CONFUZZION: A Java Virtual Machine Fuzzer for Type Confusion Vulnerabilities

William Bonnaventure
SnT, University of Luxembourg
Luxembourg, Luxembourg
william.bonnaventure@gmail.com

Ahmed Khanfir
SnT, University of Luxembourg
Luxembourg, Luxembourg
ahmed.khanfir@uni.lu

Alexandre Bartel*
Department of Computing Science
Umeå University, Sweden
alexandre.bartel@cs.umu.se

Mike Papadakis
SnT, University of Luxembourg
Luxembourg, Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
SnT, University of Luxembourg
Luxembourg, Luxembourg
yves.letraon@uni.lu

***Abstract*—Current Java Virtual Machine (JVM) fuzzers aim at generating syntactically valid Java programs, without targeting any particular use of the standard Java library. While effective, such fuzzers fail to discover specific kinds of bugs or vulnerabilities, such as type confusion, that are related to the standard API usage. To deal with this issue, we introduce a mutation-based feedback-guided black-box JVM fuzzer, called CONFUZZION. CONFUZZION, as the name suggests, targets security-relevant object-oriented flaws with a particular focus on type confusion vulnerabilities. We show that in less than 4 hours, on commodity hardware and without any predefined initialization seed, CONFUZZION automatically generates Java programs that reveal JVM vulnerabilities, i.e., the CVE-2017-3272. We also show that state-of-the-art fuzzers or even traditional automatic testing techniques are not capable of detecting such faults, even after 48 hours of execution in the same environment. To the best of our knowledge, CONFUZZION is the first fuzzer able to detect JVM type confusion vulnerabilities.**

***Index Terms*—Fuzzing, vulnerability, Java Virtual Machine**

I. INTRODUCTION

The Java Virtual Machine (JVM) is a piece of software that runs programs, on different computing architectures, represented as Java bytecode. JVMs are widely used in backend and frontend applications, such as Apache spark, desktop and mobile applications, which unfortunately are at risk if the JVM used is vulnerable. For instance, some JVM versions contain type confusions which are critical vulnerabilities allowing attackers to escape the Java sandbox and execute arbitrary code.

In the past 5 years, at least 5 CVEs, such as CVE-2017-3272, allowed performing a type confusion in Java [1], [2], [3], [4], [5]. Therefore, we focus our research on finding such vulnerabilities inside the JVM. A popular technique to exploit type confusion vulnerabilities in Java is to set the Security Manager to `null`. This allows bypassing the Java sandbox and getting arbitrary code execution on the host machine with the privileges of the JVM process [6].

*At the time this research was conducted Alexandre Bartel was at the University of Luxembourg and the University of Copenhagen.

A type confusion vulnerability permits breaking the object encapsulation and thus allows reads and writes on object fields that may not be permitted otherwise. While this allows to break out of the sandbox, it also enables to hide control flow from static analyzer tools which assume that the principle of object encapsulation is respected in the programs they analyze. Therefore, type confusion vulnerabilities could also be exploited to hide control flows and thus bypass vetting mechanisms such as the ones preventing malicious Android applications from entering the Google Play store [1].

A potential approach to discovering type confusion is manual code analysis and testing. However, this approach is time consuming, error prone and requires that the analyst has deep knowledge of the target program. Another approach is to use an automated testing approach such as fuzzing which helps at generating corner cases. Fuzz testing refers to a technique that pseudo-randomly generates and executes test inputs with the intend to trigger crashes and reveal issues in a program. Fuzzing techniques are frequently used to automatically uncover bugs and potential security vulnerabilities. In recent years, AFL [7] has been the reference fuzzer for C and C++ programs. Though, fuzzers are becoming more popular in all kind of languages: C, C++, and Java [8], [9], [10], [11], [12], [13], [14], [15], [16].

