. Our current implementation resides in a project called `jqf-crash`. The first attempt did not store inputs for mining in memory. Instead, it exported the feature paths of *each* valid input to disk and only loaded them for mining as needed. This however can generate a lot of data. To control this, we have set a limit for the disk space usage. The downside is that we can quickly exhaust that limit and thus not export further data at some point. We attempted to mitigate this by limiting the disk usage in stages. Nonetheless, this introduced additional complexity which sometimes made it difficult to know which data has been written to disk and which not.

Another downside of our original implementation was that our generators were not capable of producing parameter streams for spliced inputs. Therefore, we could not further mutate them via Zest. Finally, we also attempted graph-based (versus sequential) pattern mining on the original project. However, this again introduced some complexity as we did not use a graph mining algorithm for directed graphs[40]. Thus, we had to augment our inputs in particular ways to reconstruct necessary relationships from the undirected graph mining results. Due to the above difficulties, we decided to re-implement our approach when investigating the feasibility of splicing for crash debugging. This is why the project `jqf-crash` now contains both the implementation for rare branch targeting and the implementation of splicing for crash reproduction.

In the following we will *only* describe the implementation and usage of `jqf-crash`. The project `jqf-feature` will neither be further discussed nor evaluated. However, its source code is made available in our repository for reference.

[38]The JQF version is 2.0. at the commit `71b3eac88029ee76dc95ced2edd9ed9fde73f4ea`
[39]https://gitlab.informatik.hu-berlin.de/krausrom/feature-fuzzing
[40]As we had some difficulties finding an implementation for directed graphs which is peer-reviewed and would suit our needs.

Implementation overview One can distinguish two main components of our implementation. The fuzzing framework and the generators. The fuzzing framework contains all classes we have written to adapt JQF/Zest to our needs. This includes several `Guidance` classes which implement different components of our fuzzing algorithms. An overview of the fuzzing framework classes is given in Figure 28 (page 103).

You may recall that Guidances are the classes which implement the core mechanics of a fuzzing algorithm in JQF. All our Guidances are built on (i.e., extend) the ZestGuidance (which is JQF's implementation of Zest). Some of them only add additional tracking of data. Others impact Zest's procedure more drastically. For instance, we have written a HitTrackingGuidance which can track (and export) how often certain branches have been hit. This is not something which Zest normally performs, however it is relevant for our approach (e.g., to identify rare branches) and for our evaluation. The HitTrackingGuidance realizes that by processing the runCoverage object provided by the ZestGuidance. In all case that we ran experiments for Zest, we actually ran the HitTrackingGuidance. This is because the HitTrackingGuidance is a simple wrapper which only collects additional metrics. It does not change the processing of the ZestGuidance in any other way.

The InputTrackingGuidance is a Guidance we use to store up to five Zest inputs for each valid branch we observe. It is an extension of the HitTrackingGuidance (and thus inherits and *keeps* its hit tracking functionality). The input-storing is realized in a "first-come first-serve" manner. This means that we store the first five inputs we observe for each given branch. Further work might e.g., put more focus on a higher variability in the set of inputs we store per branch. We store inputs as a preparatory step so that we could later mine on them (see below). The inputs are stored as SpliceInputs. This is an extension of the class which the ZestGuidance uses to represent parameter streams (namely, the class LinearInput).

Finally, we have the FeatureGuidance. This is the Guidance which implements our targeted pattern mining and splicing approach. It extends the InputTrackingGuidance. Therefore, the FeatureGuidance inherits both the hit tracking capabilities of the HitTrackingGuidance and the input storing of the InputTrackingGuidance. Once the FeatureGuidance decides to perform a targeting, it selects a rare branch according to the FairFuzz rarity cutoff[41] [LS18] or our adapted heuristic. It then uses the functionalities of the InputTrackingGuidance to obtain previous inputs for these branches to perform pattern mining on their tree paths. The pattern mining is realized with a PatternMiner instance. This class performs sequential pattern mining (see below) and reconstructs the feature trees from the sequential patterns according to our heuristics. The reconstructed tree patterns are represented with FeatureGraph objects. This is an immediate representation which is later converted to input-specific model tees when it is being spliced. That conversion has to be implemented by the corresponding generators[42]. A

[41] Our implementation of the FairFuzz rarity cutoff is in fact also slightly adapted. If the FairFuzz rarity cutoff is only 2 branch hits, we arbitrarily set it up to 10. The reason is that a cutoff value of 2 limits us to mine on at max two inputs per branch. This could lead to patterns which could be less reduced than if we mined on more inputs. For this reason, we make this arbitary increase.

[42] In fact, it is realized by the inputs themselves. This is however almost equivalent in our case,

FeatureGraph consists of FeatureNodes. Each FeatureNode is an abstract representation of a tree-model node. It has a type and a payload (both Strings). All our inputs (i.e., JavaScriptDocument and XmlDocument instances) must be able to convert their models to a FeatureGraph and back again. We enforce this with an interface they must implement (i.e., FeatureInput)[43]. Since FeatureNodes only consist of Strings, we can encode each path in a FeatureGraph as a sequence of strings. These sequences are then used as input for sequential pattern mining.

The sequential pattern mining is realized with the SPMF [FVLG+16] implementation of the CM-SPAM algorithm [FVGCT14, FV22]. SPMF is a Java framework for pattern mining. Our implementation only includes the SPMF-classes necessary for CM-SPAM to our project. The SPMF version we use is 2.53. The CM-SPAM algorithm has been chosen as it mines for closed patterns and it allows us to set gap constraints. This eliminates many sub-patterns from the results list and it allows us to only obtain path patterns which form are strictly contiguous sequences of nodes. Our relative *minsup* value has been set to 1.0. This means that we will only obtain (sub-)paths which are shared by *all* inputs. The gap constraint has been set to 1 (which means *no* gap in the CM-SPAM implementation of SPMF). This enforces strict contiguity for the components of a pattern. Without that we could find node sequence patterns which are *not* strictly contiguous. These would not be real (sub-)paths of a tree model[44] and are thus not interesting to us.

Once we have performed pattern mining and obtained the reconstructed pattern trees, we can employ them for splicing. The splicing is implemented by each input class *individually*. Meaning, we have a separate implementation for our JavaScriptDocument and XmlDocument classes. One could possibly think about building an abstract tree input class as a basis for these objects. This could combine the implementation of tree-based splicing into one. However, we found it difficult to realize within the scope of this project.

The splicing is realized by calling a splice(FeatureGraph ...) method on an input (i.e., either on an XmlDocument or JavaScriptDocument). The call is performed by the FeatureGuidance if it has obtained a new input from Zest and decides to splice a mined pattern (i.e., a FeatureGraph) to it. If the splicing succeeds, the FeatureGuidance calls a getParameterList() method on the input to obtain the parameter stream for the updated model. The input then attempts to reconstruct the parameter stream necessary to generate it in its current state (i.e., after the splicing). The FeatureGuidance could then store this stream in Zest's queue for further processing should the spliced input e.g., uncover new branches.

In addition to these Guidances and other classes, we have also implemented a prototype which can be used to (ideally) reproduce crashes with pattern mining and splicing. We will only briefly mention them as were were unfortunately not able to

because the generation routines are mostly embedded within the classes of the inputs.

[43] You can see how this interface relates to our inputs in Figure 31, page 106.

[44] I.e., they would not correspond to strictly contigous sequences of nodes in the underlying tree.

perform a thorough evaluation of this use case. You can find the classes for crash splicing also in the class diagram for `jqf-crash` (Figure 28, page 103). Again, all our Guidances extend the ZestGuidance as we also base this approach on Zest. The first class is the CrashTrackingGuidance. We use this Guidance to collect several inputs per (deduplicated) crash and to collect metrics on each crash (e.g., how often it occurred and its number of unique path traces). Next, we have the CrashSplicingGuidance. This Guidance can learn the patterns for a set of crashing inputs and then attempts to re-generate the crashes via splicing while running a Zest campaign. It extends the CrashTrackingGuidance so that it inherits its crash metric collection. Finally, we have the NoExpCrashSplicingGuidance[45]. This is an adapted version of the CrashSplicingGuidance developed specifically for evaluation purposes. This Guidance does *not* perform a Zest campaign alongside splicing. Instead it splices to *each* (mutated) input of a given input list. Our comparison baseline would have been the CrashMutatingGuidance. This Guidance also obtains a set of crashing inputs, however it does not perform splicing but simply mutates their parameter streams to ideally produce more crashes. The original goal of our evaluation was to see whether splicing can be more effective at re-generating crashes with more varied path traces compared to simple crash mutation. Unfortunately we were not able to fully perform these experiments. Preliminary experiments indicate that crash splicing could work, however it does seem (considerably) less efficient than crash mutation. At least for our analyzed metrics.

The second major component of our implementation are the generators, namely the XmlTrackingGenerator and JavaScriptTrackingGenerator. They create instances of type XmlDocument and JavaScriptDocument respectively. These are classes which represent XML and JavaScript documents as tree-based inputs. You can see (partially reduced) class diagrams for our XML classes in Figure 29 and for JavaScript in Figure 30 (pages 104 and 105).

The main difference between ours and JQF's generators is that our generators store and produce an object which contain an *explicit* tree-based representation of the generated inputs. JQF's generators only partially build explicit tree-based models[46]. Furthermore JQF's generators always return a String as a result. However, to perform tree-based splicing we actually need explicit tree-based models which furthermore need to support splicing. Moreover, our spliced models need to be converted back to parameter streams so that we can use them for Zest mutation. We enforce these functionalities in with the FeatureInput interface which generated inputs must implement. You can see how this interface relates to our XmlDocument and JavaScriptDocument inputs in Figure 31 (page 106). The downsides of our custom generators are that they are considerably more complex which likely negatively impacts the throughput (i.e., the generated inputs per time). Furthermore, they perform a discretization of all random operations. Therefore, e.g., our randomly generated doubles only have a precision of up to 10^{-3}.

[45] "NoExp" stands for "No (Zest) Exploration"

[46] Namely, JQF's XmlDocumentGenerator does so, while the model is only implicit in the JavaScriptCodeGenerator

Usage To run `jqf-crash` you need to have its packaged `jar`-file. You can obtain it when building `jqf-crash` with the "`mvn package`" command within its project directory (once it is installed). Installation instructions for `jqf-crash` will be provided with our repository[47]. Pre-built images of `jqf-crash` can also be found in our repository. You will find them as `jqf-crash-1.0-SNAPSHOT-cli.jar`.

To run `jqf-crash` you need an SUT and a test driver. We recommend that you set up an IDE project (e.g., with the IntelliJ IDEA) for the SUT and load it (and `jqf-crash`) as a dependency. The test driver must be a Java class which is annotated with @RunWith(CrashJQF.class). This will run the test with `jqf-crash`. You will also need a test method within the test class. This method should take as a parameter the inputs which our (or JQF's) generators will create. The generators are specified with the @From(NameOfTheGenerator.class) annotation before the parameter. Within the test-method you should then initialize your SUT and run the created input on it. This method will later be used by `jqf-crash` to run inputs on the SUT. You can find examples of this entire setup in the "Software/test-subjects" folder of our repository for each of our SUTs.

We advise you to create a `.jar` package which contains your test driver the dependencies it needs (e.g., the generators you employ). You can the run `jqf-crash` as follows:

```
java -jar ./jqf-crash-1.0-SNAPSHOT-cli.jar ./test_driver.jar TestClass testMethod
```

The first parameter is the `jqf-crash` build that we want to run. Next comes the packaged `jar` of our test driver (`test_driver.jar`). Afterwards comes the name of the test driver class (*without* ".java" at the end) and finally the name of the test method.

The above call will start `jqf-crash` with the CrashTrackingGuidance. You can use options to change the selected Guidance. For instance, you can add the option `-f` to run targeted crash splicing with our FeatureGuidance. To run Zest with our HitTrackingGuidance employ the option `-z`. These are only some of the available configurations. You can find an overview of the options if you run `jqf-crash` without any parameters. Meaning, if you call:

```
java -jar ./jqf-crash-1.0-SNAPSHOT-cli.jar
```

Further examples can be found in our experiment scripts (which can be found in the Data folder of our repository, once you extract the archive of an experiment).

