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

Grammarbased Fuzzing of Workflow Engines (GroWE)

Masterarbeit

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

eingereicht von: Alena Schemmert
geboren am: 29.08.1995
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Ulf Leser

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

Abstract. This thesis explores the domain of grammarbased fuzzing within a coverage-guided and a blackbox testing framework. The primary focus of this study centers on the practical application of applying grammarbased fuzzing techniques in the context of nextflow, a well-known Groovy-based data pipeline software. The main focus is on fuzz testing the whole execution of a data pipeline and not only the parsing or compilation stage.

The research effort involves the implementation of two different grammarbased generator testing strategies and one non grammar-based mutation strategy to test the execution of data pipelines within nextflow. Tested in a fuzzing campaign we could invetigate the coverage und bug finding capabilities of both generators and look at the ddifference of coverage-based and black-box fuzzing. The coverage results let us explore the differences between the parsing and execution of data pipeline scripts within nextflow.

We found that malformed scripts will pass the compilation stage and will throw exceptions during runtime of nextflows data pipline execution.

Contents

1 Introduction **1**

2 Background **3**
2.1 Fuzzing . . . 3
2.1.1 Grammarbased Fuzzing . . . 4
2.1.2 Coverage-Guided Fuzzing in Java with JQF . . . 7
2.1.3 American Fuzzy Lop . . . 10
2.2 Data Pipelines and Workflow Engines . . . 10
2.2.1 Nextflow . . . 11
2.3 Literature Review . . . 13

3 Implementation **17**
3.1 Used Libraries . . . 17
3.1.1 JQF . . . 17
3.1.2 GramTest . . . 17
3.2 Generators . . . 18
3.2.1 Semantic Generator . . . 18
3.2.2 Syntactic Generator . . . 21
3.2.3 Wrapping Command Generator . . . 22
3.2.4 AFL Fuzzing in Java with JQF . . . 23

4 Experiments and Evaluation **24**
4.1 Experimental Set Up . . . 24
4.1.1 JQF Configuration . . . 24
4.1.2 Setup for Generator Comparison . . . 25
4.1.3 Reproducing Runs . . . 28
4.2 Results . . . 29
4.2.1 Analyzing Generated Inputs . . . 29
4.2.2 Analyzing Coverage . . . 33
4.2.3 Analyzing Crashes . . . 37
4.2.4 Comparing Coverage-Guided and Random Tests . . . 40
4.2.5 Findings . . . 42
4.3 Discussion and Threats to Validity . . . 43

5 Conclusion **46**
5.1 Future Work . . . 47

Bibliography **i**

Online References **iv**

Appendix **vi**

1 Introduction

Testing is an important part of writing software, as it ensures that the algorithms work according to the specification and can improve the reliability. For software testing many different approaches exists as which one of is fuzzing. Fuzzing, in general, describes the testing of software with randomly generated input data.

Unlike traditional software tests that are written for internal functions with fixed defined parameters, fuzzing focuses on the external interfaces of software and tests them with a wide variety of random data. Throughout many such tests, crashes and exceptions are logged, which can provide insights into potential faulty implementations. This can be highly effective in discovering errors within the software, but it is also limited by randomness, especially when applications expecting highly structured data need to be tested [1, 2]. Such applications include compilers and interpreters, but also internet browsers or software libraries that expect structured files. For testing such applications, there are already a number of so-called grammarbased fuzzers that can generate either code fragments [3] or simple file formats [4]. Furthermore, there are projects like CSmith, which specifically test various C compilers [5], or Superion, which can manipulate existing input files using grammars [6].

However, all these projects only consider the generation and processing of a single file in their experiments. In many applications, different files can be loaded through references in data that has already been read. This includes compilers or software linkers used in the compilation of software source code.

One specific type of software that also compiles multiple referenced input files in a single execution are workflow engines. Workflow engines take task descriptions in a formal description, such as a Domain Specific Language. The description of which tasks needs to be processed is parsed, and then executed in a corresponding workflow. Data pipeline tools are often used in bioinformatics for example for gene sequencing which is a manifold process and uses a variety of different existing command line tools. As most software, data pipeline tools and compilers for the workflow definitions may contain bugs.

To test such software is a challenging task, as the correct result of a data pipeline depends not only on the workflow engine itself but also on the used tools within. In general testing bioinformatics software alone is a complicated task, as they deal with large input and output data and verifying the correctness of those is very difficult in practice [7] . And workflow engines process many different of those tools at the same time, which makes them a quite complex system

One workflow engine that gained more interest and users over the last years is *nextflow*. It can process pipelines written in the most common scripting languages. Workflows and pipelines can be defined with a DSL (see Section 2.2) which allows users also to deploy those on clouds and clusters [30].

Fuzzing techniques are often apply to smaller libraries that are generally used for very distinct purposes like image type conversion or for parsing specific data types. If fuzzers test a bigger software a common approach in greybox fuzzing is to test the input handling parts of them (e. g. parsing or compiling) [8, 9, 10]. But the main purpose of workflow engines is to process different inputs from the user and execute different other applications with the input data.

This thesis will investigate if we can use grammarbased fuzzing techniques as a sensible option to test workflow engines during pipeline execution and what a possible test set-up can look like. Also, we will try to answer what the bug-finding capabilities of grammarbased fuzzing in workflow engines is on the example of *nextflow*.

The thesis is structured as followed: The Section 2 will provide background information on the topic of this thesis Section 3 focuses on the implementation and used libraries to explore the possibilities of grammarbased fuzzing on workflow engines. Section 4 describes and evaluates the experiments performed and Section 5 gives a brief conclusion and lays out possible future work.

2 Background

2.1 Fuzzing

Fuzzing is testing software by feeding random data to the software under test (SUT). This technique was first introduced by Miller et al. in 1990 as *An Empirical Study of the Reliability of UNIX Utilities* [11]. The researchers tested UNIX tools with a program called “fuzz”, which generates a stream of random characters to be consumed by a SUT. This approach is nowadays called blackbox fuzzing [12]. Blackbox fuzzing can be described as a random input fed to the program without knowledge of its internal behaviour or implementation. The SUT is then tested with many variants of the first input or completely new ones in the hope of triggering vulnerabilities or crashes. While doing so, the fuzzing software monitors the SUT and looks for those interesting tests, reports them and uses this generated data as base for the next input data and test.

This simple approach has found thousands of security vulnerabilities, mainly in code that handles binary input data. The effectiveness of blackbox fuzzing relies heavily on a diverse corpus of well-formed input seeds, which will traverse a variety of the program by themselves. To those seeds the fuzzing software will add noise by for example changing byte per byte over each iteration. This idea has been proven effective as it has found several bugs [31]. On the other hand this technique is limited as the generation of new interesting executable inputs has very low probability [13] as it does not take into consideration the expected structure or syntax of the input data.

Complementary to blackbox fuzzing is whitebox fuzzing, where the test inputs are based on knowledge of the SUT and its architecture. It extends the systematic dynamic test generation from the scope of unit testing to the whole program [12] Whitebox fuzzing is able to learn about the SUT while testing and therefore need not revisit already executed paths like blackbox fuzzing [32].

With whitebox fuzzing we will start with a well-formed input seed, but use dynamic symbolic execution of the SUT to gather constraints on inputs from conditional branches visited along the execution. For the next execution the collected constraints are systematically negated and solved with a constraint solver. Those solutions are used to generate new inputs that hopefully will trigger different program execution paths [12].

An apparent solution to the fast but naive blackbox fuzzing and the complex computation heavy whitebox fuzzing is the combination of those approaches called greybox fuzzing [14]. With greybox fuzzing the test generation is as fast as with blackbox fuzzing, but to address the problem of repetitive execution of the same program path, it will use additional feedback from the program which parts are already executed [32].

There are many different approaches on how to address the challenge of fuzz testing software with random data. And as you might have guessed, there are many different names for similar ideas. To anticipate this, Table 1 can be used as a reference for the following chapters.

Table 1: Ideas of Different Fuzzing Techniques

Fuzzing Techniques		
Name	Main Idea	Synonyms
whitebox	symbolic execution and knowing the software, search in program, constraint solver, mutation or generation based	$\sim$ automated dynamic test generation
greybox	uses feedback from program for next test, often used with coverage, mutation or generation based	guided, feedback, coverage
blackbox	tests not necessarily matching the input format, only crashes are logged	blind, random testing

2.1.1 Grammarbased Fuzzing

There are different approaches on how to include feedback of the target software to perform better fuzz testing. Two research fields that has gained more interest over the last years are grammarbased and greybox fuzzing. As explained in the previous Section, greybox fuzzing combines the best of both previous existing approaches: rapid execution (blackbox) while using feedback from the fuzzing targets about which parts where already executed (whitebox).

One concept in greybox fuzzing is to use a grammar definition to produce valid test inputs for a target software. This can be done by generating test inputs that match a specific grammar (Section 2.1.1) or by mutating an existing test input whithout violating the syntax. The main idea behind both strategies is to use structured inputs for testing, because software written to handle files of a given type will likely fail early in the execution process when given random bytes

Although grammarbased fuzzing can be applied as blackbox fuzzing, researchers use it in combination with some sort of feedback driven approach [14, 15] One technique that pairs well with grammarbased fuzzing is coverage-guided fuzzing [2, 6].

To understand how coverage-guided approaches in fuzzing work, we need to know what coverage and instrumentation are.

Coverage. Code coverage is a percentage measure in software engineering, to which degree the source code of a program is executed while running certain tests. If we reach a high coverage during test execution, we can suggest that we have a lesser probability of undetected software bugs existing in the program [33].

The first published reference for software testing with all possible input combination was published in 1963 by Miller and Maloney in Communications of the ACM (Association for Computing Machinery). They described programs and their flowcharts as logical trees and suggested, that if the tree was used for systematically creating combinations input data that the SUT must handle and "testing these cases and subsequently assisting the programmer in locating mistakes in the program" [16].

In the best case scenario this approach would reach a nearly perfect condition coverage. Condition coverage describes that every boolean expression has been executed with the evaluation of true and false. This is also called predicate coverage and is one of the smallest unit of coverage criteria (besides line coverage) that you can measure[17].

The next lesser detailed coverage metric is called branch coverage where the measure is whether each conditional statement of was executed with all possible outcomes (for example both true and false branches of an if statement). This is very similar to the conditional coverage but consider that not all boolean expressions are used in a statement or sometimes such statements have multiple parameters for true and false branch. Statement Coverage describes only that all statements must be executed, regardless which branch was chosen. And at last you can measure code coverage by checking if each function in a program has been called [18, 34].

Instrumentation. To gather the coverage information during test execution the program must be instrumented so that the test suite can observe executed code.

Code instrumentation happens by adding code to the existing software that will report the current executed function, statement or branch. Code for instrumentation can be added during development manually with, for example, logs or with tools that automatically add to the sourcecode.

Another option is to add instrumentation code at compile time or if there is no access to the source code it can be added to the compiled executable via binary translation.

During runtime the instrumentation can be realised by running the program fully supervised and adding the instrumentation directly before. For example Java programs can be instrumented with a so-called Java Agent at runtime while running on the Java Virtual Machine (JVM) [35](see Section 2.1.2). A Java-Agent can be used to start and observe a given application and also run code before the main function.

Grammars. A grammar of a language is used to describe how symbols from an alphabet can be used to form to valid words. Further it defines how these words can be combined into valid sentences[36]. Just because a word or phrase is syntactically correct does not necessarily mean that it is semantically correct, or even has a meaningful interpretation.

For example, in English, grammar rules dictate how words should be ordered in a sentence and how sentences should be combined into paragraphs to form a meaningful document. An error in grammar might result in a sentence that is technically syntactically correct but still does not make sense in the context of the entire document.

In software engineering syntax refers to the set of rules that describe how a program must be structured in order for it to be considered valid and understandable by the programming languages compiler or interpreter [37].

Semantics, on the other hand, deals with the meaning and logic of the program. It determines whether the code performs the intended operations correctly. Even if the syntax is perfect, the code may not work as expected if there are semantic errors.

Notation Form - BNF. To be used in software development grammars and their rulesets themselves need to be formalized and machine readable. One useful and widely used notation form is the Backus-Naur-Form (BNF).

BNF is a meta syntax notation that can be used for context-free grammars. It is often applied in computer science for defining programming languages, document formats or communication protocols.

The notation form firstly was introduced in 1960 by John W. Backus to describe the syntax of ALGOL-60, an international scientific programming language. It later was also named after Peter Nauer, who also contributed to the development of said programming language and to the definition of the notation format [38, 39].

This is an example of a really simple grammar written in BNF that would produce the following outputs: "Hello World!" or "Hi World!"

Listing 2-1: Simple grammar

```
1 <hello-world> ::= <greeting> <world>
2 <greeting> ::= "Hello" | "Hi"
3 <world> ::= " World!"
```

2.1.2 Coverage-Guided Fuzzing in Java with JQF

One state-of-the-art fuzzing tool for Java libraries is JQF . It "uses the abstraction of property-based testing, which makes it nice to write fuzz drivers as parametric JUnit test methods." [19]

By building it on top of junit-quickcheck[1] the developers around Rohan Padhye enabled users of this framework to apply fuzz testing with coverage-guided fuzzing algorithms such as Zest (2.1.2) by writing junit-quickcheck style unit tests [19].