Type confusion is usually caused by missing checks, on the JVM, allowing attacker controlled objects in either the C++ or the Java code of the JVM. Therefore, to uncover such vulnerabilities one needs to check whether the Java code of the JVM has any missing checks. Therefore, fuzzers have to generate Java programs that are syntactically valid and use the standard Java API in such a way that checks all API-related entry points. In other words, the semantics of the programs generated by the fuzzers need to explore the different combinations of the Java API usages. This, of course, defines a very large space for the fuzzer.

Current research on Java fuzzers [14] [17] [15] [16] is

[1]Android apps are written in Java and rely on the Java Class Libraries.

focusing on testing Java programs and not the JVMs. This means that existing fuzzers are mutating or generating test inputs of a Java program that runs on a JVM. For example, if the Java program processes XML documents, the Java fuzzer generates XML documents. As a side effect of testing Java programs, current Java fuzzers only partially and indirectly test a subset of the JVM methods. They are thus highly unlikely to test all possible JVM method calls and situations that may lead to a type confusion.

JVM fuzzers do not generate inputs for Java programs, but for the JVM. That is to say they generate Java programs which are given as input to the JVM. State-of-the-art JVM fuzzers, such as ClassFuzz [18] or ClassMing [19], mainly target the Java parser and the language features but not the Java API. Thus, by design, they lack mutation operators that add JVM method call instructions limiting the exploration of the API usage. Of course, this is happening in order to allow them to focus on the language features and avoid spending time on JVM method combinations. They also rely on user-defined sets of test cases (frequently called seeds), that define the program ingredients to be used when generating tests. Unfortunately, these two choices, by design, limit the ability of these fuzzers to discover type confusion vulnerabilities when starting from scratch.

CONFUZZION creates Java programs through successive mutations starting from a – potentially empty – seed program. At each iteration, a new Java program is tested and executed inside the JVM in order to generate a syntactically and semantically valid complex program that may contain a type confusion. In essence we evolve a Java program by searching for possible valid JVM method call combinations that may lead to a type confusion. This is a fundamental difference from existing approaches and one way to address the problem of revealing type confusion vulnerabilities.

All-in-all our approach aims at testing the semantic usage of APIs within Java programs, with particular focus on the Java API and type confusion bugs. Another important characteristic of our approach is that it can be used to test the JVM evolution. In particular, one can define a list of JVM classes and methods that have been changed/evolved since the last release, which will be used by the fuzzer, i.e., they will be instantiated and tested on different combinations of statements. This allows reducing the space of programs that the fuzzer will target and direct testing towards the most risky method invocations.

Overall, CONFUZZION is the first tool, to our knowledge, that targets the semantic usage of Java APIs wrt type confusion. Since CONFUZZION generates programs, it is conceptually close to compiler testing [20]. Though, our target is the JVM and not the Java compiler. Nevertheless, according to Chen et al. [20] very few compiler testing techniques focus on Java and none of those target type confusion.

In this paper we make the following contributions:

- We design Confuzzion, a novel black-box JVM Fuzzer able to find type confusion vulnerabilities. Confuzzion is a solid open-source code base for future development in the field of Java security research. To the best of our knowledge, Confuzzion is the first Fuzzer of its kind.
- We compare the Fuzzer to state-of-the-art JVM Fuzzers and automated testing tools targeting the JVM.
- We evaluate the Fuzzer and show that it can find real-world vulnerabilities in a reasonable amount of time (hours). CONFUZZION finds, for example, the type confusion known as CVE-2017-3272 [4] with a median time of 4 hours.

II. MOTIVATION

A type confusion is a wrong assumption of an object's type at execution time. This means that the object's type at runtime is incompatible with the object's type at compilation time. In Java, type confusions are critical vulnerabilities since they allow to disable the security manager and thus evade the Java sandbox [6].