Analysis Scripts To perform the plotting of data and statistical analyses we have written several Python 3.7.9 scripts. These can be found in the "Software/scripts" folder of our repository. The purpose of these scripts should be mostly evident from

[47]This implicitly includes the replication package.

their names. All parameters are specified within the scripts themselves. Thus, you have to update the scripts and re-run them to e.g., plot data for different SUTs. The plotting and statistical analyses are conducted with the Python libraries Matplotlib (version 3.5.3) [Mat], NumPy (version 1.19.4) [Num] and SciPy (version 1.7.3) [Sci]. The employed versions can also be found in the "requirements.txt" file of the "scripts" folder. We advise you to create a virtual environment with Python to run these scripts.

Patch files In addition to our `jqf-crash` sources, you will also need patch files for JQF. This is because we require access to certain attributes of `ZestGuidance` objects which are otherwise not available. Furthermore the patch files fix a crash which apparently can occur when the ZestGuidance forces an export of data but the last export is less than 1 millisecond ago. You will find two files in our repository. The ZestGuidance.java file is our adapted ZestGuidance. The adapted janala.conf file instructs JQF to exclude our packages from its instrumentation. You will need to overwrite these files in your local JQF installation and re-build JQF to run `jqf-crash`. Our pre-built jqf-crash-1.0-SNAPSHOT-cli.jar archives should work out of the box (i.e., without these additional preparations). The patch files are located in the repository at "Software/patch-files".

5. Evaluation

In the following, we will discuss the evaluation of our approach. The discussion will encompass the design of our experiments, the presentation and analysis of our results and finally the discussion.

5.1. Evaluation Setup

Subjects For our test subjects we have oriented ourselves on the Zest [PLS+19c] evaluation. Namely, we evaluate our approach on the following benchmarks:

- Apache Maven [Mav] (version 3.5.2)
- Apache Ant [Ant] (version 1.10.2)
- Google Closure [Clo] (version v20180204)
- Mozilla Rhino [Rhi] (version 1.7.8)

Maven and Ant are build systems which process XML inputs. Closure and Rhino are a JavaScript compiler and an implementation of JavaScript in Java respectively (however, our test driver for Rhino also performs a compilation of JavaScript). The versions that we employ are the same as in the original Zest evaluation [PLS+19c].

These benchmarks are four out of the five SUTs which Zest was originally evaluated on [PLS+19c]. We do not include BCEL [BCE] as its generator does not have a similar, easily recognizable tree-based structure as JQF's XML and JavaScript generator. Future work might explore whether our approach could also be extended to this benchmark.

Generators To create our inputs we have employed our custom XML and JavaScript generators, namely the XmlTrackingGenerator and JavaScriptTrackingGenerator. We do this as we require inputs which have an explicit tree-based model and implement functionalities for splicing. This would not the case with JQF's generators. We use our generators both for the experiments with our approach and with Zest. The reason we also use it for Zest is that we observed different branch-ID mappings when employing different generators. For this reasons we employ the same generators in both experimental conditions.

The XML generators require dictionaries of keywords to populate the contents of their models (e.g., for tag names). We have employed the dictionaries provided by JQF for Ant and Maven respectively. They were used in all experimental conditions.

Data Collection To evaluate our approach we have performed several fuzzing campaigns to compare our technique to Zest. We analyze the same data set to investigate all of our research questions. Our data collection consisted of two campaign sets. One preparatory set and one to compare our approaches.

The preparatory sets consist of 20 Zest fuzzing campaigns we have performed on each SUT. Every campaign had a duration of 12 hours. The goal of these campaigns was to establish which branches can be "typically" considered rare. It would be these branches for which we would later compare the branch hit counts and unique path traces between our approach and Zest. The number of repetitions and the duration has been chosen to reflect our later evaluation setup which compares our approach to Zest.

We have identified the valid rare branches of each campaign at 1 hour, 6 hours and 12 hours. We sample the branches at different points in time as the rarity of a branch could change over time. Since we could perform targeting over the entire course of a campaign, we have to consider rarity at different time points. Our sampling points have been chosen to reflect the start, middle and end of a campaign. The validity of a branch has been established according to the ZestGuidance. To collect the branch hit counts, we have performed the campaigns with our HitTrackingGuidance as it is a wrapper of JQF's ZestGuidance for precisely that purpose. The hit counts have been tracked by analyzing Zest's runCoverage. Rarity has been established according to the implementation of the FairFuzz [LS18] heuristic in our approach. This has been realized with a Python script (`get_rare_branches.py`) after the campaigns. The script unifies the rare branches of a campaign into one set (i.e., it unifies the rare branches of different time points). Afterwards it unifies the rare branches of all campaigns of one SUT into one set. This gives us the rare branches for this SUT. One could think that it would be better to calculate an intersection of the rare branches of different campaigns. However, we have found that it can reduce the set of rare branches to 0 on *all* SUTs. Apparently, rarity can vary considerably between different campaigns. Possibly, the results would be different if we had more fine-granular sampling points. Nevertheless, to account for this apparent variability, we calculate the union of the rare branches of

different campaigns. This lead to the following number of rare branches:

- Ant: 484 (26%)
- Closure: 3583 (21%)
- Maven: 274 (14%)
- Rhino: 241 (4%)

The percentages reflect the proportion of valid rare branches to all valid branches that we have analyzed for these SUTs.

After these steps we have run our evaluation experiments. These also consisted of 20 Zest fuzzing campaigns per SUT. Each campaign had a duration of 12 hours. The 20 campaigns have been performed to account for the variability due to the random nature of fuzzing. It is the same number of trials as in Zest's [PLS+19c] evaluation and the one of FairFuzz [LS18]. However, it falls short of Klees et al.'s [KRC+18] 30 repetitions. The duration is four times as long as Zest's original 3 hour runs. However it also falls short of Klees et al.'s [KRC+18] suggested 24 hour durations. We have chosen smaller durations due to time constraints. Further research could investigate longer durations.

The campaigns for our approach have been performed with the FeatureGuidance. The campaigns for Zest have been performed with the HitTrackingGuidance to collect additional performance metrics which the ZestGuidance does not provide (see below).

Analyzed Metrics To analyze RQ1, we collect the hit counts for each valid and invalid branch. Validity is established according to the ZestGuidance. The counts are tracked within the HitTrackingGuidance. Our FeatureGuidance extends the HitTrackingGuidance. Therefore, our approach inherits the hit tracking from there. Branch hit counts are counted once per input. Meaning, if a single input hit a branch several times it is still only counted as one hit.

In addition to the hit counts, we also analyze the number of unique coverage traces. This is done by tracking the coverage hashes of inputs which hit a rare branch of our preparatory campaigns. The hashing is performed with a method provided by Zest's runCoverage object. The hash should consider both *which* branches have been hit and *how often* (within a single trace).

To investigate RQ2 we analyze the data provided by the plot_data file produced by the ZestGuidance[48]. This includes the number of total and valid branch hits. Additional metrics are e.g., the number of discovered paths and generated in/valid inputs.

[48]It is also produced for our campaigns as each Guidance we run extends the ZestGuidance.

For the analysis of RQ3 we track and export data for each branch targeting within the FeatureGuidance. All metrics are tracked individually for each branch targeting and only refer to one particular targeting[49]. They include the

- **Splicing attempts**: The number of times it was attempted to splice while targeting a given branch
- **Splicing successes**: The number of times that we succeeded at splicing a substructure to an input
- **Splicing (valid) target hits**: The number of spliced inputs which hit the branch (separately tracked for valid and any coverage)
- **Splicing valid inputs**: The number of spliced inputs which produced valid coverage
- **(Valid) Target hits at start**: The number of target branch hits before the first splicing (separately tracked for valid and any coverage)
- **(Valid) Target hits at end**: The number of target branch hits after the last splicing (separately tracked for valid and any coverage)

This base data is later analyzed in different combinations to investigate how often splicing apparently succeeded at re-generating our target features and how effective the patterns seem to be at hitting our target. The FeatureGuidance exports this data in a file per fuzzing campaign (splicing_events.csv).

Statistical Analysis To analyze whether our observed differences in RQ1 and RQ2 are statistically significant we perform Mann-Whitney U tests with $\alpha = 0.05$. This particular test and alpha level can often be found in fuzzing research [HGM+21, NNTH+21, MKC20]. Its advantage is that it does not make an assumption on the distribution of the (fuzzing) algorithm performance [KRC+18].

Hardware All experiments were conducted on compute servers of the Humboldt University of Berlin with an *Intel(R) Xeon(R) CPU E7-4880 v2, 2.50GHz* CPU, 1 TB of main memory, running on *openSUSE Leap 15.3*.

5.2. Experimental Results

In the following section we will present the results and analysis of our experiments for each research question. For brevity we will often refer to our approach of rare branch targeting with pattern mining and splicing simply as "splicing" (or "splice"). It should be clear from the context whether "splicing" refers the the act of "tree attachement" or to our approach in more general.

[49] A "targeting" means the phase during which we target a single branch with splicing.

RQ1: Can our approach hit rarely visited areas more often and with more varied path traces compared to Zest? To analyze RQ1 let us first look at a the comparison between the mean number of branch hits from our targeted splicing approach compared to Zest. This data is presented in Table 5. It shows the mean total branch hits for each approach and SUT. Furthermore, it shows the quotient of the means between the splicing approach and Zest for each SUT. This allows one to see quickly whether splicing produced a higher mean ("Splice / Zest" > 1) or lower mean ("Splice / Zest" < 1). Significant differences are highlighted in bold. Values are rounded to 5 decimals after the comma. A shown p-value of 0.0 means that it is smaller than 10^{-5}. This holds for all p-values we report in our tables.

SUT	Splice	Zest	Splice / Zest	p
ant	2546529.5	2242306.8	1.13567	0.0531
closure	15465995.5	24107462.7	0.64154	**2e-05**
maven	66358298.35	48804136.0	1.35969	**1e-05**
rhino	1091169.35	611630.15	1.78403	**0.00234**

Table 5: Mean Total Branch Hits

One can see that splicing produced on average higher hit counts for rare branches on all SUTs except for Closure. The differences are significant for all SUTs, except for Ant where it closely misses significance. These results indicate that splicing *can* be effective at increasing the hit counts of rarely visited branches. The increases reach on average from 13% (Ant, not significant) to up to 78% (Rhino, significant). Nevertheless, one can also see that our approach is significantly worse on Closure. The reasons for that could be manifold. First, Closure had much more rare branches than any other SUT[50]. Thus it could be that increases for some rare branches could impact the mean less drastically than with a smaller set. Furthermore, the high number of rare branches could indicate that the conditions to trigger those rare branches could be much more intricate than on other SUTs. Our simple pattern mining approach could thus often fail to correctly detect those patterns and thus not lead to an increase in branch hits.

One of our ideas was to particularly support Zest's fuzzing of valid processing stages. Let us thus now analyze the mean hit counts for valid inputs. This data is presented in Table 6.

You can see a similar pattern to Table 5. The differences in the mean are better for splicing on all SUTs except for Closure. However, this time they are significant in all cases. Furthermore, the relative increases are much higher, with at least 38% on Ant and more than 100% on Maven and Rhino. These results indicate that we can target valid rare branches quite effectively on average. The only exception is again Closure. We assume that the reasons are similar to what we have discussed when analyzing all hits (i.e., not only valid ones).

[50]Closure had 3583 rare branches we analyzed. The next highest number was on Maven with 274.

SUT	Splice	Zest	Splice / Zest	p
ant	1079963.65	779026.4	1.3863	**0.00163**
closure	12411865.7	20501664.95	0.60541	**2e-05**
maven	17802706.0	7774842.35	2.28978	**0.0**
rhino	739115.05	281247.4	2.62799	**0.00012**

Table 6: Mean Total Valid Branch Hits[51]

Finally, let us analyze whether our approach can also produce more unique traces compared to Zest. This data is presented in Table 7.