Although intentionally written for Java Code, fuzz testing with JQF can also be applied on different JVM-based programming languages such as Groovy (see Section JVM 2.1.2).

Initially JQF was released with one algorithm to track and to calculate branch coverage during testing. In the first implementation the Algorithm uses JaCoCo[2] for the instrumentation it collects every visited branch and uses the hash from this branch to increment a counter at the position of the hashed ID. The coverage `ArrayList` has a fixed size of $2^{1}6$ possible entries. When fuzzing a target it can be possible that different branches will generate the same hashed index and therefore would be handled as the same branch covered. So this branch coverage logged while fuzzing only represented the filling degree of the coverage `ArrayList` . This is a simplistic fast approach, that gives a good overview what the SUT and all of it used libraries have executed. However, those numbers can also be misleading because they do not represent the covered parts of the software at all, but a heuristic evaluation.

In May 2023 JQF released a new version including a second algorithm for collecting coverage information: fast non colliding coverage instrumentation. This new branch coverage was implemented by the researchers of CONFETTI[9, 40] and it improved in their measuring the execution speed by 7-10 times. Instead of using an `ArrayList` and filling only specific indices of this, the faster version stores coverage counts in a map. The keys to be stored in the List are generated by incrementing a plain integer for every newly visited branch in the instrumentation code and then played this back via reflections with JaCoCo to the coverage map[41]. So every key in the map is one unique branch, meaning there is no collision.

But still it only presents the absolut number of visited paths in the fuzzing target and not how much percentage of it was covered.

To compensate for blank numbers of branches visited, the developers themselves compare the generated coverage of different fuzzing algorithms [8, 9]. Often, the most common or naive approach is used as comparison and is called baseline. Comparing a fuzzing approach against a baseline is a common method for evaluation [14, 10].

[1] `https://github.com/pholser/junit-quickcheck`
[2] `https://github.com/jacoco/jacoco`

Zest-Algorithm. JQF uses a specific algorithm called Zest to maximize the coverage during one fuzzing run. The idea is to combine generator based testing with the "power of fuzzing" [8]

The working principle behind coverage-guided fuzzing (CGF) is to use known inputs and mutating them randomly with small operations like bit flips or byte level splicing to produce new inputs.

If such mutated input has generated new coverage it is saved for further iterations in this run. By random byte manipulation those inputs often generate invalid inputs. Many problems found by CGF tools therefore are located in the syntax analyzing part of the software. To overcome this and find deeper laying bugs CGF tools often need long-running (24h) fuzz testing [20], what can be really interfering for continuous integration.

Zest, however, focuses on the semantic analysis stage of the program. To achieve this QuickCheck-like random input generators are converted to deterministic parametric generators. Parameters in this case are a sequence of untyped bits that are used to generate a syntactically correct input. The key insight in this approach is that bit-level mutations on this parameters will lead to structural mutations of syntactically valid inputs.

In their paper Padhye et al. [8] showcase a simple QuickCheck-like XML document generator as shown in Listing 2-2.

Listing 2-2: Simple XML Document Generator from the Zest-Paper[8]

```
1   class XMLGenerator implements Generator<XMLDocument> {
2   @Override // For Generator < XMLDocument >
3   public XMLDocument generate(Random random) {
4       XMLElement root = genElement(random, 1);
5       return new XMLDocument(root);
6   }
7
8   private XMLElement genElement(Random random, int depth) {
9       // Generate element with random name
10      String name = genString(random);
11      XMLElement node = new XMLElement(name);
12      if (depth < MAX_DEPTH) { // Ensures termination
13          // Randomly generate child nodes
14          int n = random.nextInt(MAX_CHILDREN);
15          for (int i = 0; i < n; i++) {
16              node.appendChild(genElement(random, depth + 1));
17          }
18      }
19      // Maybe insert text inside element
20      if (random.nextBool()) {
21          node.addText(genString(random));
22      }
23      return node;
24  }
25
26  private String genString(Random random) {
```

```
27          // Randomly choose a length and characters
28          int len = random.nextInt(1, MAX_STRLEN);
29          String str = "";
30          for (int i = 0; i < len; i++) {
31              str += random.nextChar();
32          }
33          return str;
34      }
35  }
```

By overwriting `java.Random` with `junit.quickcheck.random.SourceOfRandomness` and saving each single bit that generator has gotten via `random.nextInt()` or `random.nextChar()` for generating a test input the algorithm is able to generate similar new valid inputs by just flipping single bits in the source of randomness.

For example: the function `genString()` in the Generator (2-2) got `nextInt() = 3` and `nextChar()` produced `f, o, o` the bits saved for this would be `s1 = 0000 0011 0110 0110 0110 1111 0110 1111` . The first eight bits are for `3`, the next eight for `f`, and last two sequences of eight bits for too `o`

When the Zest-Algorithm mutates this sequence to

`s2 = 0000 0011 0110 0010 0110 1111 0110 1111` it would produce 3 and b o o. Considering the selected three chars are used in an XML element definition both the first definition `<foo> ... </foo>` and the mutated version `<boo> ... </boo>` are syntactically and semantically correct.

The Zest-Algorithm will use valid inputs for mutations in the next iterations trying to maximize the covered branches of the current target software. But success or failure depends on the quality of the underlying generator that will build the input structure. A generator that focuses on different aspects of the input will cover more diverse features of the target.

Java Virtual Machine. The Java Virtual Machine (JVM) will run programs that are compiled to Java bytecode on multiple different platform. A JVM language can every language be called that produces a valid class file that the JVM can host. A class file is such that contains the instructions (Java byte code) for the JVM .

There are multiple existing languages that were ported to the JVM like Ruby and Python, but also completely new languages where created. The most popular newly created languages to compile to Java byte code are Kotlin, Scala and Groovy. One great feature of JVM languages is their incompatibility with each other, meaning within Java programs Scala or Groovy libraries can be used and vice versa [42].

2.1.3 American Fuzzy Lop

Another popular greybox fuzzing tool is AFL (American Fuzzy Lop). It uses a mutation-based approach to test a program, meaning it take seed input files and modifies them or combines different seeds. Because it uses coverage-feedback it is neither blackbox nor whitebox fuzzing. Blackbox fuzzing would use no program feedback for the next tests and whitebox fuzzing mostly uses complex and expensive analysis or constraint solving. AFL uses program instrumentation to determine if a generated input creates more coverage. If so, it is added to the seed corpus for upcoming tests to use. The developers of AFL recommend recompiling the source code with an own companion tool that functions just like gcc or clang. So instead of using gcc for compiling C programs you should use afl-gcc and afl-g++ for C++ programs [43]. For apps where the sourcecode is not available there is an experimental support called qemu, which adds on-the-fly instrumentation and is approximately 2,5x slower.

While testing (e.g. execution) the instrumented program AFL is able to log the exercised branch and can count the executed times for overall and each test [44] Mutated output that caused crashes or hangs are saved to the respective folder for further investigation.

AFL implements different mutations strategies, which where successful in finding bugs. Since the publication in AFL has found countless bugs and vulnerabilities in libraries, command-line tools and even operating systems. The tests made with AFL also contributed in making non-security improvements to plenty of core tools [31].

2.2 Data Pipelines and Workflow Engines

Nowadays for almost every problem in data analysis there is an already existing piece of software to solve it. In bioinformatics data analysis pipelines are one of the main tools for researchers as they help to integrate new technologies and computational tasks [21]. Often it can be quite tedious to write scripts that combine those tools in a single data pipeline to execute them. This is where data pipeline tools come into use that will execute those pipeline as so-called workflow.

To process a given data pipeline those tools use a formal scripting language or define an own language for this purpose. A standard that can be used as a formal definition of data pipeline is the Common Workflow Language (see 2.2). If a workflow is defined in a custom language it is often called a Domain Specific Language (see 2.2). *nextflow* is one popular workflow engine that uses its own DSL for running pipelines.

Domain Specific Languages. DSLs are build on top of existing programming languages with the aim to simplify the original syntax [22].

It can be viewed as a way to increase abstraction in software development even further. DSLs can be used to model specific concepts from a domain directly in to a language and design an abstraction which purpose solely supports solving tasks out of this domain [23].

One good example of now widely used abstraction of a language that took away a huge amount of detail and programming knowledge from the users is HTML . For designing a web page the users do not need to know about the complex layout algorithms that internet browsers will us to display the page [23]. Wąsowski and Berger stated the following definition:

"A domain-specific language (DSL) is a computer programming or modeling language of limited expressiveness focused on a particular domain or its aspect." [23]

So a DSL can be used for various software tools as supporting language to facilitate task definitions highly adapted for the users needs.

Common Workflow Language. Another way of describing data pipelines is the open standard called Common Workflow Language (CWL). It can be used to describe how to run and connect command line tools [45].

The language specification is designed by the bioinformatics community and focuses on reproducibility on any hardware. CWL tries to enhance principle and standards of coding pipelines and uses YAML as notation form. As parameters needs very explicit definitions the language is considered "verbose"[24]. But CWL is not closed to bioinformatics and a variation of different institutions from fields like Radio Astronomy, Geospatial information or for cross domain data analysis use CWL to describe their workflows [45].

Known data analysis tools that use CWL as workflow definition are Galaxy and Toil [25]. Both tools are open source and written in Python as the reference implementation of CWL `cwtool`.

2.2.1 Nextflow

nextflow[3] is a workflow engine that can easily be used from the command line to process user defined data pipelines. The developers state their software "enables reproducible computational workflows"[25].

The tool is written in Groovy and instead of static grammar like workflow definitions it provides the user with the object-oriented style of Java[24]. *nextflow* follows the approach of creating programmable and modular structures of workflows.

[3] `https://nextflow.io`

On their website Seqera Labs define their workflow engine as followed: “Nextflow is a free and open-source software distributed under the Apache 2.0 licence, developed by Seqera Labs. The software is used by scientists and engineers to write, deploy and share data-intensive, highly scalable, workflows on any infrastructure.”[30]

nextflow uses workflows to run a user specified data pipeline. Users can define processes with inputs and outputs that will run either Groovy code or another *nextflow* or any other script (e.g. shell or Python). Workflow definitions contain processes that will be executed. Processes can be joined via pipes in a shell script style. For input data *nextflow* uses channels as stream-like objects. Channels can contain different data types, such as numbers, strings and file paths.

A simple workflow written in the *nextflow* DSL 2 that uses a shell script for writing to a text file and reading it again could look like presented in Listing 2-3.

Listing 2-3: Example *nextflow* Workflow Script

```
1  #!/usr/bin/env nextflow
2  nextflow.enable.dsl=2
3  process foo {
4      input: val cheers
5      output: path '../../../foo.txt'
6      shell:
7          '''
8          # nextflow will create working directories for
9          # each task where this file would go
10         echo !{cheers} >> ../../../foo.txt
11         '''
12 }
13 process bar {
14     input: path x
15     exec:
16         x.eachLine{ line -> println line }
17 }
18 workflow {
19     channel.of('Hello world!', 'Hi!', 'Bonjour!') | foo | bar
20 }
```

The above example emits three greetings via a channel and passes them to the first process `foo`. The foo process uses basic shell functionality to write one greeting at a time to the file `foo.txt` which also defines as the output of the process. Outputs are equivalent to return values except processes can have multiple. By using the pipe operator the output declaration of foo can be used as the input of process bar because the types match.

The `bar` process will take the input file x and prints all file lines via Groovy code. Both processes will be invoked for each entry but the second process bar will not start before all three foo processes are finished Channels do not have to pick the items in the same order of the declaration as seen in the output in Listing 2-4

Listing 2-4: Command Line Output from hello.nf

```
1
2  alena@lenovo:~/source$ nextflow run hello.nf
3  N E X T F L O W  ~  version 23.04.3
4  Launching test.nf [extravagant_cuvier] DSL2 - revision: eee7e769a5
5  executor >  local (6)
6  [99/dd4cc7] process > foo (2) [100%] 3 of 3
7  [ba/d42f91] process > bar (1) [100%] 3 of 3
8  Bonjour!
9  Hello world!
10 Hi!
11 # ... 3 times
```

As stated previously the channel took the items in different order and emitted them to the `foo` process which written them in this order to the output file.

On the other hand of the differing execution order, *nextflow* enables you to continue with the work by writing checkpoints during the execution which can be reused or resumed if an error occurs.

Nextflow can run tasks parallel out of the box on different platforms like Amazon AWS, Kubernetes, Google Cloud without the need of modifying the scripts.

Today, plenty of researchers at sequencing facilities use *nextflow* for their workflow system [46]. To make the individual workflows from different people shareable and curated the project nf-core started. The goal of nf-core is to share portable and reproducible pipelines that will work regardless the operating systems, hardware or *nextflow* versions [46]. As of 2023 there are 86 predefined curated pipelines for data analysis available as part of nf-core [47].