A. Type confusion vulnerabilities are a major issue in Java

Type confusion vulnerabilities in Java are prominent. To demonstrate this fact, we examined the extend to which the 100 most recent public Java releases/updates have confusion type vulnerabilities. In particular, we checked the extend to which 5 known confusion vulnerability CVEs, listed in Tables II, affect the Java versions and updates from 1996 to 2020. Java versions range from 1.6 to 14, i.e., version 1.6 update 1 to update 45, version 1.7 update 0 to update 80, version 1.8 update 5 to update 251, version 9 update 0 to update 4, version 10 update 0 to update 1, version 11 update 0 to update 6, version 12, version 13 update 0 to update 2 and version 14 update 0 to update 1. Interestingly, we found that 76% of the above Java releases are affected by at least one of these 5 CVEs. This is alarming as it suggests that the sandbox mechanism can be completely bypassed on at least 76% of the Java releases. Moreover, the actual percentage is probably much higher, as we only considered CVEs which we know for sure are type confusion vulnerabilities.

B. Existing fuzzers are not designed to find type confusion vulnerabilities

State-of-the-art JVM fuzzers such as ClassFuzz [18] or ClassMing [19] aim at testing the Java parser and the generic functionality of the JVM thus they do not explore the conformance of the Java API usage within programs. This is because these fuzzers do not search for combinations of method invocations, i.e., the mutation operators they use do not include any operator that adds invoke instructions. Moreover, existing fuzzers heavily rely on the properties of user defined sets of test cases, called seeds. Unfortunately, this reliance introduces a risk of missing vulnerabilities contained on methods that are not called by the user defined tests/seeds. Even when seeds contain calls to the vulnerable method(s), the fuzzers are still unlikely to recreate the triggering conditions since they do not explore different method call combinations. In other words, seeds are unlikely to have the right triggering

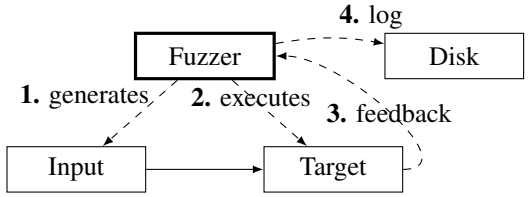

Fig. 1. Generic Design of a Fuzzer

set of method invocations (typically includes multiple different method invocations).

State-of-the-art JVM fuzzers are also limited by the way they operate. In particular, existing fuzzers rely on differential testing. This means that they test the two JVMs and contrast their results. Though, if the two JVMs contain the same vulnerability [2] they will not detect it because the output would be the same. This situation calls for the development of a new approach to detect subtle vulnerabilities in the JVM.

With CONFUZZION, we are able to create Java programs from scratch to call any method of the JVM and create the right conditions to trigger the vulnerability. Furthermore, we add post-conditions in every generated Java program to check if there is a type mismatch. This enables to automatically flag any type confusion vulnerability that is triggered during the execution.

In the next section, we provide the necessary background on fuzzers technologies.

III. BACKGROUND

In this section, we describe the required background related to fuzzers as well as the JVM environment. In Section III-A, we briefly describe what fuzzers are and the different flavors of fuzzers. Then, in Section III-B, we explain the architucture of the Java virtual machine and its environment as well as the difference between Java fuzzers and JVM fuzzers.

A. Fuzzers

The general goal of a fuzzer is to generate test inputs, based on random changes, to identify corner cases on a target program. Figure 1 describes a typical feedback-guided fuzzer workflow. The fuzzer first generates an input (step 1). The fuzzer then executes the target program with the generated input (step 2). It then takes feedback from the execution of the target program into account (step 3) to generate a new input (back to step1). If a bug is detected, the corresponding input and all the necessary information to reproduce the bug are logged (step 4). The fuzzer continuously runs until, for instance, a certain amount of time has elapsed.

1) Generator-based fuzzers: Generator-based fuzzing ensures the generation of inputs that verify specific structures and rules to restrict the fuzzing effort to a targeted property or level of the system under test. This further maps to Grammar-based fuzzing if the generator is following a given grammar. Similarly, model-based fuzzing refers to a generator following a model of specifications. In every iteration, the generator constructs inputs by taking a sequence of selection decisions.

A mutation-based fuzzer starts with a potentially empty seed. At each iteration, the fuzzer applies one or more mutations to generate the next input. Frequently used mutation operators include bit-flip and adding/removing bytes. The fuzzer either finds a new interesting input that crashes the program or uses the new input as the next seed.