SUT	Splice	Zest	Splice / Zest	p
ant	57514.35	56763.85	1.01322	0.47348
closure	358226.55	457653.8	0.78275	**0.0**
maven	1380289.3	1136089.3	1.21495	**6e-05**
rhino	322861.1	255072.55	1.26576	**0.00771**

Table 7: Mean Unique Traces

It is evident that splicing was able to significantly increase the number of unique traces on Maven and Rhino for around 20%. There is also a slight increase on Ant which is however not significant. For Closure splicing performed worse. Again, the reasons for that could be the higher number of rare branches and the possible higher complexity of necessary patterns. It is interesting that the increases on Ant are considerably smaller compared to Maven and Rhino and that they are not significant. Possibly it is because Ant can have a slower throughput compared to Maven. This means that we can run less inputs per time. We will therefore splice less often on Ant which might decrease the number of unique traces one can discover in a targeting period. This reasoning is supported by the fact that Ant has the lowest absolute number of rare branch hits of all SUTs (see column "Zest" of Table 5). You can see that we have on average 2 million hits of rare branch on Ant compared to 48 million hits on Maven. This either supports a lower average throughput or suggests that the necessary features for rare branches are more complex on Ant. You can see a similar pattern for the splicing hits (see column "Splice" of Table 5). This time we have on average 2 million hits of rare branch on Ant compared to 66 million hits on Maven. Since we have a higher number of splicing hits on Maven, it is arguably also more likely that splicing will uncover more unique traces. Of course, it could also be that there are less unique traces to find on Ant. But this hypothesis can neither be verified nor rejected based on the data we have. Taken together, the data suggests that splicing can be effective at uncovering new path traces for rare branches, however the relative increases are smaller compared to the hit counts and are only significantly better for half of the SUTs.

RQ2: Can our approach increase the *overall* coverage compared to Zest? To analyze RQ2 let us first have a look at the mean covered branches over time. Zest

refers to the covered branches as "probes". We plot this (and the standard error of the mean)[52] in Figure 23. The metrics are averaged per minute. The black horizontal line in each plot shows the average start time of the first targeting.

The first thing one can notice is that our approach and Zest are quite similar in terms of branch coverage. This indicates that our approach does not incur so much overhead that it drastically reduces the coverage. Nevertheless, on most SUTs our approach either ends with a slightly worse or nearly identical coverage. Only on Ant does our approach visibly (but still slightly) outperform Zest.

Furthermore, it appears that the beginning of the targeting period does mostly not positively influence the coverage. On the contrary, you can see relatively well on Maven and Rhino that the beginning of targeting coincides with a slight downward trend compared to Zest.

Table 8 shows the results of the statistical tests on the final branch coverage per SUT. The "Zest" and "Splice" columns show the average final values.

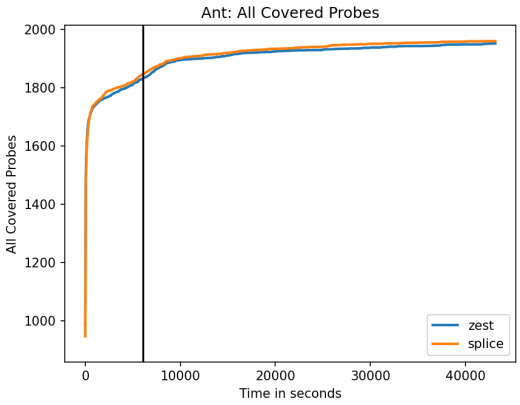
SUT	Splice	Zest	Splice / Zest	p
ant	1959.55	1951.3	1.00423	0.14752
closure	17069.6	17149.95	0.99531	0.2853
maven	2199.0	2198.75	1.00011	0.08056
rhino	6219.0	6232.3	0.99787	0.87105

Table 8: Mean Covered Branches

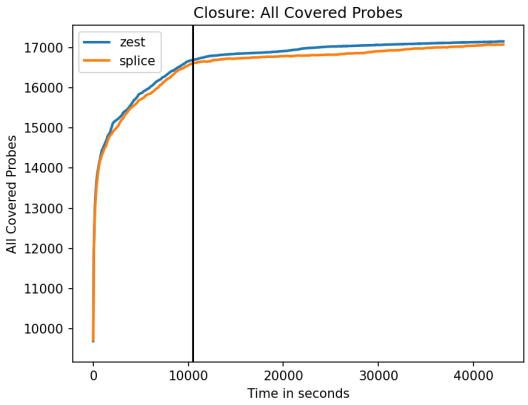
You can see similar results to the plots. The final branch coverage is nearly identical on all SUTs. Splicing only slightly outperforms Zest on Ant and Maven (the XML SUTs). It is slightly worse on Closure and Rhino (the JavaScript SUTs). The differences are not significant in any case. Taken together the results thus suggest that splicing performs similar in terms of branch coverage. The results are somewhat better on the XML bechmarks. The reason could be that the structure of XML files is simpler compared to JavaScript (only 4 node types compared to more than 30). Therefore it could be simpler for our approach to learn effective patterns on XML which could then have positive influences on the coverage. Nevertheless, we see that the positive differences are quite small and also not significant in any case. Moreover, we can also see that our targeting start can introduce a slight downward turn in the branch coverage. This turn is somewhat catched up on Maven, but neither Closure nor Rhino can recover from that. This again indicates that our splicing approach might work better for XML than for JavaScript.

Let us now analyze the mean valid branch coverage. The corresponding plots are given in Figure 24. You can see a similar result situation as before. Splicing performs worse on all SUTs except for Ant. This mirrors the results for the total coverage only that Maven is now also visibly worse. Furthermore, you can again see that the

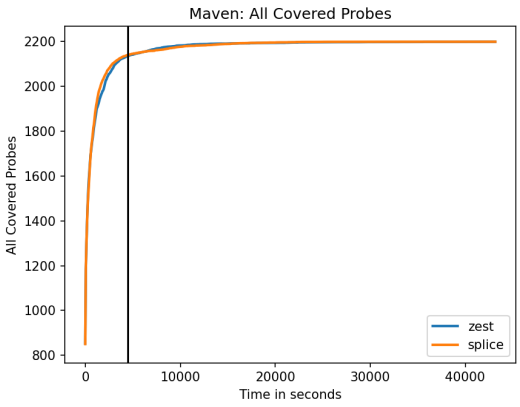
[52]The standard error might be difficult to see. We plot it as a colored, slightly transparent area surrounding the lines. It is visible if you look at the Rhino plot in Figure 26 on page 81.



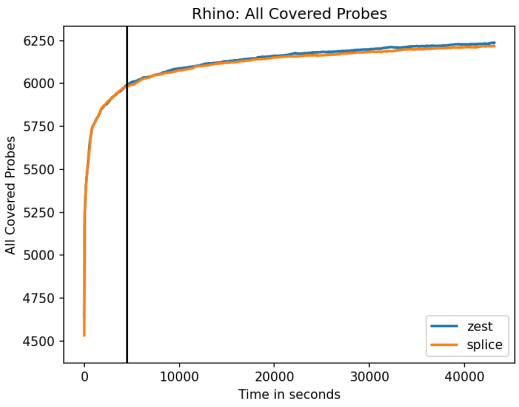
(a) Ant



(b) Closure

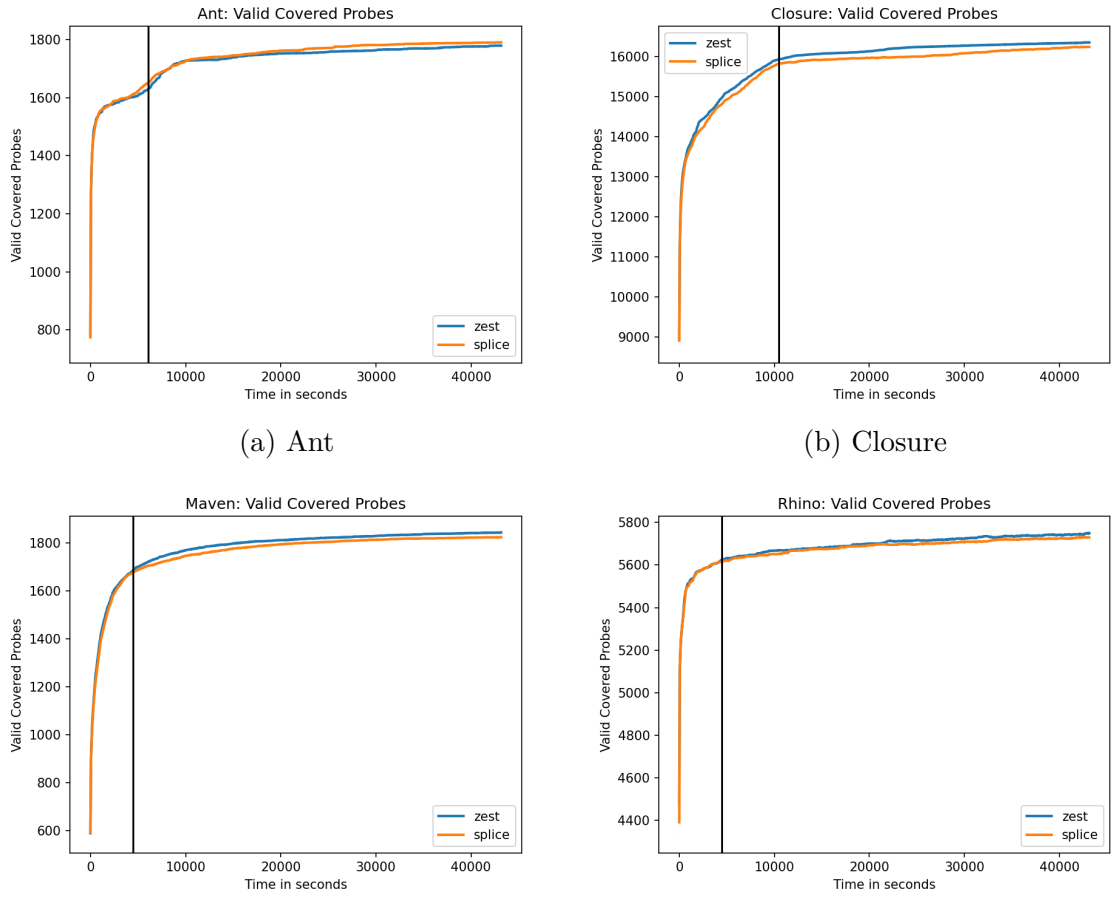


(c) Maven



(d) Rhino

Figure 23: Mean Covered Branches



(a) Ant

(b) Closure

(c) Maven

(d) Rhino

Figure 24: Mean valid covered branches

beginning of the splicing often marks a downward turn in the coverage compared to Zest. Finally, the difference between splicing and Zest seems more drastic compared to what we have seen for the total coverage. This is interesting because we had the reverse situation regarding the rare branches (RQ1). Namely, the branch hits were drastically better for the valid rare branches compared to all rare branches.

Table 9 shows the result of the statistical analysis for the valid covered branches at the end of a fuzzing campaign. You can again see that splicing performed worse on all SUTs, except for Ant. The differences are only significant for Maven. This is interesting as splicing often performed quite well on Maven regarding the rare branch hits (RQ1).

Taken together, one can not say that our splicing approach can increase the total coverage. Neither the overall nor the valid coverage is positively affected in most SUTs. On the contrary, it is often worse. Furthermore, the beginning of the splicing often marks a certain downward turn in the branch discovery rate.

SUT	Splice	Zest	Splice / Zest	p
ant	1791.25	1780.0	1.00632	0.25573
closure	16235.95	16347.85	0.99316	0.08339
maven	1823.85	1843.6	0.98929	**0.01605**
rhino	5732.0	5745.2	0.9977	0.55172

Table 9: Mean valid covered branches

Nevertheless, we want to analyze a final branch coverage metric, namely the discovered paths. The number of discovered paths refers to the number of inputs that Zest stores in its queue at a given time. Zest stores an input in its queue if its coverage trace contains new branches or it managed to increase the hit counts for some branches. Thus the number of stored inputs can be an indicator for the number of discovered "paths" through an SUT.

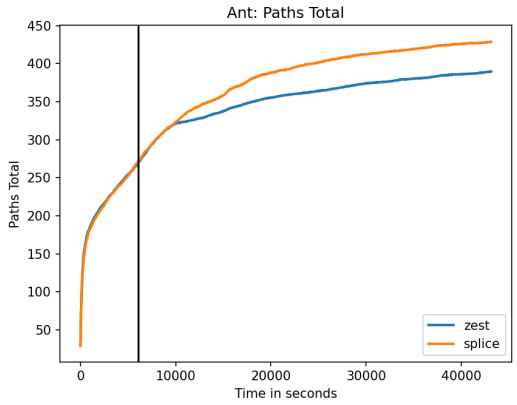
The plots of the mean paths total are presented in Figure 25. This time you can see a more positive picture for splicing. On most SUTs splicing outperforms Zest in terms of discovered paths. Moreover, the average start time of the targeting phase often marks the point where splicing tends to overtake Zest. Only on Closure is splicing noticeably worse compared to Zest.