2.3 Literature Review

In the research field of grammarbased and coverage guided fuzzing there are different studies and projects that use structured input files for testing.

In this research field there are mainly two different approaches to create new test inputs for the SUT. Inputs can be created by mutating existing files or data or new test can be created from scratch which is generally referred to as generational approach.

The following sections presents a selection of projects and researches focusing either the field of fuzzing in Java or using grammar aware strategies or combination of both.

Csmith. A test generation tool used to test C compilers is Csmith, develop by researchers of the University of Utah. Yang et al. created Csmith to improve the quality of C compilers [5]. The test cases generated by CSmith follow a specific standard for C programs, the C99 standard.

The tests were run randomly without using a guided fuzzing approach, however the researchers measured the coverage their tests have reached on the tested compilers. The researchers have found that their random generated test files did not improve branch, function or line coverage of the source code of GCC and LLVM when added to existing test suites.[5] Over three years of testing the developers have found more than 325 bugs with those test cases.

Tribble. One fuzzer that is capable of coverage guided fuzzing JVM-based application is Tribble [48]. This research is the result of the dissertation of Nikolas Havrikov about *Grammar-based Fuzzing Using Input Features* [26].

He defined the grammar coverage metric k-path coverage. By combining this metric with a graph representation of a grammar, it can be used to make judgements about the variety of a single input or even a set of inputs.

This measure can be used in an algorithm to maximize this coverage. To evaluate this approach a blackbox fuzzer was written and compared to the closest blackbox fuzzing approach in terms of code coverage.

Tribble is written in Java and Scala. To collect coverage information, Tribble uses the widely used Java Code Coverage library JaCoCo. Users can use tribble to test different Java based applications by writing a grammar in BNF (see Section 2.1.1) or Scala DSL (see Section 2.2) format.

While Havrikov shows that his approach has found different bugs than his compared baseline and measured "impressive " code coverage [26], yet this interesting approach has not found any application in other studies yet.

OSS-Fuzz. A continuous fuzzing project for open source software is hosted by Google and will run fuzzing campaigns on tests for the projects provided by maintainers. OSS-Fuzz will build and run the tests in Google Cloud Service buckets with the distributed fuzzing infrastructure ClusterFuzz[49]. The developers of OSS-Fuzz claim that the project has helped to identify over 10000 vulnerabilities in 1000 different projects. [49]

OSS-Fuzz provides a platform for continuous coverage guided fuzzing for different applications written in a variety of languages such as C/C++, Rust, Go and Java/JVM. But whether the tests are truly grammarbased relies on the test cases provided by the developers of the open source software.

OSS-Fuzz was used as a source for benchmark programs in a wide set study *On the Reliability of Coverage-Based Fuzzer Benchmarking* by Böhme et al. [27] and tested those on a different fuzzer evaluation platform called FuzzBench. The researchers explored the interrelationships and level of agreement on coverage guided und bug finding guided fuzzing techniques over campaign length and retries. The came to the conclusion tha it does not seem to have any benefit of running a fuzzer program combination more than 20 times and that rather shorter fuzzing campaigns may not strongly agree with the results from longer campaigns.

jsfunfuzz. Jsfunfuzz is a grammarbased blackbox fuzzer for testing JavaScript engines. When it was first introduced in 2007 it had a large impact [3] as it found about 280 bugs and over 1000 vulnerabilities in th Mozilla JavScript engine [50].

The JavaScript generator was highly adapted for the Mozilla JavaScript engine but can be used to test other script engines. For example Superion [6] used it as baseline when testing WebKit, ChakraCore and JerryScript. Holler et al. used it 2012 for LangFuzz [3] as comparison. Today an updated version of Jesse Rudermans tool is maintained by Mozilla and public available on GitHub.

Langfuzz. LangFuzz works by combining generation and mutation when creating test inputs. The research result of two researchers from the Saarland University and one developer from Mozilla is published under the title *Fuzzing with Code Fragments* [3].

By using a given language grammar different code fragments can be learned by parsing tests from a given test suite or sample code base [3]. The test generation is based on replacing different statements in one of the beforehand learned examples.

As LangFuzz is only a proof of concept it will need a simplified version of the language grammar because the used subsystem ANTLR parser will only support a subset of the ANTLR syntax. But the tests run on the Mozilla JavaScript engine have found 105 vulnerabilities.

The approach can be adapted to use another grammar as long as it is weakly typed, such as PHP. LangFuzz has found 18 defects in the PHP interpreter [3].

Grammarinator. Written in Python Grammarinator creates test files for fuzzing using an ANTLR v4 grammar. Grammarinator can be installed and used as a command line tool via pip. In their research paper the developers evaluated their generational approach with the Fuzzinator framework. Fuzzinator is a blackbox fuzzing tool for Python libraries.

The researchers present 89 issues by using the generated test files in the JavaScript engine JerryScript.[28]

Superion. Superion is a project that uses grammar-aware greybox fuzzing to also test different JavaScript engines like JerryScript, WebKit, libplist and ChakraCore. Additionally, in their case study the developers evaluated their approach on XML engines. The strategy for generating test inputs is to manipulate and trim existing input files using grammars[6]. The project is written in C++ and will use ANTLR as a parser for manipulating inputs by replacing subtrees

For the mutation the developers compared two different strategies: dictionary based and tree based. The developers compared their approach with AFL and found that the valid inputs improved code coverage and bug finding capabilities.

QuickFuzz. QuickFuzz is a blackbox fuzzer for testing software and libraries handling common image and media file formats [4]. Written in Haskell it uses MegaDeTh to derive information about all nested types to create a working instance. With this tool QuickFuzz is able to create and mutate filetypes like Gifs, Jpegs or HTML and PDF documents. The variety of different file formats is created by using existing libraries for those types.

QuickFuzz uses existing fuzzers for execution and can fuzz software written in any language. In their published research from 2016 the developers present 14 different security issues they have found in different libraries and browsers that handle different file formats [4].

MoFuzz. MoFuzz is a project that applies fuzz testing to Model Driven Software Engineering (MDSE) tools [15]. Using three different approaches for test generation one idea is to use coverage-guided mutations on models and was evaluated with two different variants. The third strategy was to use grammarbased fuzzing with automated model generation.

The researchers have applied their strategy to real world applications and have found that using a model based approach covers more code of the analyzed software and triggers more crashes than other coverage guided fuzzers [15]. MoFuzz is written in Java based on JQF and can be applied to MDSE tools that build upon the EclipseModelingFramework.

3 Implementation

This Section will describe the used libraries and required modifications for the implementation. Further we will take a look at the defined generators and grammar to produce input data.

As described before the target software of the fuzzing campaign is written the JVM-based language Groovy so the implementation for the tests is also written in Groovy.

As grammarbased fuzzing with coverage guided fuzzing techniques has achieved good results in finding vulnerabilities in JVM based applications [15, 40, 8, 29, 10] the implementation to test *nextflow* follows this approach.

3.1 Used Libraries

The project is build upon Java 11 and the used Groovy version is 3.0.15 to be compatible with *nextflow* 23.04.2. Besides those two necessary libraries we also need a fuzzing framework: in this set up we will use a clone of JQF 2.0. All mentioned repositories and clones can be found online[4].

3.1.1 JQF

The whole implementation is set up to use JQF as a fuzzing driver. To be a suitable driver during testing the implementation contains a main method which will start the fuzzing campaign. This allows us to add own methods to analyse occurred events.

To have faster access to test files and the exceptions that the SUT might have thrown during processing, we can add to JQFs handleResult method. Besides JQF saving the generated input used to generate the current script file, we will log the file name and exception. If the tested inputs have not raised any error, they are deleted from the hard drive.

3.1.2 GramTest

To create syntactically correct testcases we need a suitable generator. One open source Java based project published under Apache License 2.0 is *GramTest* written by Dr. Asankhaya Sharma [5].

GramTest enables users to create test inputs based on a given grammar definition written in BNF. It is build on top of ANTLRs parse tree visitor and will generate test strings by traversing the tree and making random choices on which path to use.

[4] `https://github.com/schemmea`
[5] `https://github.com/codelion/gramtest`

To make this a suitable generator for JQF to use during fuzzing some slight adjustments are needed. As mentioned in Section 2.1.2 JQF needs to be able to reproduce the inputs. To achieve this, the test generation in the `GeneratorVisitor` class has to use the `junit.quickcheck.random.SourceOfRandomness` when shuffling paths or deciding which option to take (see Listing 2-2).

The main benefit of using *GramTest* is the easy adaption of the ruleset to generate new test scripts, meaning no implementation is needed.

3.2 Generators

For the fuzzing campaign two generators were implemented to measure how worthwhile it is to ensure that the tests are semantically correct and executable.

In the following Sections we will call this approach which favors semantic correctness "semantic generator". We will compare this to the "syntactic generator" which will generate simpler and not necessarily executable tests files.

3.2.1 Semantic Generator

The semantic generator is the main investigation of this thesis and the other proposed generators and tests are used to evaluate the quality of it. The generator will use *GramTest* with an especially designed grammar and some postprocessing to produce a workflow which calls the defined processes with suitable scripts.

Grammar for Semantic Generator. The grammar used for *GramTest* to generate individual test inputs is only a small representation of the *nextflow* DSL2. It is quite simplistic compared to the capabilities of *nextflow* to sort, convert and manipulate input data for different tools or even execute tests in containers in a user specified cloud tool. The grammar is kept simple due to the fact that *GramTest* will use many bytes from the `SourceOfRandomness` during one generation. Also by increasing the complexity and depth in the grammar definition the time for each generation will increase exponential.

To redeem more generation speed constant strings are written inline in each rule. Declaring rules with constant string values to use in other rules would otherwise create a larger syntax tree that has to be traversed. Additionally, *GramTest* uses the rule count to determine how many paths on the tree will be visited to create a test string, to ensure every rule can be visited on average.

In the used definition in one generated document takes about 5430 KiloBytes from the source of randomness to be generated. Most of them are used for generating names for processes and channel items.

Following grammar (Listing 3-5) was used to generate the test inputs .

Listing 3-5: Grammar Used in Semantic Test Input Generator - shortened for readability

```
<nextflow> ::= "#!/usr/bin/env nextflow" <processes> <workflow>
<processes> ::= <process> <process> | <process> <processes>
<process> ::= <process1> | <process2>
<process1> ::= "process " <identifier> " {" <body> "}"
<identifier> ::= <char> | <char> <identifier>
<body> ::= "input: val variable output: val variable" <content>
<content> ::= <templatescript> | <script>
<templatescript> ::= "script:" " template 'scriptname'"
<quotes> ::= '"""' #constant to prevent regex misinterpreting
<script> ::= "script:" <quotes> "scriptplaceholder" <quotes>
<process2> ::= #analogue to <process1> but with two inputs and outputs
#...
<workflow> ::= "workflow " <workflowbody>
<workflowbody> ::= "{" <channel1> <channel2> "}"
<channel1> ::= <channel> ".set{ch1}" "ch1 placeholder1"
<channel2> ::= <branched> ".set{ch2}" "ch2 placeholder2"
<channel> ::= "Channel.of" <open> <channellist> <close>
<branched> ::= <channel> ".multiMap { it -> one: two: it }"
<channellist> ::= <item> "," <item> | <item> "," <channellist>
<item> ::= <number> | '"' <identifier> '"' | <uint> ".." <uint>
#...
<char> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<uint> ::= 0|1|2|3|4|5|6|7|8|9
<open> ::= "("  #constant to prevent regex misinterpreting
<close> ::= ")" #constant to prevent regex misinterpreting
```

In this grammar definition there are two possibilities for the scripts to be inserted. Firstly via the `template` keyword which refers to a predefined script file in the templates-directory of the current working directory. The second option is to read a random script file and write the content in the script block wrapped with `"""`.

A test case generated by *GramTest* with this grammar will look like presented in Listing 3-6 (formatted for readability):

Listing 3-6: Test Case Generated by Grammar in Semantic Generator Before Postprocessing

```
#!/usr/bin/env nextflow
process x {
    input: val variable
    output: val variable
    script:
        """
        scriptplaceholder
        """
}
process yz_twovars{
    input:
```

```
12          val variable
13          val variable2
14      output:
15          val variable
16          val variable2
17      script:
18          template 'scriptname'
19  }
20  workflow {
21      Channel.of("x",0..5,"j".."o").set{ch1}
22      ch1  |  placeholder1
23      Channel.of("m".."x",8..5)
24                  .multiMap { it -> one: two: it }.set{ch2}
25      ch2  |  placeholder2
26  }
```

This script is not executable by *nextflow* because of the placed placeholders in line 7, 18, 22 and 25. This need to be replaced with real script contents or script names and in the workflow component we need to call the two generated processes `x` and `yz_twovars` from above.

Ensuring the Executability. After generating the string via *GramTest* with the grammar presented in Listing 3-5 the next step is to make sure the processes have linked actual executable scripts.