In the case of generator-based fuzzers, the mutation is applied on its sequence of decision. So instead of mutating the final generated input we mutate the sequence inferring it according to the guidance and its sensors (coverage, exceptions, etc.). In the end, the seed controls the random behaviour of the generator but not directly the end result that is created according to the generator rules.

CONFUZZION is a mutation-based fuzzer since it keeps the same inputs and continuously change them using mutation operators.

2) White, Grey and Black-box fuzzers: The color code represents the visibility of the target program's code from the fuzzer's perspective. A white-box fuzzer has complete knowledge of the source code of the target program. A grey-box fuzzer has partial knowledge of the source code. Finally, a black-box fuzzer has no knowledge of the source code. The intermediate 'grey-box fuzzer' terminology is often used when only the compiled binary of the target program is available. In this case the fuzzer has no direct knowledge of the source code. This type of fuzzing allows to instrument the binary target program to have, for instance, code coverage information on closed-source programs. CONFUZZION is a black-box fuzzer since it has no knowledge of the source code of the target JVM.

3) Feedback: After execution of the target program on a generated input, the fuzzer has access to feedback information. This could, for instance, be code coverage (has the new test covered more code than the previous test?) or information about the execution state (has the target program crashed? which function has crashed?). This information is used by the fuzzer to, for instance, rank inputs and/or generate new inputs. If a fuzzer relies on the code coverage it implies that this is a white-box fuzzer. CONFUZZION relies on method crash feedback to generate new inputs.

4) Smart: A fuzzer can be considered as *smart* [22] if it knows the input structure expected by the program beforehand. Sometimes smart-fuzzing is used to generate inputs that are complex as they need to satisfy a set of syntactic and/or semantic rules prior to be submitted to the target program. In fact, classic 'non-smart' fuzzing is really appropriate on binary files but when the structure becomes more restrictive like a grammar-based structure then the fuzzer may take a long time just to generate a basic valid input. That is why smart-fuzzing is useful to generate mostly valid inputs which should be accepted by the program but that can trigger errors as they are covering corner cases. CONFUZZION is a smart

[2]Vulnerabilities usually have a long life-cycle and typically affect multiple releases [21]

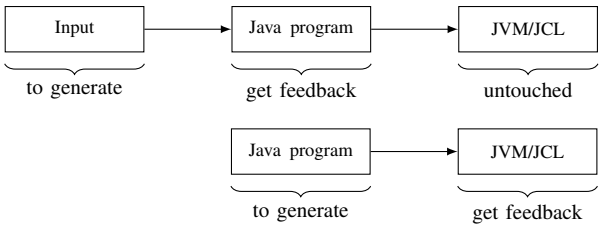

Fig. 2. Java Program Fuzzer (top) and JVM Fuzzer (bottom)

fuzzer since it knows the structure of Java class files and generates valid class files.

B. Java and the JVM

1) The Java Environment: A Java program consists of one or more `.class` files. The entry point of the program is the `main` method. A Java program runs on a Java virtual machine (JVM) which is written in C++. A Java program and the JVM rely on a set of basic Java classes called Java Class Library (JCL). The JCL is shipped with every JVM. The JCL contains classes such as `java.lang.Class`, `java.lang.String` and `java.io.File`: it provides a basic layer to access the filesystem, the network, cryptographic primitives, etc. Together, they form the Java Runtime Environment (JRE). The JVM loads class files through a *class loader*. The class loader checks the structure of the class files as well as syntactic and semantic properties of their code.

2) Java Program Fuzzers: A Java fuzzer aims at testing the implementation of a Java program. As illustrated in Figure 2 (top), a Java fuzzer generates input to a Java program, executes the target program with the generated input and analyzes feedback information from the execution of the Java program. Kelinci [15] and JAVA-AFL [16] are examples of Java fuzzers. They instrument the bytecode of the target program to extract coverage information.