A similar picture is presented if we look at the results of the statistical analysis in Table 10. The mean paths total at the end of a fuzzing campaign are higher for splicing on all SUTs except for Closure. The differences are significant for all SUTs. However, the relative improvements are this time noticeably smaller compared to the ones we have seen for the rare branch hit counts. This time they lie between 3% and 10% (compared to improvements of up to 100% for the valid rare branch hit counts, see Table 6 at page 74).

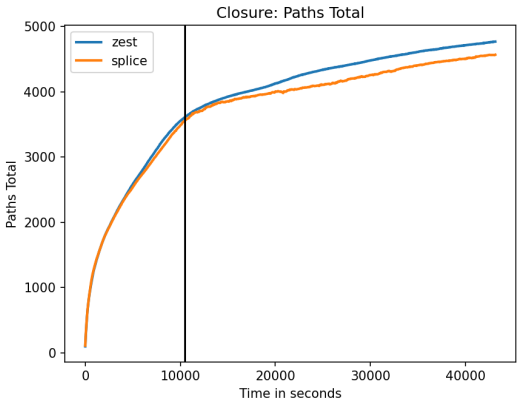
SUT	Splice	Zest	Splice / Zest	p
ant	428.6	389.45	1.10053	**0.0**
closure	4569.7	4771.25	0.95776	**0.00129**
maven	974.3	939.6	1.03693	**6e-05**
rhino	1841.5	1772.35	1.03902	**0.00604**

Table 10: Mean paths total

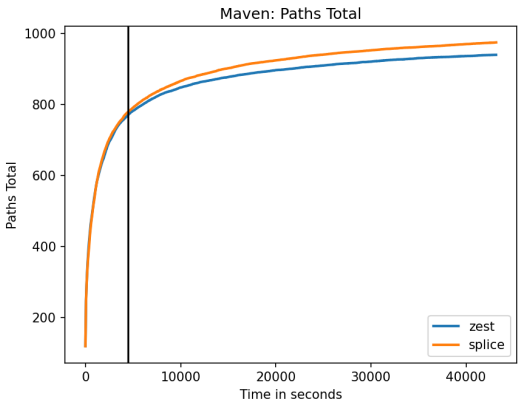
These results are interesting. On the one hand they mirror the observations of RQ1. Namely, improvements on all SUTs except for Closure. On the other hand they indicate positive coverage effects in the number of paths. This is in contrast to our observations for the discovered branches. Taken together it seems that splicing can not be effective at discovering *more* branches, however it might help to explore the branches *more thoroughly*. Meaning, we can produce more diverse paths for our targets. This could explain the positive observations regarding rare branch hits and paths total but the lacking improvements in the overall coverage. It is also supported by the fact



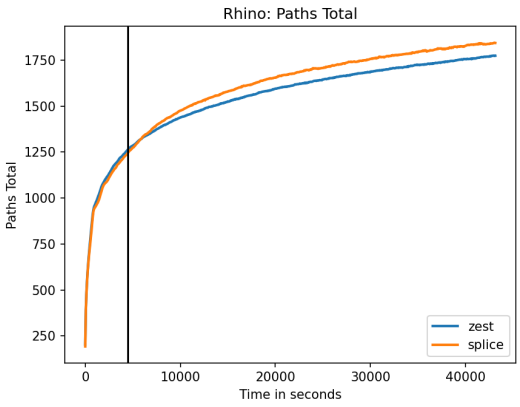
(a) Ant



(b) Closure



(c) Maven



(d) Rhino

Figure 25: Mean paths total

that the unique path traces for the rare branches have been mostly higher on splicing than on Zest (see Table 7, page 74). However one might wonder why the differences for unique rare branch path traces have been significantly better only on Maven and Rhino, whereas total paths are also significantly better for Ant. The reason could be our adapted rarity heuristic. You may recall that we switch to a different rarity cutoff heuristic if we fail to find new targets with FairFuzz's formula. This heuristic is not considered in our analysis of RQ1. Therefore, branches in RQ1 are only categorized as rare according to FairFuzz's cutoff value. It is thus likely that our approach at some point begins to target branches which are not rare according to FairFuzz. This could positively impact the number of discovered paths for these branches, however it would not be reflected in our results for RQ1.

Finally, we want to discuss our performance degradation heuristic. One can see that the mean start of targeting phase coincides reasonable well with the beginning of the performance plateauing on Zest (i.e., with the inflection point where the curves start to flatten). Thus, one could say that our performance degradation heuristic generally worked as intended. However, one should be careful to generalize this heuristic on other SUTs and circumstances because we have built this heuristic based on observations we have made for these SUTs in particular and with Zest.

Number of deduplicated crashes and generated inputs In the following we want to discuss some additional performance metrics which have not been formulated as research questions. These can shed additional light on the impact of our approach.

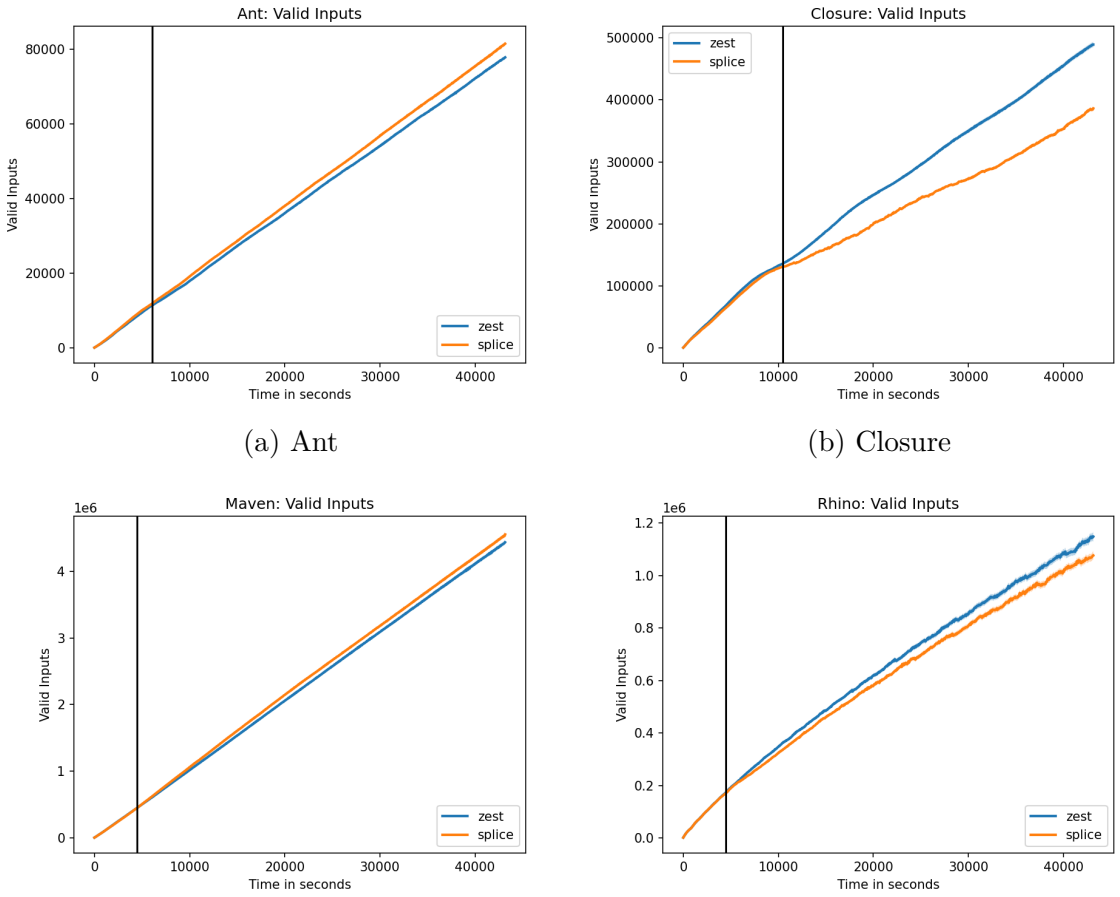
The first metric we analyze is the number of unique crashes found by splicing compared to Zest. The deduplication was conducted according to our more restrictive stack trace hashing. We will not discuss the results according to Zest's deduplication because we consider ours to be a better indicator for the actual number of unique bugs found. Therefore "deduplicated crashes" will in the following only refer to ones deduplicated according to our heuristic.

We have compared the mean number of deduplicated crashes at the end of a fuzzing campaign between splicing and Zest. The results of the statistical test are presented in Table 11.

SUT	Splice	Zest	Splice / Zest	p
ant	1.0	1.0	1.0	1.0
closure	1.95	1.8	1.08333	0.49856
maven	0.0	0.0	nan	1.0
rhino	5.55	5.5	1.00909	0.72292

Table 11: Mean deduplicated crashes

You can see that both approaches perform nearly identical. Splicing tends to find on average more crashes on Closure and Rhino. However, the gains are very small (on



(a) Ant

(b) Closure

(c) Maven

(d) Rhino

Figure 26: Mean Valid Inputs

average less than one bug) and not significant in any case. Thus, one can not say that our approach is in general beneficial for the bug discovery. At least concerning the number of discovered bugs according to our stack trace deduplication and concerning the investigated SUTs.

The next metric we will analyze is the number of generated valid and invalid inputs. This can give us an idea whether our approach can generate more valid inputs compared to Zest and whether we can see any impacts on the throughput (i.e., the overall number of executed inputs during a fuzzing campaign).

The plot of the mean number of generated inputs is presented in Figure 26. You can see that splicing tends to more valid inputs on Ant and Maven (the XML benchmarks). However, splicing tends to be worse on Closure and Rhino (the JavaScript benchmarks). This reflects our previous observation that our approach might work better for XML than for JavaScript (see RQ2). The table for the corresponding statistical analysis is presented in Table 12.

SUT	Splice	Zest	Splice / Zest	p
ant	81469.3	77749.0	1.04785	0.10173
closure	386943.25	489867.35	0.78989	**0.0**
maven	4545513.65	4435413.1	1.02482	0.63594
rhino	1069704.65	1132802.65	0.9443	0.42488

Table 12: Mean valid inputs

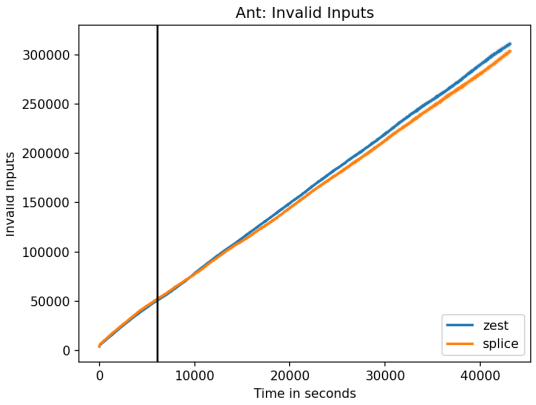
You can see a similar situation to the plots. Our splicing approach produces on average slightly more valid inputs on Ant and Maven. These results are however not significant. Splicing performs worse on Closure and Maven. The difference on Closure is also statistically significant. Thus, one can state that our approach has a certain advantage for XML, however it tends to be worse on JavaScript.

Let us now discuss the invalid inputs. The average number of generated invalid inputs is plotted in Figure 27. Splicing produces less invalid inputs on all benchmarks. Again, the difference is particularly visible for Closure. These impressions are confirmed by the statistical analysis presented in Table 13. You can see that splicing produces on average less invalid inputs on all benchmarks. Again, the differences are only significant for Closure.