The semantic generator will search for the script name placeholder and from a pool of prepared files replace it with a random file name. For the other script placeholders between `"""` from the same pool one random file will be picked and the content written to the placeholder. While replacing the script placeholders a distinction is made between processes with one or two inputs so that the selected script will have the same amount of inputs.

The used template scripts are basic bash scripts with operation like `df`, `ps aux`, `echo` or simple calculation with the given inputs from the process. The provided set of template files includes 28 files. Inputs and outputs are have a constant name called `variable` to ensure that the defined inputs are usable in the scripts and processes can be joined.

After the processes are executable tasks for *nextflow*, they need to be called from the workflow-block in script. Via regular expressions all randomly created process names are collected and sorted by one and two inputs. Processes with one input replace `palceholder1` joined via pipes (`|`) and those with two inputs replace `palceholder2` and joined in the same way. In the given example in Listing 3-6 there is only one process with one input and one process with two inputs. So in the finished workflow body will the process call looks like this: `ch1 | x` and `ch2 | yz_twovars`.

When more than two processes with one input where generated, we also can join them in the workflow with a two inputs process with a method like notation: `twovars(a(ch2.one), b(ch2.two))`. Channel a and b will each use the data of one branch of the channel `ch2`.

Although at first sight it looks like all inputs generated by this generator will be executed with no error, due to the nature of random generating tests not every run will succeed. With that said the test input is ready for fuzzing.

3.2.2 Syntactic Generator

The syntactic generator uses a similar grammar definition as the semantic generator (Section 3.2.1). The main difference is that the grammar (Listing 3-7) does not contain any placeholders for scripts or process calls. We will use this as comparison to the semantic generator as there no comparable generator for script files in the *nextflow* DSL2.

If the grammar used the same `identifier`-rule as the semantic generator it would produce scripts that are not executable most of the time. Generated process names would have an extremely low chance being generated a second time in the workflows process calls. To make up for this only the first five letters of the alphabet are used with one char in the rule so that the probability of calling a generated process is increased drastically.

The third difference compared to first described grammar lays in the `script`-rule where just a set of possible template scripts are presented.

Listing 3-7: Grammar Used in Syntactic Test Input Generator - shortened for readability

```
1  <nextflow> ::= "#!/usr/bin/env nextflow" <processes> <workflow>
2  <processes> ::= <process>  |  <process>  <processes>
3  <process> ::= "process" <char> "{" <input> <output> <script> "}"
4  <input> ::= "input: val variable"
5  <output> ::= "output: val variable"
6  <script> ::= "script: template " <tick>  <magicstring> <tick>
7  <magicstring> ::= "script1" | "script2.sh" |
8                   "script3" | "script4.txt" | "script5"
9  <workflow> ::= " workflow " <workflowbody>
10 <workflowbody> ::= "{" <channel> <pipe> <calls> "}"
11 <channel> ::= "Channel.of" <open> <channellist> <close>
12 <channellist> ::= <item> "," <item> | <item> "," <channellist>
13 <item> ::= <uint> | '"' <identifier> '"'
14 <calls> ::= <char> <pipe> <char> | <char> <pipe> <calls>
15 <char> ::= a|b|c|d|e
16 <uint> ::= 1|2|3|4|5|6|7|8|9
17 <open> ::= "("  #constant to prevent regex misinterpreting
18 <close> ::= ")" #constant to prevent regex misinterpreting
19 <pipe> ::= "|"  #constant to prevent regex misinterpreting
20 <tick> ::= "'"  #constant to prevent regex misinterpreting
```

Besides not containing any placeholders there is no rule in the grammar that considers processes with to input variables.

This grammar handed to *GramTest* will generate finished test inputs right away and will look like Lsiting 3-8.

Listing 3-8: Test generated by syntactic grammar

```
1  #!/usr/bin/env nextflow
2  process a{
3      input: val variable
4      output: val variable
5      script: template 'script4.txt'
6  }
7  process c{
8      input: val variable
9      output: val variable
10     script: template 'script5'
11 }
12 workflow {
13     Channel.of("a","b",7) | a | b
14 }
```

While this script follows the *nextflow* syntax it will not be a valid input during fuzzing because the process called in the workflow section does not exist.

One substantial benefit of using this smaller grammar is that it will only use about 26 KiloBytes to generated one test input. In the evaluation (see Section 4) we will see if this approach is superior or inferior to the semantic generator.

3.2.3 Wrapping Command Generator

As described *nextflow* is a command line tool which has additional options for maintaining projects. To include those in the fuzzing campaign both grammar generators are wrapped in a `CommandGenerator` which chooses from a given list of *nextflow* commands. The main idea behind this setup is to test the whole workflow engine and not only its script parsing capabilities.

The used commands also include parameters:

- run $< script > (-with-docker|-with-report|-with-trace|-with-timeline)$
- clean $[a-z]$
- secrets $(list|delete[a-z]|get[a-z]|put[a-z]|set[a-z])$
- ...

When the run-command gets selected a test script will be generated and saved to a dedicated output directory. In this directory the mentioned template files are also stored in a subdirectory called “templates”. The run command will also be started with an additional parameter `-cache false` to decrease the memory usage by *nextflow*.

All commands with their options are passed to the *nextflow* main launcher in the fuzz unit tests. Possible commands that this generator will return are stored in an array of strings. The `run` command will look like this:

```
["run", "scriptname.nf", "-with-timeline", "-cache", "false"]
```

3.2.4 AFL Fuzzing in Java with JQF

Another way of evaluate the semantic generator is to compare the results to basic grey-box fuzzing from AFL. For fuzzing a JVM based program with AFL the developers of JQF have written an adapter for AFL to use their own code for instrumentation during tests.

This driver defines a two proxies over files to AFL. In the output proxy the driver saves the status for a mutated input as a feedback for AFL. From the input proxy is read, if AFL ready for the next test. With this driver setup, a target program can be fuzzed in a JQF like environment via unit tests.

For this fuzzing tests a separate JAR is used which will not contain the source code from the grammar generators. JQF-AFL fuzzing will take a set of random seed files and mutate these files by changing bits in the content and pass this as a stream to the unit test.

This tests will focus on executing those mutated script files because we want to ensure that more than the command parser will be executed. If these tests also mutated commands via byte manipulations probably only random utf8 characters would be passed to nextflow which would not trigger a deeper execution of source code.

If a test has triggered something interesting (an exception or more coverage) for AFL the current input stream is saved in the queue directory for further iterations.

Additional to the grammar based test with the proposed generators the AFL tests have the parameter `-ps 1`. This tells *nextflow* to use only one thread in the thread pool while running scripts because the instrumentation does not work well with multiple threads running[43, 51].

4 Experiments and Evaluation

In this section we want to evaluate the proposed implementation from Section 3. To do this, we want to primarily answer the following research questions:

RQ1 How does the proposed technique of the semantic generator performs in terms of generated valid inputs?

RQ2 How does the proposed technique of the semantic generator performs in terms of code coverage?

RQ3 Can the described techniques discover bugs behind the parsing and compiling stage of *nextflow*?

RQ4 Do we observe a better performance regarding RQ1-RQ3 by using a coverage-guided fuzzing technique?

To answer RQ1, RQ2 and RQ3 we will use a fuzzing-campaign run with JQF. Due to the nature of random testing one single run with different input data the test results are not necessarily reliable. To overcome this varying performance, we follow Klees et al. [20] recommend guided lines and also considered the results presented by Böhme et. al in their study on the reliability on coverage-guided fuzzing [27]. For all experiments the configuration described in Section 4.1.1 and Section 4.1.2 was used.

With both presented generators the two different configurations ran 20 times on the virtual machine for one hour each. The test set-up ran on a linux virtual machine with an AMD Epyc Processor (version 23.1.2, 8 cores), 32GB memory running Ubuntu 22.04.3 LTS.

4.1 Experimental Set Up

4.1.1 JQF Configuration

Although JQF is a tool that can be used as is, to be used with the implemented generators and the selected target, some configuration are needed.

The tests will use a JAR with all necessary libraries included built via Gradle. The main class starts the fuzzing run with the`ZestGuidance` (see 2.1.2) to control the mutation of the random input stream for the generators.

The maximum of bytes prepared for the random input stream is set to at least 5,5 MegaByte (see 3.2.1) via `-Djqf.ei.MAX_INPUT_SIZE`.

The maximum memory for the JVM was set to 7 GigaBytes via the `-Xmx` option. For coverage calculation the`ZestGuidance` uses the FastNonCollidingCoverage described in Section 2.1.2.

Which classes should be instrumented or excluded by the JavaAgent can be defined in a separate configuration file. Publications ([9, 8]) in the Java based fuzzing research set up the configuration is set up for the fuzzer to use coverage information about the SUT and its used libraries, to decide which input to save for next tests. The idea behind this is that an input that covers new external library code will more likely also cover more code of the SUT [9] when paired with a coverage-guided fuzzing algorithm.

However, this observation could not be made in beforehand made tests with the described set-up for this thesis. The tests included 20 runs with the same set-up and a configuration to instrument all libraries. The difference between coverage-guided (Zest-Algorithm) and non-guided (blackbox) where about 0.5% and without any statistical significance.

The used configuration in this campaign will exclude all calsses, modules and libaries except from *nextflow*. This should improve execution speed and allow us to have better insights on the fuzzing target *nextflow*.

As we want to use the `ReproGuidance` (see Section 4.1.3) we need to exlude some static classes from *nextflow* that will cause errors from JQF when it tries to instrument them: `HistoryFile`, `ProcessConfig`, `TaskTemplateEngine`, `ScriptRunner`, `AssetManager` and three different exception classes from *nextflow*.

One addition that is made to the configuration as well are the generated scripts themselves. *nextflow*-scripts are compiled as Groovy class and would be count as new covered branches during testing. Also, *nextflow* will generate and compile config and dummy files for each script in the current session. Those were also excluded from being instrumented.

4.1.2 Setup for Generator Comparison

To evaluate the implemented generators run in a setup similar to the one described in the paper *Semantic Fuzzing with Zest* [8].

The generators where both tested with the `ZestGuidance` and with no guidance at all. When running JQF with no guidance coverage is still calculated but not used as feedback for generating the next test inputs. No guidance means this is not a guided fuzzing test, that will not apply algorithms on maximizing a specifi metric like coverage or bugs.

Both generators where used with the same unit test and ran guided by the `ZestGuidance` and in comparison with no guidance. The used unit test is set up as shown in Listing 4-9.

Listing 4-9: Unit Test Used in Fuzzing Campaign with JQF for Both Generators

```
1      @Fuzz
2      public void testNFCommand(@From(CommandGenerator.class)
3                                      String[] command) {
4          if (command[0] == "run") {
5              //avoid try catch (Throwable) in Launcher
6              Launcher launcher = new Launcher().command(command)
```

```
7
8                 CmdRun myRunner = new CmdRun();
9                 myRunner.setArgs(command.tail().toList());
10                myRunner.setLauncher(launcher);
11
12                myRunner.run();
13            } else {
14                int status = new Launcher().command(command).run();
15                Assume.assumeTrue(status == 0)
16            } }
17        @After
18        public void cleanUp() {
19            //session is not destroyed after exception
20            def sess = (Session) nextflow.Global.getSession()
21            if (sess != null) {
22                sess.cleanup()
23                sess.destroy()
24            }
25            nextflow.Global.cleanUp()
26            nextflow.Plugins.stop()
27        }
```

As we want to test the whole workflow engine and not only the parsing capabilities we use the main entry point of nextflow. The Launcher (Line 6) is also executed when *nextflow* is used via commandline.

In order for JQF to observe triggered failures in the target program, the script execution is not launched via the main entry point of *nextflow*. We use the `CmdRun` class instead, that is also executed under normal conditions. The `Launcher` would catch any `java.lang.Throwable` during execution of any command.

In the whole source code of *nextflow* are 23 Try-Catch-Clauses in 14 classes that will do so and will not necessarily rethrow their caught exception. Some of them are deeper nested that a workaround would be unreasonable. Therefore, we do not expect to see many unique crashes or exceptions during testing.

To tell JQF if the input was valid, the test will use `org.junit.Assume` (Line 15) on the returned launcher status. After every test, the cleanUp-method is called to ensure that no artefacts of unsuccessful runs are still up in the current JVM.

AFL Test Set Up. The described AFL setup (see Section 3.2.4) also run on the same machine twenty times for one hour. The used unit test in these runs is different from the plain JQF-setup (Listing 4-9). AFL did not start fuzzing if an exception occurs in pilot run, so the whole test is wrapped with a Try-Catch-Clause and feedback on the validity is also via `org.junit.Assume`.