3) JVM Fuzzers: A JVM fuzzer aims at testing the implementation of a target JVM. As illustrated in Figure 2 (bottom), a JVM fuzzer does not generate input for a Java program. Instead, it generates input to a JVM. This input is actually a Java program. The JVM fuzzer then executes the generated Java program on the JVM and analyzes feedback information from the execution of the JVM running the Java program. ClassFuzz [18] and ClassMing [19] are examples of JVM fuzzers. Moreover, they are differential fuzzers for the JVM. This means that a given generated input is executed on a reference JVM and on the target JVM. If the feedback of the two executions differs, an inconsistency is detected. CONFUZZION is also a JVM fuzzer.

IV. THE CONFUZZION APPROACH

CONFUZZION is a mutation-based, feedback-guided black-box smart fuzzer. Its overall architecture is illustrated in Figure 3.

It operates by evolving a single program, hereafter referred to as the master program. Initially, the master program is an empty one or the user defined seed program. The process

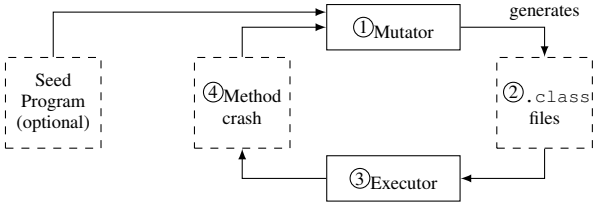

Fig. 3. Confuzzion overview. The fuzzer – with an optional seed – relies on mutation operators (1) to generate an input program (2). The execution of the input (3) sends feedback information about method crashes (4) back to the fuzzer which use it when generating new inputs.

is then entering a loop where, at each iteration, additive mutations, i.e., mutations that add instructions, are applied to the master program. In order to evolve valid programs, at each step of the process, the mutations that result in invalid programs are discarded, this is done through compilation. We also discard mutations that do not conform to some additional checks; we check some basic semantic properties of the mutations, such as the number of arguments given as parameter to a method, and their runtime behaviour, i.e., returning errors and type confusions appearing during execution at the target JVM. To increase diversity, escape from stacking points, i.e., programs that hang on almost all mutations, and avoid bloating [23] we set checkpoints (a predefined threshold number of iterations) at which we backtrack to a previous state. This is happening by selecting a random number of the latest applied mutations, which are reverted. This means that whenever we reach this threshold number of mutations, we backtrack to the master program that appeared before applying this randomly chosen number of mutations and continue the evolution cycle from there.

A. Evolving the master program

1) Generating programs: To reveal type confusions inside the JVM we need to generate programs that are syntactically correct and run properly in the JVM. This is challenging because of the following two reasons: a) the JVM expects highly structured inputs and b) fuzzing requires non-trivial, semantically diverse class files.

To bypass these problems, we generate programs from Jimple [24], which is an intermediate language between Java and Class files. We use Jimple because it is simple, compared to bytecode, and works well together with Soot [25], a popular code analysis framework. Soot provides semantic information about the programs we evolve and thus, it provides vital feedback that helps evolving our test programs.

Another advantage of using Jimple is that it allows generating programs that are semi-valid, i.e., they can run in the JVM but they involve "unusual" constructs and operations, which may be hard to generate using the Java compiler. This is because the Jimple to Class compiler is generally more permissive and generates different bytecode structures than the Java compiler. Interestingly, generating semi-valid programs allows testing the JVM outside the comfort zone shaped by the Java compiler, i.e., Java compiler restricts JVM inputs to only those Class files that can be produced by the compiler.

TABLE I
SUMMARY OF CONFUZZION'S MUTATION OPERATORS.

Category	Mutation operator	Description
Program	AddClassMutation	generates a new Java class.
Class	AddFieldMutation	adds a new field to an existing class.
	AddMethodMutation	adds a new method to an existing class.
Method	AddLocalMutation	adds a new local variable and generates a corresponding object.
	AssignMutation	assigns an existing value to a new variable.
	CallMethodMutation	adds a method call and generates the necessary arguments.