SUT	Splice	Zest	Splice / Zest	p
ant	303871.2	311075.95	0.97684	0.56085
closure	316232.55	407030.15	0.77693	**0.0**
maven	26156885.45	27049608.15	0.967	0.18954
rhino	865879.4	892781.7	0.96987	0.71498

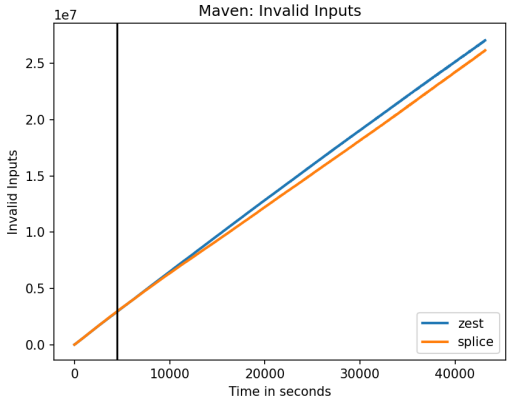
Table 13: Mean invalid inputs

Since splicing produces slightly more valid inputs but slightly less invalid inputs for XML one can conclude that the overall throughput appears to be similar on the XML benchmarks. However, for JavaScript our approach produces less valid and invalid inputs on both benchmarks. This indicates an overall lower throughput. The reason might be that JavaScript has a more complex tree structure which makes splicing more complex. On the one hand, it could take more time to find a suitable node to splice to in an input. On the other hand JavaScript patterns might also require *completion*. This introduces additional overhead which is not present for XML. We are not quite sure why the throughput of our approach is noticeably worse for Closure than for Rhino. A cursory review of our splicing logs indicates that we have on average performed less targetings on Rhino compared to Ant. Therefore, the overhead imposed on Closure would also be more higher.

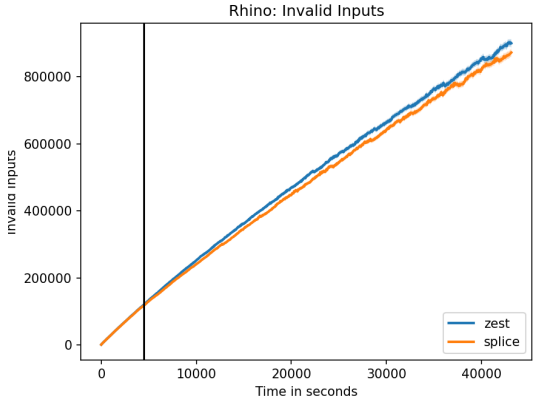


(a) Ant

(b) Closure



(c) Maven



(d) Rhino

Figure 27: Mean invalid inputs

RQ3: How effective is our approach with regards to feature learning and re-generation? In the following we want to better understand how well our splicing approach can learn patterns which increase the hit counts of targets and how often splicing can produce inputs with our desired features.

Let us begin by investigating the feature re-generation. For this let us first analyze the success rate. This gives us the number of times that we could splice to an input divided by all attempts we made at splicing[53], i.e., $\frac{splicing\_successes_i}{splicing\_attempts_i}$ for each targeting i. That metric is a rough estimate for the ratio of produced inputs which should have our target features. The mean success rate, the standard deviation and standard error of the mean[54] are given in Table 14.

SUT	Mean	Std-Dev	Std-Err
ant	0.9857	0.05097	0.00264
closure	0.75487	0.33825	0.01111
maven	0.99811	0.02485	0.00089
rhino	0.81456	0.32377	0.01617

Table 14: Success rate

You can see that splicing succeeds relatively often with a rate of at least 75% on each SUT. Once more, the rates are best for the XML benchmarks. This again supports our previous observation that our approach seems to work particularly well for XML. This is also indicated by the higher standard deviation of at least 30% on Closure and Rhino. This suggests that it can be partially more difficult to splice certain patterns for JavaScript. That could then lead to a higher variation in the success rate because for some patterns the success rate can be quite high while for others it might be lower. It is interesting that the performance is again worst for Closure. This could indicate that the patterns learned for Closure could be more complex than on Rhino. The effect would be that it could be more difficult on Closure to find suitable splicing targets in an input. However, this is only a theory. One would need to analyze the mined patterns more thoroughly to confirm it. A cursory review of our splicing logs indicates that patterns can be quite complex on Closure, however we can also see relatively complex patterns on Rhino.

To further investigate the effectiveness of feature re-generation, let us next have a look at the valid input rate of splicing successes. This metric represents the number of times splicing produced a valid input divided by the number of times that splicing succeeded, i.e., $\frac{splicing\_valid\_inputs_i}{splicing\_successes_i}$ for each targeting i. One can say that it gives us an estimate of the probability that a splicing success produces a valid input. This

[53] We have calculated this fraction for each targeting individually. The reported mean reflects the mean of these fractions.

[54] We report the standard error in most of the following tables to get an idea of how "reliable" our mean is. However, we will often not further analyze it as it is typically rather low.

metric is interesting for us because our goal was to particularly focus on valid processing stages of an SUT. The analysis table for this metric is given in Table 15.

SUT	Mean	Std-Dev	Std-Err
ant	0.28022	0.09883	0.00512
closure	0.43617	0.25536	0.00881
maven	0.23533	0.03727	0.00133
rhino	0.52389	0.25445	0.01325

Table 15: Valid input rate of splicing successes

You can see that the rate is relatively low with barely more than 50% on Rhino and only 23% on Maven. This indicates that our splicing implementation is not particularly effective at generating valid inputs. It is interesting that the JavaScript performance is this time better than for XML. The reason could be because XML also has *content* constraints for validity while validity constraints should be mostly structural on JavaScript. For instance, it could be for Maven that tags with the name "A" are not allowed to have tags with the name "B". This would be an invalid input on Maven even though it would be syntactically valid XML. JavaScript nodes do not typically have "content" in addition to their type and their child nodes. For example, a node for a "while" loop or a "statement" do not have additional metadata content. They only have their child nodes which further specify the subtree[55]. Thus the validity constraints should be mostly structural on JavaScript which should also be more easily achievable by splicing.

In summary, the relatively low validity rate indicates that even *if* our splicing succeeded, that it is far from guaranteed that we will also produce a valid input. This is a downside as our original goal was to especially target valid processing stages of an SUT.

Taken together, the results suggest that splicing succeeds relatively often on average with a rate of at least 75% on each SUT. This indicates that our operation for feature re-generation succeeds most of the time. However it does not tell us whether the splicing produces inputs which *really* contain our desired features. For instance, it could be that our splicing implementation could be somehow faulty so that we do not modify the model in such a way that our wanted features are in fact introduced to our target input. To analyze this, we would need to have some way to check whether the spliced inputs really exhibit the features that we want. Probably, it would be best to analyze this in a separate experiment as it could introduce additional overhead. One option could be to check whether our spliced inputs contain our desired "feature paths" (excluding ones which were present before).

Our current results at least indicate on average a high success rate of splicing. Furthermore, our results of RQ1 indicate positive impacts on the hit counts of rare SUT

[55]This not to say that there is no JavaScript node with metadata comparable to XML. For instance, nodes for "identifiers" do have a freely selectable name (e.g., think of variables). However, many JavaScript nodes do *not* have that, whereas it is the case for *every* XML node.

areas. Thus, there is some reason to believe that we can generate inputs with relevant features. However, the data does not allow to make a conclusive statement on that. The relatively low validity rate shows that splicing can not be very effective at generating valid inputs. But again the results of RQ1 indicate particular benefits for the valid hit counts. Thus, it could be overall not too problematic that we such a relatively low validity rate.

In summary you can thus state that splicing succeeds relatively often with an average rate of at least 75%. However, it does not allow us to conclude that a similar rate of inputs also *really* exhibit the features afterwards. This requires further investigation. The results of RQ1 suggest that splicing can produce inputs which increase the hit counts of rarely visited regions. However, this does not allow one to make conclusions on how effective splicing is at feature re-generation.

Let us next investigate how effective our patterns are. This means to analyze how "well" splicing is able to produce inputs which hit our targeted branches. This question is related to the first part of RQ3, however it is also distinct. Because a target hit does not necessarily entail that the targeted features have been introduced by splicing. Nevertheless, if spliced inputs are noticeably successful at increasing the target hits then there is increased reason to believe that the feature re-generation also might work.

To investigate the effectiveness of our learned features, let us first analyze how many of the targetings[56] had at least one spliced input which managed to hit the target branch. Thus, we divide the number of targetings which fulfill this criterion by the number of all targetings. That gives us the proportion of targetings (per SUT) where spliced inputs fulfilled their goal at least once. This data is given in Table 16. Please note that we do not report the mean or its related metrics as we only analyze a ratio in this case.

SUT	**Ratio**
ant	0.81984
closure	0.49246
maven	0.95903
rhino	0.55335

Table 16: Ratio of targetings where splicing resulted in at least one target hit

You can see that our approach appears fairly successful for XML where at least 80% of the targetings hit the target branch at least once when splicing. This rate is much lower with only 50% for the JavaScript benchmarks. This gives a rough idea of the proportion of patterns which can be deemed "successful". However, one has to be careful when interpreting this data because a targeting is already considered to

[56]Again, a "targeting" refers to the entire phase of selecting a particular rare branch, learning the pattern for it and then splicing it with the goal to increase its hit counts.

be "successful" if at least *one* spliced input hit the target. This neither tells us the proportion of splicing attempts which hit the target nor whether the target hit can be really traced back to the splicing. It could easily be that the target hit is due to other reasons. For instance, because of a randomized pattern completion or because of the original input features. Nevertheless, if we accept these limitations for a moment, then one can at least say that the patterns seem more effective for XML than for JavaScript. A success rate of at least 80% seems fairly encouraging. However the success rate of only 50% for JavaScript indicates that our approach has problems to learn effective patterns there. On the flipside a potential success on every second pattern could also be not *too* bad, because it could still help increase the coverage if we are stuck at a plateau. However, our results of RQ2 do not suggest overall coverage improvements by our approach. Furthermore one has to bear in mind the limitations of our analyzed metric for this particular aspect.

To better understand the effectiveness of splicing, let us analyze the ratio between splicing target hits and splicing attempts and per targeting, i.e., $\frac{splicing\_target\_hits_i}{splicing\_attempts_i}$ for each targeting i. Again, we calculate the average of all these fractions per SUT. This gives us an estimate probability that a learned pattern will also result in a target hit after splicing. The data is presented in Table 17.

SUT	**Mean**	**Std-Dev**	**Std-Err**
ant	0.1802	0.16606	0.0086
closure	0.08314	0.20842	0.00685
maven	0.22514	0.06245	0.00224
rhino	0.16596	0.29294	0.01463

Table 17: Splicing hits to attempts ratio

You can see that the probability that a splicing attempt will result in a target hit is rather low. The results are again best for XML and in particular for Maven with a 22% target hit rate. However, taken together one can not conclude that splicing has a high chance to result in a target hit. It is also interesting that the JavaScript benchmarks have a relatively high standard deviation of at least 20%. This indicates a noticeable variability in the hit rate of patterns and thus possibly in their "quality"[57]. The relatively low standard deviation on Maven (6%) suggests that most patterns have a relatively equal chance at triggering the target. However this probability seems still rather low.

Even though the probability of target hits when splicing appears small, it still could be that the splicing makes a positive impact on the hits of a targeted branch. For this let us analyze the average quotient between the target hits at the end of a targeting and

[57]I.e., some patterns could be particularly ineffective while others are better.

the target hits at the start of a targeting, i.e., $\frac{target\_hits\_end_i}{target\_hits\_start_i}$ for each targeting i. This data is presented in Table 18.

SUT	Mean	Std-Dev	Std-Err
ant	24.09423	74.99507	3.8831
closure	143.13347	721.89357	23.74855
maven	11.68576	59.82011	2.1419
rhino	289.0845	1648.96666	82.44833

Table 18: Average quotient of target hits at the end of a targeting and the target Hits at the start of a targeting

You can see that the results are quite positive on average. Particularly, on the JavaScript benchmarks we can end up with target hits which are more than 280 and 140 times higher than the value before the targeting. At the same time, we have a relatively high standard deviation on all benchmarks (compared to the mean) which indicates that the hit increases can vary drastically from pattern to pattern. The high standard errors for Closure and Rhino also somewhat dampen the increase one might expect on average. However, even if we subtract the standard error from the means, we end up with relative increases of more than 100 and 200 times on Closure and Rhino respectively.

It is interesting that the average relative increases are far lower on the XML benchmarks compared to JavaScript. Our intuition is that there could be fewer branches with particularly difficult feature requirements. Thus, the target hits at the start of a targeting could already be relatively high which would then lead to a lower relative increase. However, this is only a hypothesis which we have not investigated further. It should also be noted that we do not focus on hits which happened after splicing but consider *all* registered target hits during a targeting. Thus we can not state that these increases are necessarily due to splicing.

For completion, we also present the average absolute increases. These are calculated by subtracting the target hits at the start of a targeting from those at the end (i.e., $splicing\_target\_hits\_end_i - splicing\_target\_hits\_start_i$ for each targeting i) and then by averaging these per SUT. This data is presented in Table 19.