Listing 4-10: Unit Test Used in Fuzzing Campaign with AFL

```
1   @Fuzz
2   public void testAFL(@From(InputStreamGenerator.class)
3                  InputStream inputStream) throws IOException {
4
5       String filename = getFileName();
6       Launcher launcher =  new Launcher();
7       try {
8           serializeInputStream(inputStream, filename);
9           String[] args = new String[]{"run", filename,
10                                         "-cache", "false",
11                                         "-ps" , "1"};
12          int launched = launcher.command(args).run();
13          Assume.assumeTrue(launched == 0);
14      } catch (IOException e) {
15          throw new RuntimeException(e);
16      } catch (Exception e) {
17          Assume.assumeNoException(e);
18      } finally {
19          //instead of @After
20          Plugins.stop();
21          Files.delete(Paths.get(filename));
22          def sess = (Session) Global.getSession()
23          if (sess != null) {
24              sess.cleanup()
25              sess.destroy()
26          }
27          nextflow.Global.cleanUp()
28      }}
```

The tested input is taken from an InputStreamGenerator which is in this test case the raw by AFL mutated seed script files that will be written to the disk with a unique name. Afterward, the mutated script can be tested by starting the execution via the `Launcher`.

In this setup the cleanup methods are called in the finally-block (Line 18) the test. Test files are not needed after testing, so they are deleted.

The JQF-AFL driver uses a different coverage metric, so the interesting artefacts from this experiments will be the total covered branches from JQF‘s `ReproGuidance` (see next Section 4.1.3).

4.1.3 Reproducing Runs

The reproducing algorithm from JQF can be used to rerun specific test inputs on a defined testcase to examine thrown exceptions or methods called during tests [51]. In this case it will be used to compare called classes and branches within *nextflow*.

Reproducing JQF tests works by setting up the`ReproGuidance` to replay the used InputStreams whether they were used for the generator as random source or directly as input for the AFL test. This is a suggested method by Rohan Padhye [51], the main developer of JQF, to further investigate produced crashes. The tests are also run with instrumentation via Java Agent for collecting called branches.

As we want to measure the coverage within nextflow we will configure the instrumentation to only collect information about branches executed on nextflow classes. The`ReproGuidance` still uses the first described coverage calculation (see Section 2.1.2).

Normally, as stated in the JQF documentation [51] we would run the `ReproGuidance` on failures and hangs, which we could not observe in this test setup. When running the `ReproGuidance` we can adjust if all internal made calls should be logged or only from tests which had a valid results. After rerunning all inputs in a given set the calls can be saved for further evaluation.

4.2 Results

The analysis uses the beforehand mentioned in total 20 tests for each generator run with `ZestGuidance` and without (noguidance in the following) and 20 tests with the AFL set-up. Also, we will use JQFs ReppropGuidance on saved inputs for AFL and generator tests to analyze coverage within the source of nextflow.

4.2.1 Analyzing Generated Inputs

To gain a better understanding of the measured coverage we first inspect the number of tests run and the overall execution speed for each parameter combination (see Figure 1 and Figure 2).

The number of inputs represent the inputs produced by the source of randomness that were used in the generator (meaning the other nextflow commands like `list`, `clean` are also included in the number of inputs). Solid lines are the mean for every combination of guidance and generator, shaded areas represent a 95% confidence interval. This applies to all following graphs as well if not stated otherwise.

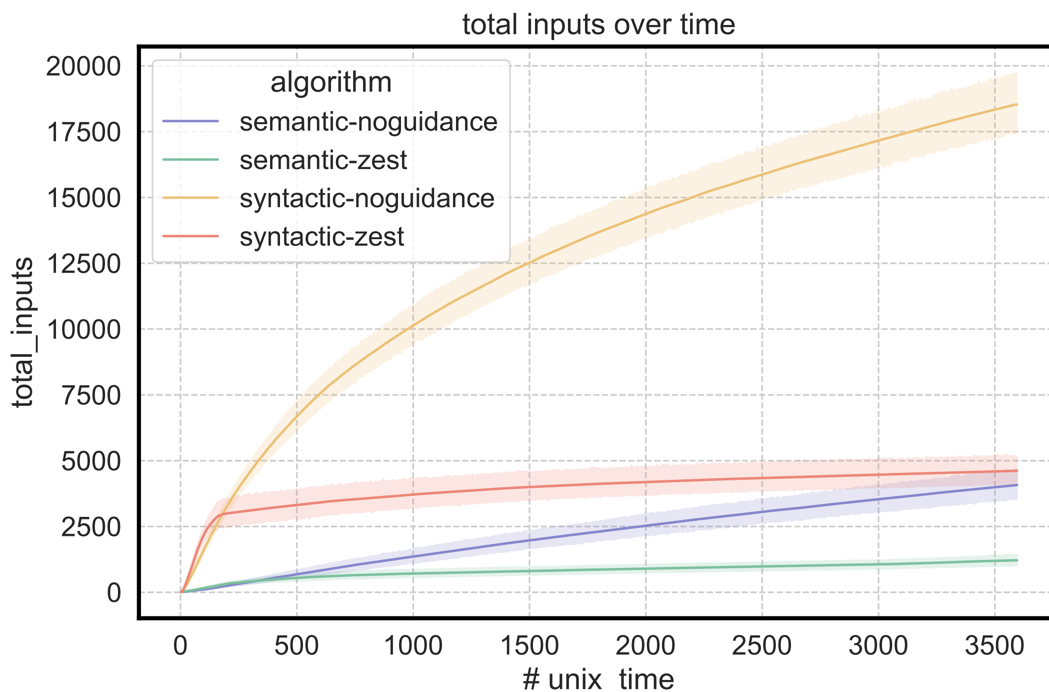

Figure 1: Inputs Generated Over Time

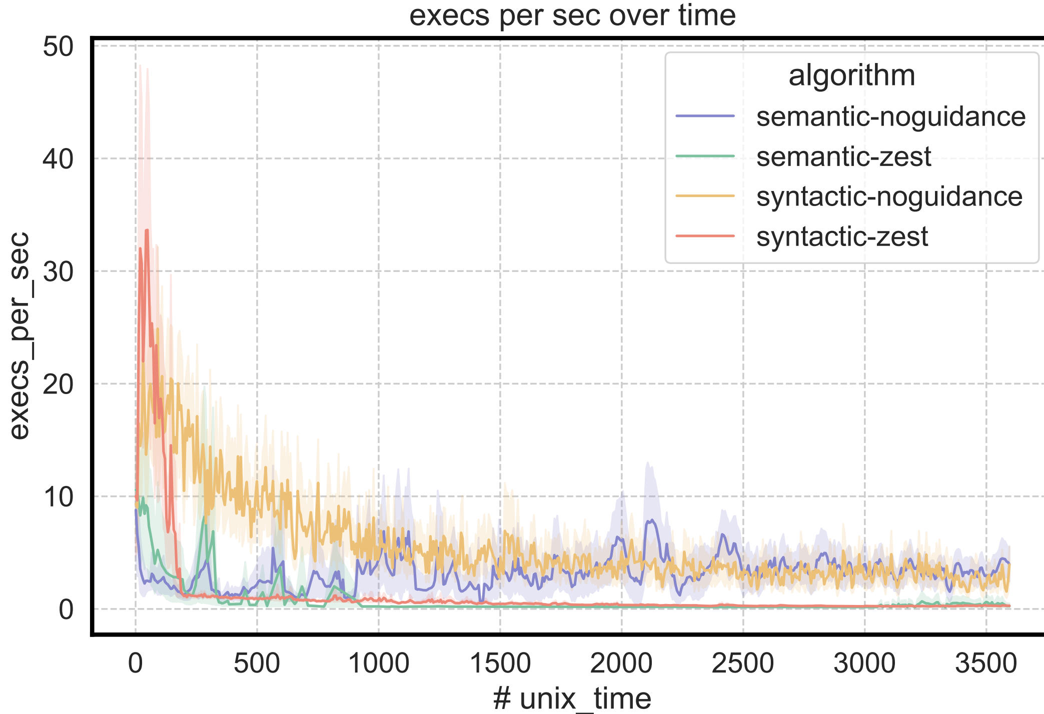

Figure 2: Execution Speed Over Time

Depending on which generator was used with which guidance the number tests executed over one hour varies (see Figure 2). For both tested generators, tests without guidance run faster than the experiments guided by the Zest-Algorithm. So reached the blackbox test with the syntactic generator 18550 executions on average, but with the semantic generator only 4091 (see Figure 1 and detailed information in Table 2). Guided by the Zest-Algorithm both generators have low slope of total inputs over time. In both tested scenarios the syntactic generator produced more inputs than the semantic generator.

The test set-up with JQF-AFL connector measures tested inputs and coverage differently so there is no data about the executions over time, but the absolute numbers of tests done by AFL (see Table 2).

Although the syntactic generator run at much higher speed at the beginning it slowed down to below 5 executions per seconds after half an hour with no guidance. The tests run with `ZestGuidance` with the syntactic generator executed about one test per second. The blackbox tests of the semantic generator executed between 0.5 and 8 tests per second. In the coverage-guided execution the execution speed was less than one execution per second.

To further investigate if the faster and more frequent executions of the syntactic generator have any benefits we take a look at the ratio of valid to invalid test inputs (see Figure 3). Valid tests mean that the return value of the Launcher had no error value nor have the generated scripts raised an exception.

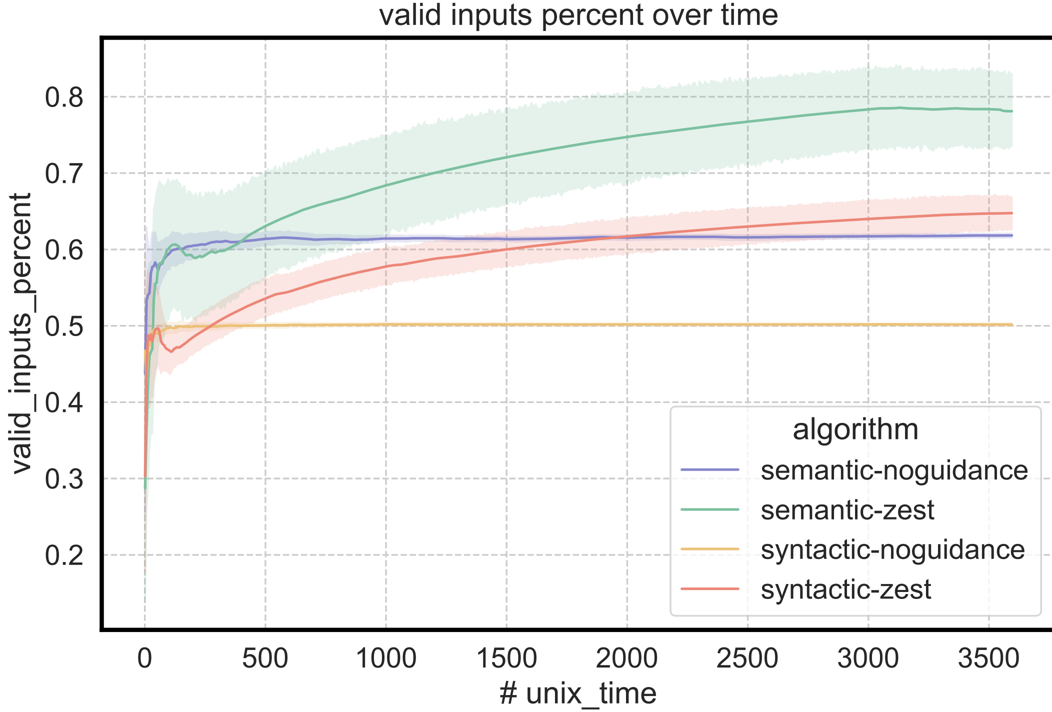

Figure 3: Valid Inputs in Percent Over Time

Figure 3 shows the percentage of valid inputs for each combination. For the blackbox tests (noguidance) the percentage of valid created tests is the same over the whole duration. And for both generators the percentage raises over time as the `ZestGuidance` will generate more of those inputs. In Table 2 the average final values of total and valid inputs over all 20 runs are presented.

Table 2: Average Inputs Tested Over All Runs

Test	Total Inputs	Valid Inputs	Relative Valid Inputs
semantic-zest	1225 (38.76%)	922 (33.62%)	78.22% (14.54%)
syntactic-zest	4619 (27.26%)	2947 (20.52%)	64.78% (7.89%)
semantic-noguidance	4091 (31.87%)	2532 (32.03%)	61.84% (1.07%)
syntactic-noguidance	18550 (14.28%)	9307 (14.41%)	50.17% (0.81%)
afl	4086 (1.48%)	not measured	not measured

Values in brackets represent the Relative Standard Derivation for the final results of all runs

In Table 2 we can see that the syntactic generator produced as much invalid inputs as valid inputs over time when running without any guidance to increase coverage or executability. When guided with the Zest-Algorithm about 64.78% of the inputs generated by the syntactic generator where valid.

Without any guidance the semantic generator generated also about 61.84% valid inputs but in total number there much fewer executions. The `ZestGuidance` executed also much fewer tests with semantic generator than both tests with the syntactic, but about 78.22% of those inputs were considered valid.

The semantic generator had an overall 11.67% better rate of valid inputs than the syntactic generator without any guidance that would maximize the execution success like Zest. We can assess statistical significance in the measured difference by using th Mann-Whitney U test over all final valid-total ratios of each run without guidance with $p < 10^{-8}$.

As for the semantic generator fewer tests were executed, we see a higher derivation in the valid-total ratio in Table 2.