You can see that the increases can seem quite quite sizable with around 23000 hits average hits on Maven. At the same time we have seen that the relative increase on Maven is only 11 times of the start value (see Table 18). Thus, the large number can be misleading when qualifying the observed increases. Because even though we have much smaller *absolute* increases on Rhino and Closure, the *relative* improvements are far better than for Maven. Thus one could say that these SUTs tend to benefit more from the targeting than Maven (and Ant). However, we have at seen for Closure that the improvements do not seem to be often (or strong) enough that one can observe

SUT	Mean	Std-Dev	Std-Err
ant	345.27415	393.59649	20.13815
closure	438.75754	1613.60018	52.99758
maven	23681.5557	14802.31657	530.00795
rhino	4405.52605	9176.19427	457.66697

Table 19: Average difference between the target hits at the end and the target Hits at the start of a targeting

significant differences in the number of rare branch hit counts. Again, one reason could be that we do not consider our adapted targeting heuristic in RQ1 which is however also considered here.

Let us finally analyze how many of the target hits during a targeting occurred with spliced inputs. For this we divide the splicing target hits by the overall target hits per targeting, i.e., $\frac{splicing\_target\_hits_i}{target\_hits\_i}$. This gives us an estimate of how many of the target hits occurred when (and thus could be due to) splicing. The averages of these fractions per SUT are presented in Table 20.

SUT	Mean	Std-Dev	Std-Err
ant	0.83722	0.30945	0.01698
closure	0.74481	0.38195	0.01659
maven	0.97699	0.11392	0.00414
rhino	0.8354	0.30603	0.01984

Table 20: Average quotient of splicing target hits to target hits per targeting

You can see that on average most of the target hits (per targeting) occurred with spliced inputs. This data suggests that it could be indeed our splicing technique which is responsible for most of the target hits. At the same time one has to be careful when interpreting this data. The reason is because our splicing probability is dependent on the ratio of shared branches between the inputs we mined on and the inputs we splice to. Thus we have a higher probability to splice to inputs which execute similar functionality to the inputs we mined on. However, such inputs with similar branches also (arguably) have a higher likelihood of triggering our rare branch due to *random mutations*. Thus we not be sure that it is really our splicing technique which is the main contributor because we have a bias to splice to inputs which might be at any rate more likely to trigger our target branch.

Nevertheless, if we ignore this limitation for a moment then the data seems relatively positive. However, the quite high standard deviations indicate a high variability. This means that it can apparently vary quite significantly from pattern to pattern how many of the target hits co-occur with splicing and which not. It is again noteworthy that the results are better for our XML subjects compared to JavaScript.

Threats to validity One threat to internal validity comes from odd observations on JQF's coverage instrumentation. We checked at the start of the project that the ID of a branch is identical, regardless whether we run a campaign with the ZestGuidance or one of our custom ones. This was performed by running a fuzzing campaign with the ZestGuidance and obtaining the input queue afterwards. We could then run this queue as seeds to our custom Guidances and see whether the branch IDs are identical for each input. This seemed to be the case. However, it seems that we have only checked it on Maven as an SUT. Because we have later found that this relationship must not necessarily hold. Even if we run the same inputs on the same Guidance. The issue seems to lie within JQF's new FastNonCollidingCoverage implementation, which is a relatively recent addition. If we use JQF's original Janala coverage implementation, then the branch IDs appear to stay identical, even between different Guidances (but provided that you use the same generators). Still it seems that the branch hit counts can vary, even with Janala. Possibly, we have not fully understood how we could correctly query the hit counts, but to the best of our knowledge, it seems fine. Luckily we are not interested in the branch hit counts, but only in the branch IDs. The reason is that we only count one hit per input and branch even if that input visited a branch multiple times. Due to these reasons we ran our experiments with the Janala coverage. Nevertheless, these are odd observations which should be definitely kept in mind when interpreting the results of our experiments.

Another threat to interval validity comes from the way we evaluate RQ1. To compare the number of hit counts between Zest and our approach we first ran several preparatory campaigns. These were meant to establish a "ground truth" which branches are typically rare per SUT. We could then track and analyze data particularly for these branches to compare our approach to Zest. To obtain the rare branches we performed multiple Zest campaigns per SUT to account for the randomness of fuzzing. Furthermore, we sampled random branches at different times to consider changing rarity. We then calculated the union to obtain the "typically" rare branches of an SUT. While it makes sense to calculate the union for the rare branches of a campaign, we find it a bit odd that we had to resort to the union to obtain the rare branches between the campaigns of one SUT. Intuitively the intersection should produce a more reliable set concerning which branches can "typically" be considered rare. However, when we intersected the rare branches over different campaigns, we ended up with 0 rare branches for each SUT. This was rather unexpected. It could be that rarity is quite variable over the course of a campaign, however we have sampled the hit counts for each campaign at exactly the same points in time. Possibly, our sampling granularity is too low with only 3 measurements per campaign (at 1, 6 and 12 hours). Nevertheless, these observations and the fact that we calculate the union over different campaigns can shed some doubt on the veracity of our "ground truth" set of rare branches.

Finally, we also wan to mention that the number of effective targetings can be quite lower than one might expect. Our implementation checks whether it should target a new branch each 10 minutes (once it switches to targeting mode). However, we have often found that only around 20 targetings have been performed in a campaign with a 12 hour duration. We have not performed a deeper analysis of this, however this could

be one of the reasons why there is only a relatively small overhead noticeable regarding the coverage metrics. Because the targeting of new branches does not necessarily occur very often. Nevertheless, even though the number of targetings can be small it seems that they can still have positive impacts on the hit counts of rare branches (RQ1).

In terms of external validity one can critique our limited set of SUTs and input formats. We have chosen the same benchmarks as in Zest's [PLS+19c] original evaluation (or at least 4 out of 5). This should provide a fair comparison between our approach and Zest. Nevertheless, an analysis of only four SUTs is quite limited in scope and could be extended in future work. Also, the selection of XML and JavaScript as input formats is rather small. It would be interesting to see how our approach performs on other tree based input formats.

5.3. Discussion

The results that we have presented paint a complex picture. One the one hand it seems that our approach can indeed be successful at increasing the number of target hits and unique traces (RQ1). At the same time these results can vary from SUT to SUT and from metric to metric. The best results have been observed for the number of *valid* rare branch hits. Here, we have seen significant increases for all subjects except for Closure. At the same time these improvements did *not* also lead to significant improvements of the valid (or overall) coverage (RQ2). On the contrary, we have seen a significant *decrease* in the valid coverage for Maven. This is particularly interesting as Maven otherwise is that benchmark where splicing seemed to work best (in relation to the number of splicing attempts per targeting). The data seems to suggest that our pattern mining and splicing approach *can* be successful at increasing the branch hits (and partially path traces) for targeted branches (RQ1). This is also indicated by the higher number of total paths we have observed for most SUTs when analyzing RQ2. However this improved exploration did not also lead to coverage increases. This was was one of our main hopes but it was not confirmed. Ou approach did also not lead to the discovery of more deduplicated bugs. Nevertheless, it could be interesting to see how the approach performs on other SUTs. Furthermore, one could possibly analyze additional metrics like the mean time to discovery of bugs. This could give further insight into the performance of our approach.

It is interesting that our pattern mining and splicing technique seemed to generally perform better for XML than for JavaScript. Again, our assumption is that patterns might be more easily learnable on XML due to its simpler tree structure. Another reason could be that certain JavaScript branches could only be triggered by *semantic* feature patterns. For instance, imagine that a certain rare branch is only triggered if you have a "while"-node which has another "while"-node in its subtree. However, it does not matter *where* and how *distant* the nodes are. Our approach could have problems with such patterns because it requires fixed sequential structures which are fully contiguous. This allows for no gaps or other degrees of freedom within a pattern. The consequence

is that we could fail to identify patterns which otherwise could be simply described with natural language. This is also indicated by our analysis of RQ3. Here we have seen that the quality of our learned patterns can apparently vary drastically. That is suggested by the high standard deviations for the target hit increases (Table 18, page 88) and the proportion of target hits which co-occur with splicing (Table 20, page 89). Therefore it seems that some patterns *can* be quite effective while others are less so.

Another interesting question is how much of the target hit count improvements (RQ1) can be really traced back to splicing. Because it could be that we sometimes only partially managed to learn a pattern. These would then not necessarily result in immediate target hits. However, these patterns could still produce inputs which are more likely to trigger the target branch if mutated. Thus, if Zest stores these inputs in its queue and mutates them later, these inputs could trigger a target hit as an indirect consequence of splicing. However, such effects are only speculative. Our analysis of RQ3 suggest that many target hits indeed seem to immediately co-occur with splicing. However, these observations only analyze the window of individual targetings. Thus, there *could* be positive effects of splicing which only indirectly occur. But we currently do not have data to either support or reject this hypothesis.

Taken together it seems that our approach can at least partially fulfill its intended goals. However, increased numbers of rare branch hits or unique traces have not brought along the coverage (or bug discovery) improvements we have hoped for.

6. Conclusion

All in all it seems that our approach of pattern mining and tree-based splicing has some potential to support a fuzzing campaign. Namely, it could possibly be used to better explore targeted areas in terms of branch hits and unique traces. Nevertheless, our results also suggest quite plainly that such improvements can not be expected to entail *overall* coverage (or bug discovery) improvements. On the contrary, we have seen that this can have *negative* impacts on general performance of a fuzzer. The reason could be that targeted areas simply do not guard new code to cover or that the patterns we splice do not reflect the necessary input features. This would then produce overhead which will not be beneficial for the fuzzing campaign. Possibly it would be a better use case for our approach if it would be employed for directed fuzzing [BPNR17, CXL+18]. However, one downside of our approach is that we *need* inputs which hit the targeted branch. Otherwise we could not learn a pattern for this branch. This is a stark contrast to the directed fuzzer AFLGo [BPNR17] which can perform its targeting in gradual steps. The reason is that AFLGo can focus on inputs which have a smaller distance to the target branches in terms of the call- and control flow graph of an SUT. Our approach does not have a similar notion of distance (in terms of input features). This is because we can *not* know beforehand which input features are necessary to reach a targeted branch.

Even though we have observed some positive results and a certain potential, we do not think that the additional overhead introduced by our approach is in general worthwhile for fuzzing campaigns. At least when considering our current approach and implementation. Especially when performed with generator-based fuzzing. The reason is that the splicing routines have to be implemented individually per generator which can be quite complicated and time consuming. It could be that the technique can be helpful in some ways but our current results do not point to improvements for fuzzing campaigns which attempt to explore SUTs as efficiently as possible, ideally without much additional work and preparation beforehand.

For future work one might nevertheless think about improving our approach. One possibility is to further develop the pattern learning. For instance, currently we only learn from positive examples. This refers to inputs which actually *hit* the target branch. However, it could also be interesting to learn similarities from inputs which do *not* hit our target and to remove these from our learned "positive" patterns. This could produce patterns which are more concise and which could thus have a higher likelihood to be spliced successfully and maintain input validity.

Additionally one could think about *improving* patterns one has mined based on experience. For example, one might try to remove certain parts of a pattern and see if it is still effective at triggering a target functionality (similar to Hierarchical Delta Debugging [MS06]). This might also lead to simpler patterns which could be more effective.

Another current downside of our approach is that we only learn and splice *one* subtree. However it could perfectly be that patterns require *several* subtrees. Thus,

one might investigate how one could best identify which subtrees are necessary and how their splicing could be coordinated (e.g., possibly they need be spliced a specific spots compared to one another).

Furthermore, one might consider to combine our approach with a call- and control flow graph analysis. We have seen that our approach was partially effective at increasing the hit counts of rarely visited areas. This suggests that our targeting of specific branches could be successful. However, there is no guarantee that rarely visited areas also guard new code to uncover. This might be one factor why we have failed to observe overall coverage improvements. It could thus be interesting to particularly focus on targets for which we *know* that there is more code to be explored. This could possibly be determined by analyzing how a certain branch is situated within the call- and control flow graphs of an SUT and whether there are unexplored areas below that branch. Focus on such targets might lead to a better tradeoff for our approach and could possibly lead to significant (overall) coverage improvements.