Answer to RQ1. By making sure the generated nextflow script files could actually be executed we achieved a significant improvement in the ratio of valid and invalid inputs compared to the simpler syntactic generator approach. However, this improvement has the downside that only about 22% of the syntactic test are executed with a higher variation.

4.2.2 Analyzing Coverage

To answer RQ2 and to evaluate if the increased data consumption in test input generation has any advantage over the faster execution, we need to analyze the coverage that both generators reached over the tests.

Although the syntactic generator run more tests with both guidance set-ups, the semantic generator reached more coverage faster during testing (Figure 4).

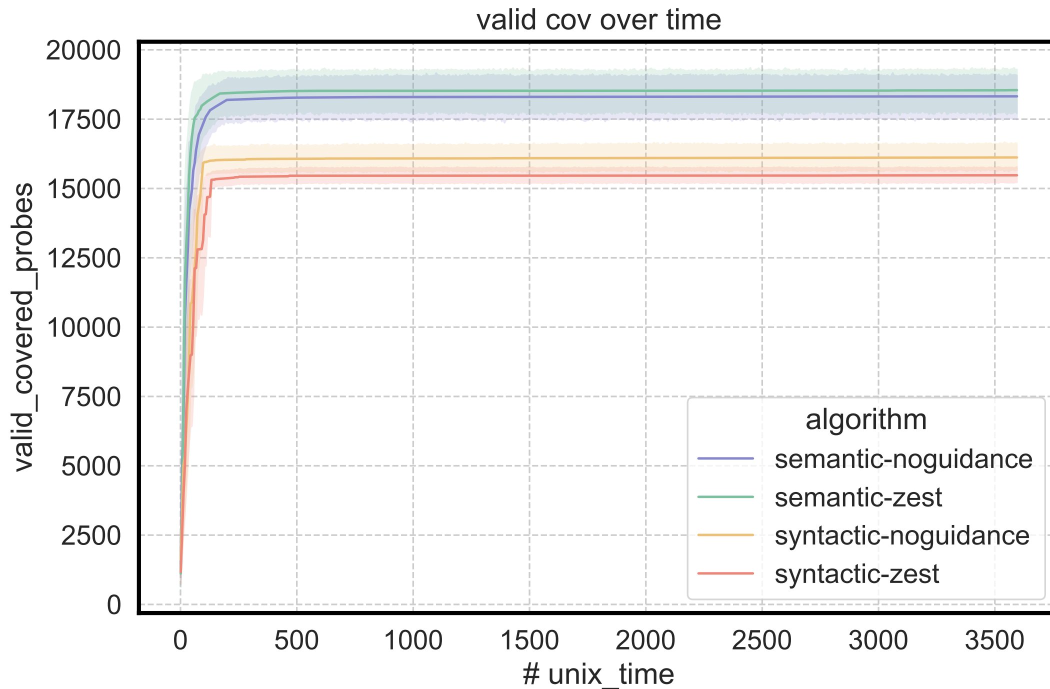

Figure 4: Valid Coverage Over Time

In Figure 4 we can see that for all tested combinations the coverage converges within the tested timeout. This is a common picture of coverage measurements in fuzzing studies that uses JQF as a fuzzing framework or as baseline [10, 40].

While the syntactic generator converged at approximately 16100 covered branches the semantic generator reached 18300 branches with no guidance on average. So we can say for certainty that the semantic approach covers more of the chosen target than the syntactic approach. To proof statistical significance we use the Mann-Whitney U with $p < 0.001$ test for maximum number of valid covered probes over the 20 experiments for the semantic and syntactic generator. The Zest-Algorithm reached 18500 covered branches with the semantic generator and only 15400 covered branches on average with the syntactic generator.

To finally answer RQ2, we compare the branches executed (tracked by the JQF `ReproGuidance`) for the saved inputs for all used generators and AFL. The information about the branches that AFL has covered was recorded by using the `ReproGuidance` on the queue of AFLs prepared inputs, as we could not measure any crashes directly with the presented set-up.

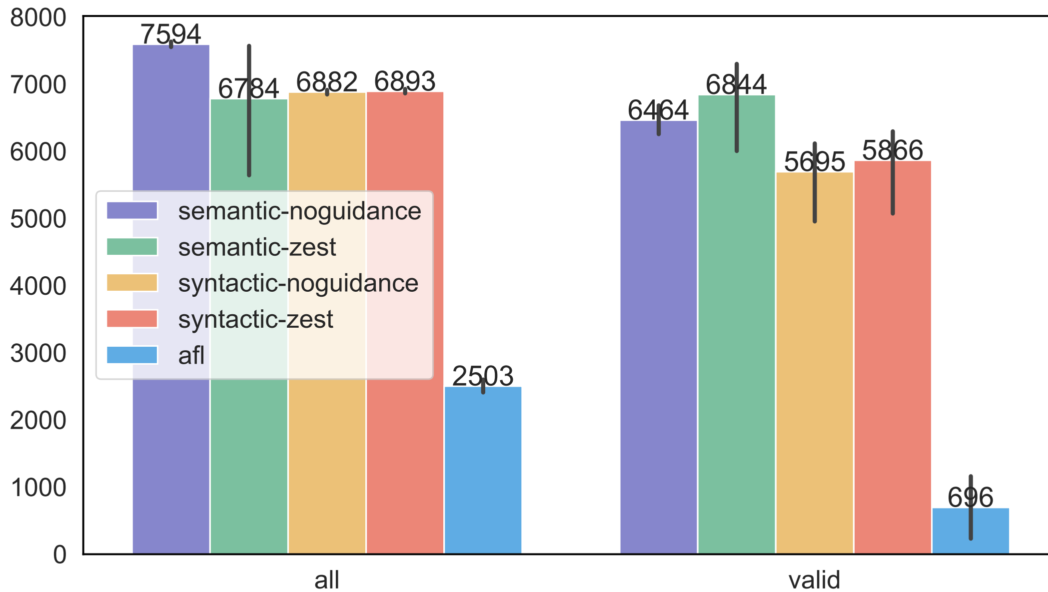

Figure 5: Average Number of Covered Branches

In Figure 5 the bars represent the average number of branches covered in all runs and the black lines represent the confidence interval. This data was gathered by using the first introduced coverage measurement algorithm by JQF (see Section 2.1.2). Therefore, the range does not match the coverage measured in Figure 4. This is here used to compare the difference between successful execution of workflows (generator based tested) and only compilation (AFL mutation based test). Comparing the coverage results of AFL and JQF, AFL only reaches about 10% of valid coverage of what JQF has gained over the same time. The coverage of AFL reaches 36% of JQFs coverage when also the invalid tests are considered.

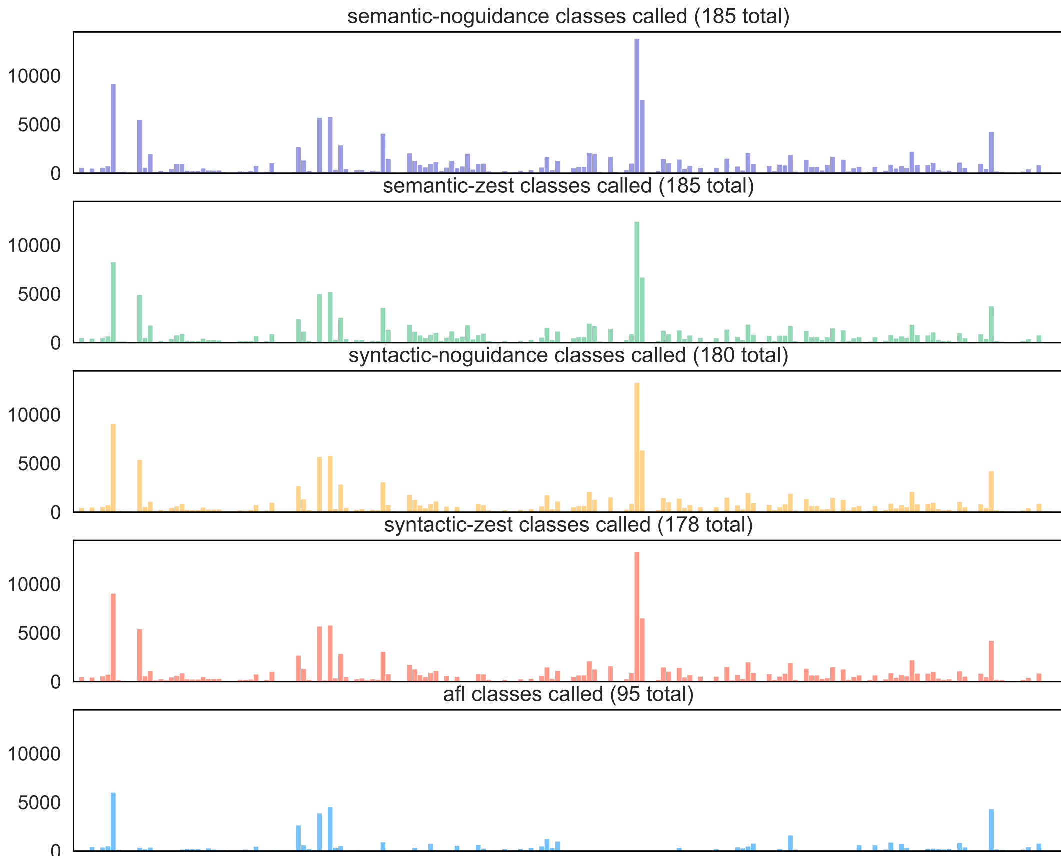

Figure 6: Classes Executed in All Tests

To understand where the difference in the covered branches is in particular, we compare the absolute number of calls of each class in nextflow (see Figure 6).

Figure 6 shows that the four generator based tests with JQF (semantic and syntactic with `ZestGuidance` and no guidance) have all a similar distribution over the classes, while the tests with AFL miss the center of the distribution in the plot. Classes located there in the distribution are responsible for executing scripts. The most calls of the scripts from the generators where made on `nextflow.processor.TaskProcessor` and `nextflow.executor.Executor`. The tests that were handed to *nextflow* by AFL during testing where not valid script files and could be executed.

The presented branches covered are in 185 out of 532 classes in the source code repository of *nextflow* for the semantic generator and for the syntactic generator there were 180 classes called. AFL executed 95 classes of which are all also executed by the syntactic and semantic generators. This distribution of different classes called we also see in Figure 6.

Beside not successfully executing a script, the tests with AFL also made fewer calls on the classes and therefore probably lesser executions overall, although the log files of AFl count 85828 tests in total and about 2496 (see Table 2) executions on average per run which is are more tests than the tests with `ZestGuidance` and without guidance has executed with the semantic generator.

In this tested set-up the grammarbased fuzzing approach covered in all tested combinations more branches and approxiamtely double amount of classes than the random byte mutation of file content by AFL.

Answer to RQ2. The grammarbased approach reached more coverage and wider class distribution than the mutation based approach with AFL. In terms of code coverage we reached within the blackbox tests higher values with the semantic generator than with the syntactic generator.

4.2.3 Analyzing Crashes

To answer RQ3 we will analyze the crashes logged during testing. As described in the setup (4.1.2) the source code of *nextflow* includes Try-Catch-Clauses that causes problems in finding deeper issues in the code.

Table 3 shows that most of the exceptions where thrown while testing with the syntactic generator. Comparing this with the unique crashes JQF has found (Figure 7), we can make the assumption that almost every exception was rated as unique crash. Unique crashes in JQF are exceptions which stack traces that have not occurred yet, because the *nextflow* scripts are compiled as classes the crash happens to be at a before unseen position and is logged.

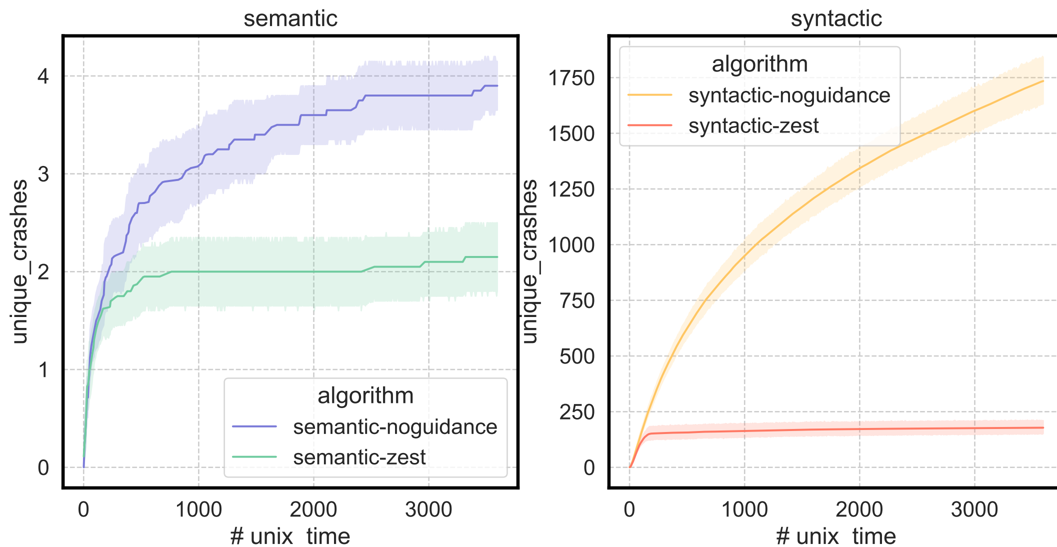

Figure 7: Unique Crashes Logged Over Time by JQF

Commands that did not run a script where wrapped with the main entry point of *nextflow* (the Launcher) and therefore any exception was caught there and only the invalid status was evaluated by JQF . Most of the invalid test inputs in the semantic generator have been those invalid `list` or `drop` commands so less unique crashes where found here.