Moreover it could be interesting to further explore the viability of our pattern mining and splicing approach for debugging purposes. Our results seem to suggest that it can be effective at targeting rarely visited areas. Thus, we could possibly also learn the features of particular crashes. We have performed preliminary experiments on that which do not necessarily indicate that we are currently more efficient at reproducing crashes compared to an approach which simply mutates crashing inputs. Nevertheless, we have seen that there is still room for improvement of our approach. Thus, it could possibly have some potential. Especially because the mined patterns could give insights on which input features likely case a bug. This is not something one can (directly) obtain when mutating crashing inputs.

Furthermore, it could be interesting to see if we could learn better patterns with graph-based pattern mining. Our sequential pattern mining approach is not ideal because we have to reconstruct trees based on estimated guesses. Graph-based pattern mining could resolve such issues. However, from our current perspective it would be ideal to use a pattern miner for *directed* graphs. We had some difficulties to find corresponding (peer-reviewed) implementations which could be easily integrated into our technique. Nevertheless, if we are not wrong, then there are ways in which information on "directedness" can be recovered from undirected patterns. One only has to augment the original graphs to represent the directedness via additional nodes and edge labels. Thus it could be interesting to see whether graph-based pattern mining could bring about improvements for our technique.

Another interesting point for further research is developing a robust heuristic which dynamically identifies the inflection point (or plateau) in the performance of a fuzzer. We think that this could be beneficial because it is likely that there are other fuzzing approaches (like e.g., Driller [SGS+16]) which want to set in when the performance deteriorates. Our heuristic seems to work reasonably well, however it has been developed based on observations we have made for our test subjects in particular and only when fuzzing with Zest. Thus one could possibly investigate whether our heuristic also works for other SUTs or other fuzzers (e.g., AFL [Zal14]). Furthermore, one might more generally analyze which markers could be analyzed to identify performance deterioration

and how existing research might be used to build a heuristic. Possibly there already exists a reliable heuristic which we have not been able to find (in addition to possibly the one of Driller [SGS+16]).

Moreover we would also find it interesting to develop approaches which attempt to identify (and ideally re-generate) input features on the parameter stream level of an inputs. This could be a more general way to identify and re-generate features with generator-based fuzzing.

Finally, it would be interesting to see how our approach works for grammar-based fuzzing. We think that our technique might indeed work better here because you likely do not have to implement the splicing and pattern mining per generator. Instead you ideally only have to do it once. Namely, for the generic derivation tree of a word. This could lead to a better tradeoff between the implementation effort and potential benefits. Possibly one could generalize our generator-based approach so much that our tree-based splicing has a similar degree of abstraction. However, we found it difficult to explore within the time for this project. Grammar-based fuzzing might thus be an interesting option to further explore our ideas as you could immediately employ it for different input formats, provided that you have a corresponding grammar.

References

[AFH+19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *26th Annual Network & Distributed System Security Symposium (NDSS)*, Februar 2019. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-3_Aschermann_paper.pdf.

[Agg15] Charu C. Aggarwal. *Data Mining: The Textbook*. Springer International Publishing, Cham, 2015. URL: https://link.springer.com/book/10.1007/978-3-319-14142-8, doi:10.1007/978-3-319-14142-8.

[Ant] Apache Ant. https://ant.apache.org/. Accessed: 03.10.2022.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018. doi:10.1145/3182657.

[BCE] Apache Byte Code Engineering Library. https://commons.apache.org/proper/commons-bcel/. Accessed: 03.10.2022.

[BCR21] M. Boehme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(03):79–86, 2021.

[BNK16] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. instrument. In *A Dictionary of Computer Science*. Oxford University Press, 2016. URL: https://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975-e-2611, doi:10.1093/acref/9780199688975.013.2611.

[BPNR17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134020.

[BPR19] M. Bohme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(05):489–506, may 2019. doi:10.1109/TSE.2017.2785841.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351266.

[Clo] Google Closure. https://developers.google.com/closure/compiler. Accessed: 03.10.2022.

[CLO18] Sooyoung Cha, Seonho Lee, and Hakjoo Oh. Template-guided concolic testing via online learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 408–418, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3238147.3238227.

[CT12] Keith D. Cooper and Linda Torczon. Chapter 5 - intermediate representations. In Keith D. Cooper and Linda Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 221–268. Morgan Kaufmann, Boston, second edition edition, 2012. URL: https://www.sciencedirect.com/science/article/pii/B9780120884780000050, doi:https://doi.org/10.1016/B978-0-12-088478-0.00005-0.

[CXL+18] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243849.

[DRRG14] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, page 37–48, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2557547.2557550.

[FR19] Gordon Fraser and José Miguel Rojas. *Software Testing*, pages 123–192. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-00262-6_4.

[FV22] Philippe Fournier-Viger. Mining frequent sequential patterns using the cm-spam algorithm (spmf documentation), 2022. Accessed: 03.09.2022. URL: https://www.philippe-fournier-viger.com/spmf/CM-SPAM.php.

[FVGCT14] Philippe Fournier-Viger, Antonio Gomariz, Manuel Campos, and Rincy Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao, editors, *Advances in Knowledge Discovery and Data Mining*, pages 40–52, Cham, 2014. Springer International Publishing.

[FVHC+20] Philippe Fournier-Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Chun-Wei Lin, and Unil Yun. A survey of pattern mining in dynamic graphs. *WIREs Data Mining and Knowledge Discovery*, 10(6):e1372,

2020. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1372, arXiv:https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1372, doi:https://doi.org/10.1002/widm.1372.

[FVLG+16] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh Lam. The spmf open-source data mining library version 2. In Bettina Berendt, Björn Bringmann, Élisa Fromont, Gemma Garriga, Pauli Miettinen, Nikolaj Tatti, and Volker Tresp, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 36–40, Cham, 2016. Springer International Publishing.

[FVLK+17] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1), 2017. URL: https://www.philippe-fournier-viger.com/dspr-paper5.pdf.

[God20] Patrice Godefroid. Fuzzing: Hack, art, and science. *Commun. ACM*, 63(2):70–76, jan 2020. doi:10.1145/3363824.

[Han12] Jiawei Han. *Data mining : concepts and techniques / Jiawei Han ; Micheline Kamber ; Jian Pei*. The Morgan Kaufmann series in data management systems. Amsterdam [u.a.], 3. ed. edition, 2012.

[HGM+21] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 230–243, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460319.3464795.

[HKZ22] Nikolas Havrikov, Alexander Kampmann, and Andreas Zeller. From input coverage to code coverage: Systematically covering input structure with k-paths. *ACM Transactions on Software Engineering and Methodology*, February 2022. URL: https://publications.cispa.saarland/3572/.

[HMB17] Nigar Hashimzade, Gareth Myles, and John Black. Markov chain. In *A Dictionary of Economics*. Oxford University Press, 2017. URL: https://www.oxfordreference.com/view/10.1093/acref/9780198759430.001.0001/acref-9780198759430-e-3765, doi:10.1093/acref/9780198759430.013.3765.

[Hol] Paul Holser. junit-quickcheck: Property-based testing, junit-style. Accessed: 28.08.2022. URL: https://pholser.github.io/junit-quickcheck/site/1.0/.

[JQF] Jqf + zest: Semantic fuzzing for java. https://github.com/rohanpadhye/JQF. Accessed: 03.10.2022.

[Kan19] Mehmed Kantardzic. *Data Mining: Concepts, Models, Methods, and Algorithms*. John Wiley & Sons, Ltd, 3 edition, 2019. doi:https://doi.org/10.1002/9781119516057.

[KHSZ20] Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. In *ESEC/FSE 2020*, June 2020. URL: https://publications.cispa.saarland/3107/.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243804.

[Lem21] Caroline Lemieux. *Expanding the Reach of Fuzz Testing*. PhD thesis, EECS Department, University of California, Berkeley, May 2021. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-43.html.

[LKBS21] Steffen Lüdtke, Roman Kraus, Ramon Barakat, and Martin A. Schneider. Attack-based automation of security testing for iot applications with genetic algorithms and fuzzing. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 92–100, 2021. doi:10.1109/QRS-C55045.2021.00023.

[LS17] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 46–56, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3092703.3092711.

[LS18] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3238147.3238176.

[LSL21] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu.

[LZZ18] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, 12 2018. doi:10.1186/s42400-018-0002-y.

[Mat] Matplotlib. https://matplotlib.org/. Accessed: 04.10.2022.

[Mav] Apache Maven. https://maven.apache.org/. Accessed: 03.10.2022.

[MdCH18] João Mendes Moreira, André C. P. L. F. de Carvalho, and Tomáš Horváth. Frequent pattern mining. In *A General Introduction to Data Analytics*, chapter 6, pages 125–150. John Wiley & Sons, Ltd, 2018. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119296294.ch6, doi:https://doi.org/10.1002/9781119296294.ch6.

[MHH+21] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[MKC20] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1024–1036, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380421.

[MS06] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Software Engineering, International Conference on*, pages 142–151, Los Alamitos, CA, USA, may 2006. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1145/1134285.1134307, doi:10.1145/1134285.1134307.

[NG22] Hoang Lam Nguyen and Lars Grunske. Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 249–261, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510182.

[NNTH+21] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[Num] Numpy. https://scipy.org/. Accessed: 04.10.2022.

[Pay19] Mathias Payer. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security & Privacy*, 17(01):78–82, 2019.

[PLS+19a] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Validity fuzzing and parametric generators for effective random testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 266–267, 2019.

[PLS19b] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3339002.

[PLS+19c] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3330576.

[Rhi] Apache Rhino. https://github.com/mozilla/rhino. Accessed: 03.10.2022.

[RLPS20] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1410–1421, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380399.

[Sci] Scipy. https://scipy.org/. Accessed: 04.10.2022.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network & Distributed System Security Symposium (NDSS)*, Februar 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[SN07] P. K. Srimani and S. F. B. Nasir. *Context-Free Grammars and Context-Free Languages*, page 304–376. Foundation Books, 2007. doi:10.1017/UPO9788175968363.010.

[Zal14] Michal Zalewski. American fuzzy lop, 2014. Accessed: 27.08.2022. URL: https://lcamtuf.coredump.cx/afl/.