Only four different types of exceptions have been logged during testing with the generator Set-Up with JQF and one different exception while fuzzing with AFL (see Table 3). Typical fuzzing exceptions like NullPointer or IndexOutOfBound could not be observed.

The exceptions thrown during the tests with AFL are collected from log files that jqf has written during testing. JQF writes all output to a log file, and so we can search for error messages *nextflow* has printed while trying to execute a script.

Table 3: Exceptions Logged by JQF in All Tests

Exception	afl	semantic noguidance	semantic zest	syntactic noguidance	syntactic zest
groovy.lang.MissingMethod	82	-	-	-	-
groovy.lang.MissingProperty	1462	5	-	32189	3391
nextflow.AbortRun	-	20	19	20	20
nextflow.Duplicate-ProcessInvocation	-	-	-	2513	354
nextflow.ScriptCompilation	76792	410	38	10471	1065

As explained the count of unique exceptions with the syntactic generator relates heavily to the number of executed tests (see Table 2). Two of those exceptions occur because of the limited process name generation in the syntactic generator (5 different in total - see Section 3.2.2). When a process is called that is not present in the current script a `MissingPropertyException` is thrown. If an existing process was invoked more than once in thw workflow definition a `DuplicateProcessInvocationException` is thrown. Both of these exceptions were triggered when *nextflow* tried to execute the script. So after it was parsed and compiled to a Java class `Script_XYZ`, where XYZ is a unique identifier for the current Script like `b6cf8a04`. In the systems temporary directory `/tmp` we will find in the *nextflow* directory, matching the unique identifier, the corresponding class file.

The `AbortRunException` is thrown by *nextflow* when an invoked script terminated with an error. The used scripts in the tests are all valid executable scripts, but in some combination with the processed values from the random generated input channels, not all used bash command line tools are executable. Also, if a `grep` command did not match, the return value from the shell is 1 and *nextflow* will count this as unsuccessful execution. All tested combination have almost the same amount of unique the `AbortRunException` is triggered, as they occurred one single time in each 20 runs, and only in one runs with the `ZestGuidance` not.

`ScriptCompilationException` is thrown if a process name was already used or if a channel was build in a false way (e. g. Channel items of numbers must not start wit a leading 0).

The `ScriptCompilationException` will also be thrown if process names equal other reserved words of the Groovy language. The semantic generator found that process names should not be `do`. Also tested via normal execution the error message for the user is `Unexpected input: '{'`. This exception occurs during the compilation process.

The `MissingPropertyException` is thrown during testing with the semantic generator where further investigated as this exception should not occur by the proposed design. When investigating the error messages raised by those scripts the exception states that the property `process` for class `Script_XYZ` is missing. On expectation of the generated script files that produced this error, we find that all the thrown `MissingPropertyException` by the semantic generator correlate with process names that are a reserved word in Groovy: `in`.

When executing these scripts via *nextflows* normal command line interface, users will get the following error: `no such variable: process`. On checking the log files from *nextflow* for running such a workflow the stack trace shows, that this exception will be thrown after the script was compiled successfully. Again, in the systems temporary directory we will find the corresponding class files in the *nextflow* folder for this run.

Inspecting the errors thrown during the AFL test set up we also see the `MissingPropertyException`, those were thrown *after* successful compilation as well. The mutation strategy of AFL also triggered a MissingMethodException, on executing a successfully compiled script. We find this information also in the log files of jqf, the log files that *nextflow* writes during execution. As the test scripts where deleted in the unit test, we can not say for certainty how those test scripts where structured. Using the `ReproGuidance` on prepared inputs of those runs that caused both exceptions we can assume that the test script looked like random unicode characters with linebreaks and brackets. Otherwise, the tests with AFL hve triggered more than seven times the amount of the `ScriptCompilationException` than the syntactic generator without guidance and about 180 times of the blackbox semantic generator test.

Answer to RQ3. In all test cases we have triggered exceptions during execution, meaning after compilation.

The proposed technique of generating semantically correct test inputs has found one bug in the workflow execution phase of *nextflow* as the script with the process name `in` was beforehand successfully compiled.

The tests with the syntactic generator revealed that process definitions and process calls are not evaluated before compilation, but at runtime when the corresponding class is started. Additionally, we found that other reserved words will throw an exception during compilation when they are used as process names like `do`.

The tests with AFL have also triggered exceptions in *nextflow* after the compilation of the test scripts, although the inputs did not look like any real scripts.

4.2.4 Comparing Coverage-Guided and Random Tests

To answer RQ4 we look back at the analysis of RQ1-RQ3.

Regarding the generation of valid inputs (RQ1 - Section 4.2.1) we do see an improvement of valid inputs generated over time in Figure 3 when the generation of new inputs is guided by the Zest-Algorithm. We can confirm this observation also by using Mann-Whitney U test on the final percentage of valid inputs for each run between the `ZestGuidance` and the blackbox (no guidance). For the syntactic generator we can confirm the statistical significance in the difference of relative valid inputs between Zest-Algorithm and no guidance with $p < 10^{-7}$. For the semantic generator we also can confirm statistical significance for this improvement with $p < 10^{-5}$.

Comparing the coverage measured (RQ2 - Section 4.2.2) during the campaign (see Figure 4), we can not see any valid improvement in coverage for both tested generators when using the `ZestGuidance`. Using the semantic generator we achieved with the Zest-Algorithm branch coverage of 18546 (9.82% RSD[6]) branches on average and wit no guidance 18322 (10.05% RSD) branches. This improvement of coverage is so small that it is not of any statistical relevance.

The tests with the syntactic generator reached with Zest-Algorithm a coverage of 15476 (4.10% RSD) branches and with the blackbox approach 16118 (7.00% RSD) This not even an improvement of coverage, but it is also not of any statistical relevance neither.

Regarding the exceptions thrown in the fuzzing campaign (RQ3 - Section 4.2.3) for both generators the `ZestGuidance` has triggered less than the blackbox testing. This can be explained with the lesser executions of the tests in both set-ups (see Table 2). Regarding the `MissingPropertyException` that occurs during execution of a script when a process name is a Groovy language key word, both the blackbox and the test with the Zest-Algorithm found the exception with the semantic generator.

As the Zest-Algorithm prefers valid inputs during tests with the syntactic generator more of the `MissingPropertyException` and `DuplicateProcessInvocationException` where thrown during the blackbox test. But both fuzzing set-ups triggered those with the same cause.

The AFL tests are also coverage guided, here the tests revealed that this non grammarbased approach triggered similar exceptions as the test with the generators. The campaign was set up to test and compare grammarbased fuzzing, so there is no baseline to compare the AFL test against a truly blackbox non-guided approach. Therefore, we can not make a statement about the effectiveness of coverage-guided mutation based fuzzing over blackbox fuzzing for *nextflow*.

[6]Relative Standard Deviration

Answer to RQ4. In the campaign we could measure an improvement of valid files generated over time for each generator while using the coverage-guided JQF driver (Zest-Algorithm).

However, we could not measure any coverage improvements in using coverage-guided fuzzing (Zest-Algorithm) over blackbox fuzzing for both generators.

For bugs found, the coverage guided approach and the blackbox approach has both triggered the same exceptions when using the generators, but the coverage-guided tests with AFL have triggered one exception more.

4.2.5 Findings

We take a look at what vulnerabilities have been found. One of them is already mentioned while answering RQ3 (see Section 4.2.3): reserved words in process names are a problem.

On investigating this further, the `ScriptCompilationException` is thrown for process names like `if`, `new`, `native` or `def`. Other reserved words like `super` or `long` will throw a MissingProcessException. But scripts containing process names equal `as`, which is also a reserved word in Groovy [52], are executed successfully.

For process names equal `in` the error message states `no such variable: process`. This is a misleading error message to the user as it was obviously defined. A distinct error message about reserved words or a `ScriptCompilationException` would be a clearer information.

Also, it is noticeable that in every single test run the `AbortRunException` was thrown only one single time. Only in one out of 80 tests a second `AbortRunException` was thrown. This could indicate, that the clean-up method was not comprehensive enough.

While testing several setups some observations where made that influenced the final test set-up presented in the evaluation.

- After one Exception (ScriptCompilation, MissingProperty), *nextflow* will throw an exceptions about the Plugins being already setup when trying to run another script in the same JVM.
- After some unsuccessful script executions *nextflow* will not start a script anymore without any error message. This behaviour was also observed during tests with the described AFL set-up.
- If an error occurred during compiling and executing of a script the user must clean up the current session by hand, to be able to start a new script in the current JVM. The cleanup methods would normally be called in `nextflow.script.ScriptRunner` after the execution of a terminated successfully.
- The combination of Java 11, Groovy, ASM, JaCoCo and *nextflow* seems to be unstable, as the measuring of code coverage was sometime simply not possible running the `AFLDriver` or the`ReproGuidance` on beforehand coverage creating test inputs.

Also, during the beginning of the implementation a bug in JQF 1.9 was found. The coverage calculation would sometimes throw a DivisionByZero-Exception. This did not need to be reported as it already was fixed in the 2.0 version published in May 2023.

4.3 Discussion and Threats to Validity

In this thesis we wanted to evaluate if grammarbased fuzzing can be applied to test workflow engines and what we need to set up to get a working example. To answer this we used a custom generator for creating *nextflow* specific workflow scripts and tested this with a state -of-the-art fuzzing driver JQF and took a look at the difference of random testing and coverage-guided fuzzing. We also did a comparison to the greybox fuzzing tool AFL to investigate what the differences would be.

We could confirm that a grammarbased fuzzing is suitable to be applied at workflow engines, as the coverage measurements match other observations ([9, 8, 15, 14]) on this metric in this research field. Also, we have seen that the proposed implementation can be applied with blackbox and coverage-guided techniques.

The research question regarding the bug finding capabilities within the execution phase of data pipeline could be answered with a yes, as we found one exception that was raised after compilation. The fuzzing campaign also found some irregularities in the handling of syntax errors while compiling scripts, as there is no distinct warning about reserved words. One could argue that this is common behaviour for a compiler, but we can imagine it would be a better user experience if a checking for reserved names was done beforehand. The answer to the question if it is really a bug how reserved words are processed is probably a matter of opinion.

For the sake of performance compromises had to be made regarding the complexity of the generated test files. While a grammar written in BNF is easy to read and understand for a human-being the generator which used the defined grammar to generate test scripts expanded this complexity by taking many iterations and bytes from the `SourceOfRandomness` to produce one test file. However, the used implementation (*GramTest*) provided valid inputs easily at low development cost.

We have observed that we did not make any coverage improvement while testing both presented generator with the coverage-guided Zest-Algorithm. One possible explanation for this behaviour could be that the scripts generated with the BNF grammar are too limited and uniform to explore more of the source code responsible for the execution. It is also possible that the sheer amount of data both generators used to create the test was too much for the Zest-Algorithm to optimize for.

The execution speeds observed (Figure 4) are really slow compared to other fuzzing campaigns run over 10 million tests [9] per hour (what would be about 2777 executions per second). Our test are besides the speed of the use generator this slow because *nextflow* itself is also slow when processing and *running* a script. We can make this assumption as the tests with AFL also show about

2500 executions over one hour, without any test script generation. The small *nextflow* script in Section 2.2.1 takes 300ms on average to be executed when repeatedly started in one JVM and the first execution in the current JVm can even take three seconds.

Also, to set up working test, a cleanup method was needed to make a next execution in the current run possible. It is not certain that this clean up was sufficient in resetting the current state of *nextflow* or if this had another impact on the execution. The only one time in every JVM triggered `AbortRunException` indicates that the clean-up method was not complete. But without adding such a clean-up functionality the fuzz testing would have been meaningless as no execution was made after one error.

One main difference between this thesis and other publications regarding fuzzing in Java that the implementation and testing set-up focused on executing scripts rather than only parsing or compiling them. Different presented algorithms ([9, 10, 29]) compare their coverage maximizing or bug finding capabilities with the benchmark used in the evaluation of the Zest-Algorithm [8]. This included tests for parsing Apache maven models, Java classes or compiling Rhino and Closure files. We tested in this thesis if we can use fuzz testing for an application that will also process a given input. We can confirm, that the JQF fuzzing framework with it intended use of writing generators and using the results of them in Unit-Tests is also suitable for this application scenario.

For further research all used tests, described grammars, log files and data analysis is publicly available.