A. Appendix

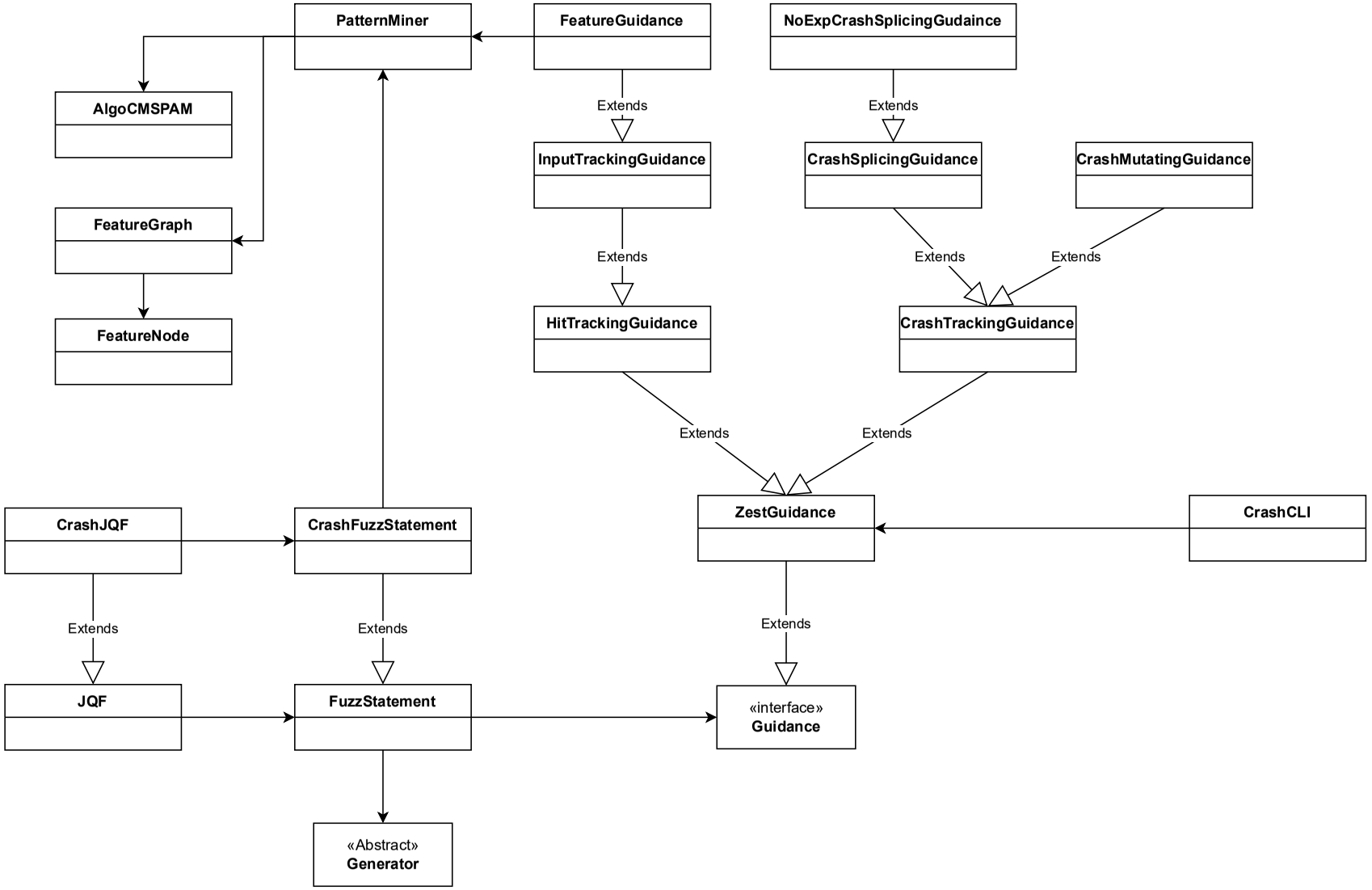


Figure 28: Class diagram of `jqf-crash`'s fuzzing framework

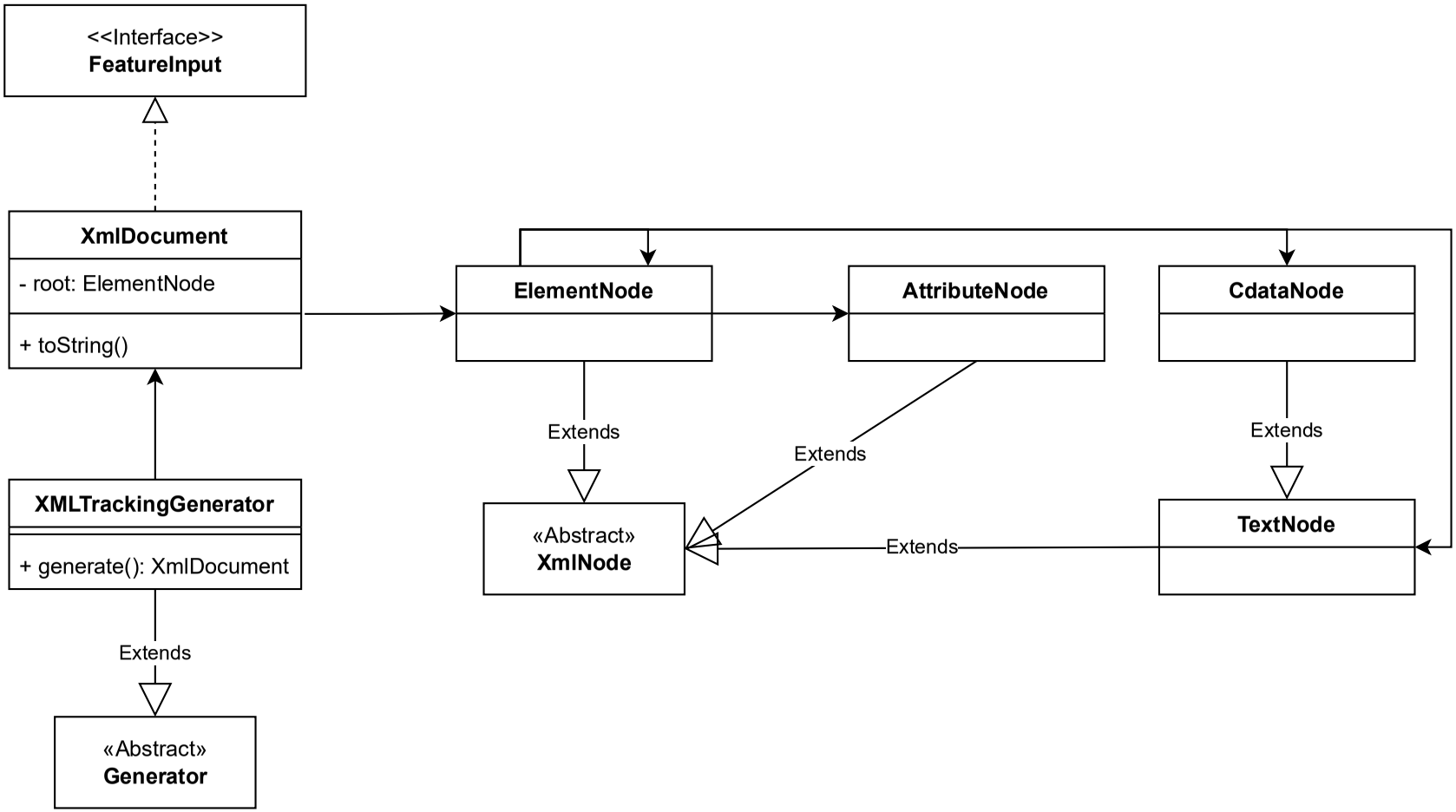


Figure 29: Class diagram of our XML generator and input

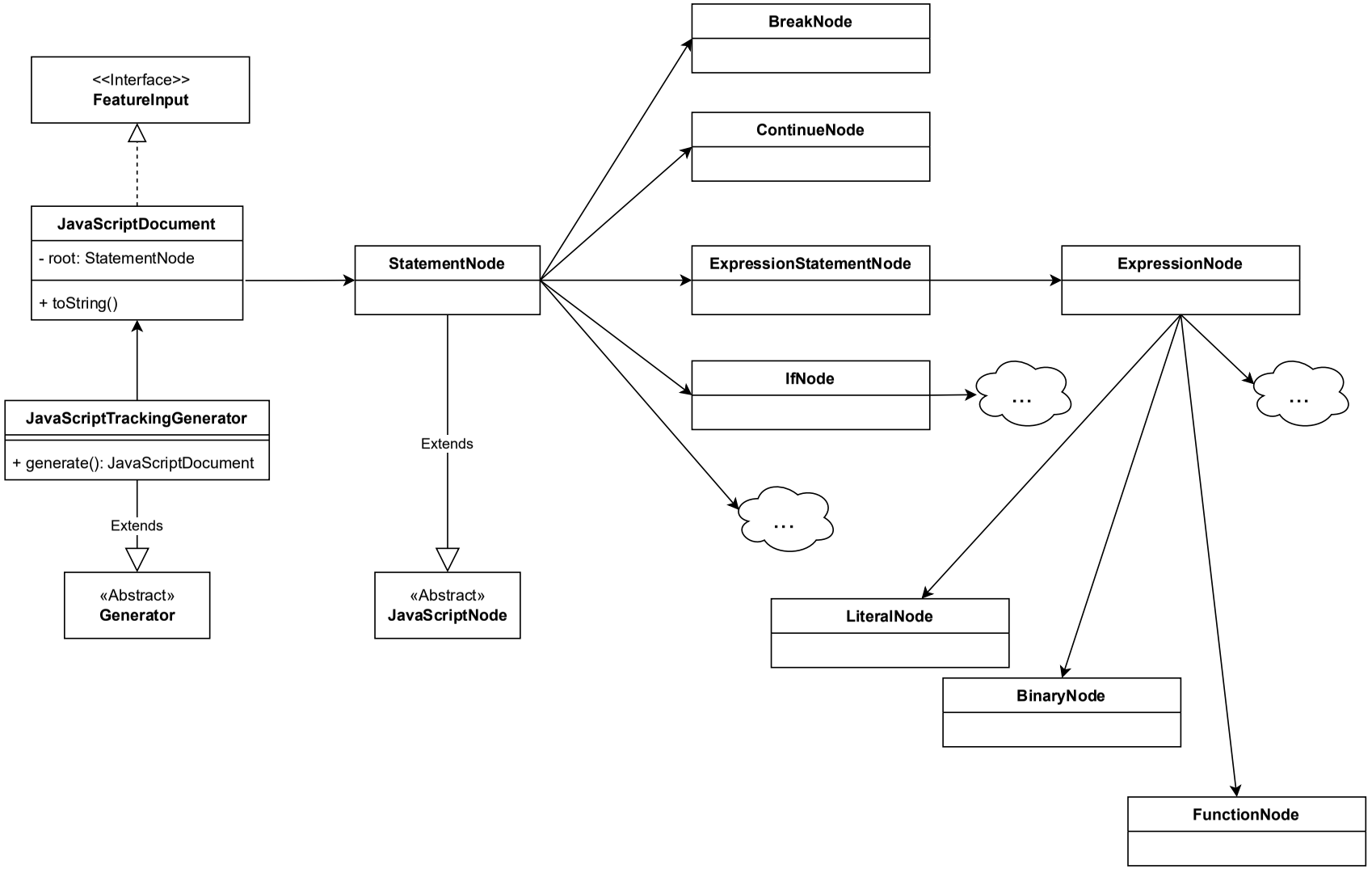


Figure 30: Class diagram of our JavaScript generator and input

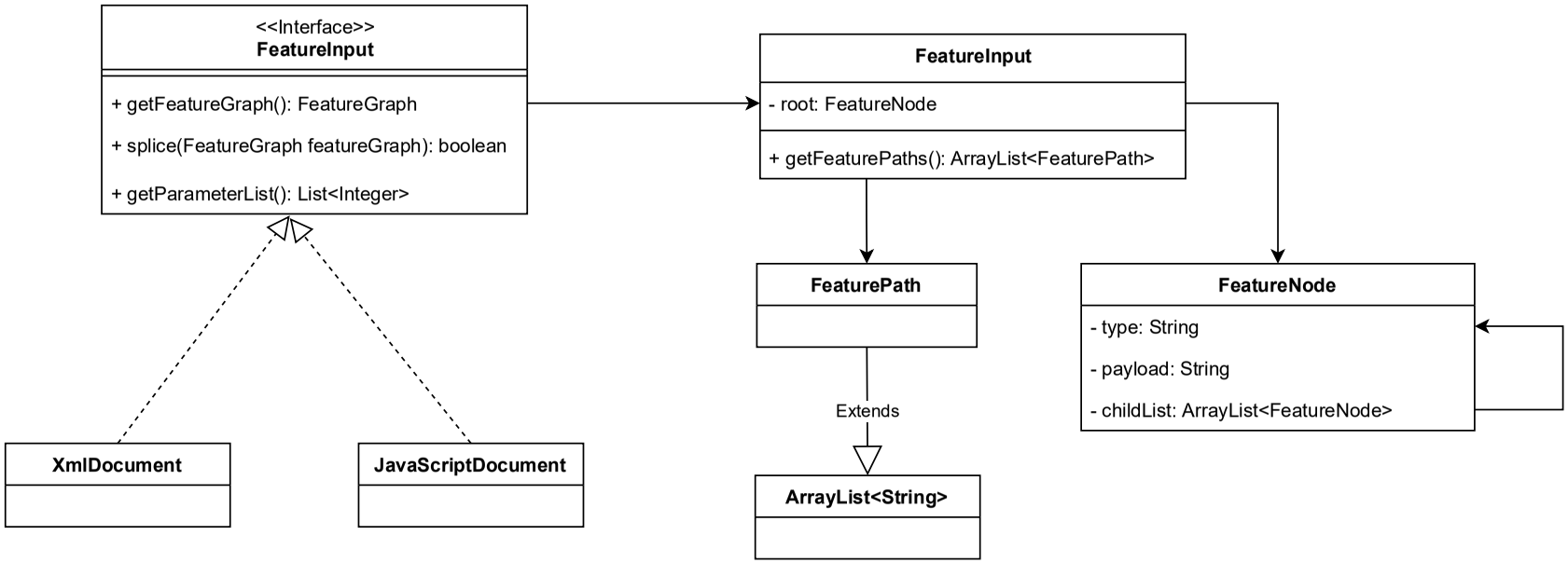


Figure 31: Class diagram of our classes for abstract tree-feature representation

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 4. Oktober 2022 .....