Internal Validity. To counteract with the non-deterministic behaviour of fuzzing [9] and to prevent systematic errors, the experiments where designed as proposed by Klees et al. [20] and also followed the guidelines provided by Böhme et al. [27]. All experiments ran 20 times for the same duration. The evaluation of the gathered data was done as described in other publications [9, 8, 15, 14] regarding this field. Another threat to internal validity is the duration for the tested campaign since both Klees et al. [20] and Böhme et al. [27] state that longer runs (24 hours) should be used for bug finding evaluations for their reliability. As we were interested in testing whether we can apply grammarbased fuzzing to test workflow engines in execution and not wanted not proof a new fuzzing algorithm, the shorter runs in the fuzzing campaign can be considered valid.

External Validity. The presented approach is highly adapted to fuzz the given target *nextflow* which could be a threat to external validity. We believe that the presented approach could be adapted and applied to other JVM based pipeline tools that use either the CWL or an own DSL. As of September 2023 the assortment of JVM based workflow tools that use scripts for definition and processing is quite small.

[6] `https://github.com/schemmea`

One suitable candidate that would fit to test this approach is another project written in Groovy: *Bpipe*[7]. It uses also an own scripting language to execute different shell scripts. The proposed grammars in Section 3.2 could be altered to fit this language definition and tested against *Bpipe*. But this would have exceeded this thesis as *Bpipes* workflow notation differs from the used one for *nextflow* by a considerable amount. As *nextflow* is a popular workflow engine it seems to be a valid representative to test grammarbased fuzzing on workflow engines in the JVM world. With manageable alterations the proposed approach could be applied to similar applications.

Construct Validity. Another concern for the empirical study is the construct validity. In this thesis we were interested whether we could use grammarbased fuzzing techniques to test workflow engines deeper in the execution phase and after the compiling stage. We did this by guiding the evaluation of the fuzzing campaign to focus on two main points often used in validation of fuzzing techniques: Examining the exceptions logged during the campaign and the analysis of the coverage measured by JQF. We compared our measured results with other publications in this research field. Also, we compared our results with a non grammarbased approach to see where exactly the differences between executed and only compiled scripts for the selected SUT are.

[7] `https://github.com/ssadedin/bpipe`

5 Conclusion

In this thesis we took a look at grammarbased fuzzing by using generators with a black-box and coverage-guided approach. Further we tested how we can apply the technique to a Groovy based data pipeline software, in particular *nextflow*. For this purpose two different generators were written and set-up to run with JQF as the fuzzing driver. One generator that mainly focused on creating executable workflow scripts (semantic generator) and one more lightweight generator that would produce simpler workflow scripts faster but without ensuring that those are really executable. We also implemented tests to run with AFL in the same configuration as comparison to a non grammarbased fuzzing technique. In the evaluation we focused on input files generated over time, what coverage we reached and what bugs the different combinations of generator and fuzzing strategy have found.

The results have shown that grammarbased fuzzing can be used to test *nextflows* workflow execution. And by replaying saved inputs and could also determine which classes are involved. For this comparison we used the prepared inputs from AFL. In our study grammarbased fuzzing was better for testing a workflow engine, as AFL successfully managed to compile scripts and start them, but no workflow was actually executed (Figure 6).

In terms of coverage and valid files created by the two implemented generators we did see an improvement in using the semantic approach. Along the investigation of occurred exceptions and the logfiles we have found some interesting behaviour from *nextflows* compilation strategy, as some errors in the workflow script only were noticed and raised at runtime.

In retrospective, we have leaned two things by using two different grammars in our tests. First, a simpler grammar with a smaller range of characters to build identifiers from will trigger fault inducing behaviour. And second, a wider grammar with the whole alphabet to build identifiers from will trigger other exceptions, in our particular case on reserved words.

nextflow is released as command line tool but the developers published their project as maven dependency, so anyone could use it in a JVM-based application. But it seems that the current implementation of *nextflow* relies heavily on terminating and the JVM being destroyed after every call over the command line.

5.1 Future Work

In the evaluation of the used generators we have observed that a coverage-guided approach with JQFs Zest-Algorithm did not statistically significant improve the coverage measured over time. One explanation could be the restricted and fairly uniform scripts produced by the generators And also concerning the generator, it will consume an astronomical high number of random bytes to generate one script. The used grammar in all tests only represents a small excerpt of the capability of the *nextflow* DSL2. But the evaluation has shown that a slightly complexer grammar reaches more coverage and triggers different exceptions. An interesting question for further research would be, if we can improve the coverage and enable the Zest-Algorithm to maximize this if an extensive grammar was used in a generator that uses less random bytes.

Bibliography

[1] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing”. In: *NDSS*. 2008.

[2] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. “Grammar-Based Whitebox Fuzzing”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 206–215. ISSN: 0362-1340. DOI: 10.1145/1379022.1375607.

[3] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Aug. 2012, pp. 445–458. URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final73.pdf.

[4] Gustavo Grieco, Martin Ceresa, and Pablo Buiras. “QuickFuzz: An Automatic Random Fuzzer for Common File Formats”. In: *Proceedings of the 9th International Symposium on Haskell.* Haskell 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 13–20. DOI: 10.1145/2976002.2976017.

[5] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI ’11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[6] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Superion: Grammar-Aware Greybox Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081. URL: https://arxiv.org/pdf/1812.01197.pdf.

[7] Amir Hossein Kamali, Eleni Giannoulatou, Tsong Yueh Chen, Michael A. Charleston, Alistair L. McEwan, et al. “How to test bioinformatics software?” en. In: *Biophys Rev* 7.3 (Sept. 2015), pp. 343–352. DOI: 10.1007/s12551-015-0177-3. (Visited on 09/21/2023).

[8] 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.* July 2019, pp. 329–340. DOI: 10.1145/3293882.3330576.

[9] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. “CONFETTI: Amplifying Concolic Guidance for Fuzzers”. In: *Proceedings of the 44th International Conference on Software Engineering.* ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 438–450. DOI: 10.1145/3510003.3510628.

[10] 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. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 249–261. DOI: 10.1145/3510003.3510182.

[11] Barton P. Miller, Lars Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[12] Ella Bounimova, Patrice Godefroid, and David Molnar. “Billions and billions of constraints: Whitebox fuzz testing in production”. en. In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 122–131. DOI: 10.1109/ICSE.2013.6606558.

[13] Patrice Godefroid. “Fuzzing: hack, art, and science”. In: *Commun. ACM* 63.2 (Jan. 2020), pp. 70–76. DOI: 10.1145/3363824.

[14] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. “Smart Greybox Fuzzing”. In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1980–1997. DOI: 10.1109/TSE.2019.2941681. URL: https://mboehme.github.io/paper/TSE19.pdf.

[15] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. “MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools”. In: *Software Engineering 2021*. Ed. by Anne Koziolek, Ina Schaefer, and Christoph Seidl. Bonn: Gesellschaft für Informatik e.V., 2021, pp. 81–82. DOI: 10.18420/SE2021_29.

[16] Joan C. Miller and Clifford J. Maloney. “Systematic mistake analysis of digital computer programs”. en. In: *Commun. ACM* 6.2 (Feb. 1963), pp. 58–63. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/366246.366248.

[17] G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The Art of Software Testing*. 2004. ISBN: 9780471469124.

[18] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. Cambridge University Press, 2016. DOI: 10.1017/9781316771273.

[19] Rohan Padhye, Caroline Lemieux, and Koushik Sen. “JQF: coverage-guided property-based testing in Java”. en. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Beijing China: ACM, July 2019, pp. 398–401. DOI: 10.1145/3293882.3339002.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating Fuzz Testing”. en. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804.

[21] Simon P. Sadedin, Bernard Pope, and Alicia Oshlack. “Bpipe: a tool for running and managing bioinformatics pipelines”. In: *Bioinformatics* 28.11 (Apr. 2012), pp. 1525–1526. DOI: 10.1093/bioinformatics/bts167.

[22] Eytan Adar. Ed. by Jon Alhajj Redaand Rokne. New York, NY: Springer New York, 2018, pp. 983–991. DOI: 10.1007/978-1-4939-7131-2_300.

[23] Andrzej Wąsowski and Thorsten Berger. *Building Modeling Languages.* Cham: Springer International Publishing, 2023, pp. 25–46. DOI: 10.1007/978-3-031-23669-3_2.

[24] Azza E. Ahmed, Joshua M. Allen, Tajesvi Bhat, Prakruthi Burra, Christina E. Fliege, et al. “Design considerations for workflow management systems use in production genomics research and the clinic”. en. In: *Sci Rep* 11.1 (Nov. 2021). DOI: 10.1038/s41598-021-99288-8.

[25] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, et al. “Nextflow enables reproducible computational workflows”. en. In: *Nat Biotechnol* 35.4 (Apr. 2017), pp. 316–319. DOI: 10.1038/nbt.3820. (Visited on 09/20/2023).

[26] Nikolas Havrikov. “Grammar-Based Fuzzing Using Input Features”. en. In: (2021).

[27] Marcel Böhme, László Szekeres, and Jonathan Metzman. “On the Reliability of Coverage-Based Fuzzer Benchmarking”. In: *Proceedings of the 44th International Conference on Software Engineering.* ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1621–1633. DOI: 10.1145/3510003.3510230.

[28] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. “Grammarinator: A Grammar-Based Open Source Fuzzer”. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 45–48. DOI: 10.1145/3278186.3278193.

[29] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. “Evolutionary Grammar-Based Fuzzing”. In: *Search-Based Software Engineering.* Ed. by Aldeida Aleti and Annibale Panichella. Cham: Springer International Publishing, 2020, pp. 105–120. DOI: 10.1007/978-3-030-59762-7_8.

Online References

[30] *nextflow*. URL: https://nextflow.io (visited on 09/07/2023).

[31] *american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 09/21/2023).

[32] Thomas Brandstetter. *Greybox Fuzzing detects security vulnerabilities in software*. URL: https://www.mpg.de/20666079/software-security-gap-fuzzing (visited on 09/07/2023).

[33] *Preface, Testing for Continuous Delivery with Visual Studio 2012*. Aug. 2013. URL: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/jj159344(v=pandp.10) (visited on 09/07/2023).

[34] Sten Pittet and Atlassin. *What is Code Coverage?* URL: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage (visited on 09/17/2023).

[35] *Java Development Kit Version 17 API Specification*. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/package-summary.html (visited on 09/07/2023).

[36] *Formal grammar*. Page Version ID: 1173871367. Sept. 2023. URL: https://en.wikipedia.org/w/index.php?title=Formal_grammar&oldid=1173871367 (visited on 09/17/2023).

[37] Dr. Chris Irwin Davis. *Describing Syntax and Semantics*. URL: https://personal.utdallas.edu/~cid021000/CS-4337_13F/slides/CS-4337_03_Chapter3.pdf (visited on 09/17/2023).

[38] *Backus-Naur Form — computer language notation — Britannica*. Aug. 2023. URL: https://www.britannica.com/technology/Backus-Naur-Form (visited on 09/17/2023).

[39] Th. Estier. *About BNF notation*. URL: http://cui.unige.ch/isi/bnf/AboutBNF.html#Naur60 (visited on 09/17/2023).

[40] *CONFETTI*. URL: https://github.com/neu-se/CONFETTI (visited on 09/18/2023).

[41] *JQF Introduction*. 2022. URL: https://github.com/rohanpadhye/jqf (visited on 09/11/2023).

[42] *Java Interoperability - The Scala Programming Language*. URL: https://www.scala-lang.org/old/faq/4 (visited on 09/17/2023).

[43] *AFL user guide*. URL: https://afl-1.readthedocs.io/en/latest/user_guide.html#settings-for-afl-fuzz (visited on 09/17/2023).

[44] *The Fuzzing Book - Greybox Fuzzing*. URL: https://www.fuzzingbook.org/html/GreyboxFuzzer.html (visited on 09/05/2023).

[45] *Common Workflow Language*. URL: https://www.commonwl.org/ (visited on 09/21/2023).

[46] *The story of nextflow*. URL: https://elifesciences.org/labs/d193babe/the-story-of-nextflow-building-a-modern-pipeline-orchestrator (visited on 09/07/2023).

[47] *nf-core pipelines*. URL: https://nf-co.re/pipelines (visited on 09/07/2023).

[48] *Tribble*. URL: https://github.com/havrikov/tribble (visited on 09/17/2023).

[49] *OSS-Fuzz*. en-US. URL: https://google.github.io/oss-fuzz/ (visited on 09/17/2023).

[50] Jesse Ruderman. *Introducing jsfunfuzz*. URL: https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/ (visited on 09/18/2023).

[51] *Fuzzing with AFL*. URL: https://github.com/rohanpadhye/JQF/wiki/Fuzzing-with-AFL (visited on 09/10/2023).

[52] *Reserved Groovy keywords*. URL: https://docs.oracle.com/cloud/latest/big-data-discovery-cloud/BDDDE/rsu_transform_unsupported_features.htm (visited on 09/20/2023).

Appendix

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 September 27, 2023 ....