We argue that this can be particularly important because JVM inputs that are not produced by the Java compiler might target part of the JVM which are naturally less tested and can reveal corner cases that attackers may exploit.

2) Mutations and objects: A `Mutation` is defined as an *addition* of code inside the master program. Table I enumerates the list of all mutations used in CONFUZZION. For example `AddLocalMutation`, a `MethodMutation` that occurs at the Method level, will add a new local variable with a random type and assign a value. If it is a reference type then it will call the class constructor to build an object and assign its reference to the local variable. With those types of mutations we can build objects and reuse local variables in subsequent mutations. Thus, it will build a complex ordered list of statements inside method bodies.

There are two other types of mutations : Class-level mutations and Program-level mutations. For example, `AddFieldMutation` adds a new field inside a class and this impacts a whole class and not only a method. `AddClassMutation` adds a new class to the Program. We define a Program as a list of Classes. This is mainly because some of the type confusions are triggered by the use of new classes that inherit from JCL classes. Thus, to be exhaustive, we need to support Program and Class levels of mutations.

B. Guiding the Search

1) Restricting the Search Space: Proof-of-concepts [3] have shown that type confusion in Java is caused by the use of specific packages of the Java Class Library and by some invalid semantic checks of the JVM. This means that in order to reveal such vulnerabilities we need to use particular classes that will be instantiated and tested on different combinations of statements.

CONFUZZION takes advantage of the above observation and restricts the search in classes that are most important, as specified by the user. This provides decisive advantages in our approach as it allows to thoroughly test specific API classes and to drastically restrict the search space to the points of interests. For instance, one can focus on the classes that were changed in the last release in order to ensure that no new vulnerabilities have been introduced.

2) Handling test bloat: Since all mutations we use are additive, the master program will grow and the fuzzer speed will decrease. This is mainly due to the time needed to compile and execute the program that increases linearly with the size of the program. Therefore, we designed CONFUZZION to be able to revert mutations. As we already discussed, we revert a random number of mutations each time the number of mutations hits a predefined threshold number of mutations. This rewind in time allows us to take a different path of mutations with a different behaviour while keeping a portion of the knowledge acquired with the remaining mutations. That is thanks to the stochastic nature of the process, when reverting a different path in the evolution tree of mutations.

3) Test Feedback and Search Guidance : In every iteration, CONFUZZION takes a sequence of decisions to generate a new input. First, it randomly selects one mutation operator then, it selects the location inside the master program where to apply it.

To better guide the search, CONFUZZION leverages the information gained by the previous executions. In particular, from every execution, CONFUZZION collects the list of called methods and records the frequency that a method was responsible in causing a crash. Then, whenever the fuzzer decides to mutate one of the seed's method-body, it will exclude the ones not being executed and will select one among that list. Once a method-body is selected, the fuzzer may decide to extend it by adding another instruction. Here, the choice of which method to call/invoke depends on its frequency of causing the program to crash, in the previous iterations.

Once CONFUZZION decides to mutate the program with an instruction addition, it selects a method to call from the target classes defined by the user or any other method generated by previous mutations. It favors the selection of complex methods over simple ones, based on the assumption that methods that lead more often the program to crash are more complex, thus more interesting to fuzz, than the ones that always execute successfully. In other words, a method that requires a specific combination of complex arguments is more appealing to re-execute with different parameters than `toString()`-like methods, which are not very meaningful for objects.

For this purpose, we define a crash density d_i of a method i as a value between 0.0 and 1.0 representing the number of crashes caused by the method over the number of times it was executed. In the initial state, all the densities are equal to 1. We also define D, the sum of all densities:

$$D = \sum_{i=1}^{n} d_i$$

We define p_i the probability that a method will cause a crash or eventually reveal a vulnerability, thus the probability to be selected for invocation by CONFUZZION. As our fuzzing iterations describe a series of Bernoulli trials, we expect the results to follow a binomial distribution. The computed p_i

[3] a proof-of-concept, or PoC, is a code that exploits a vulnerability.

```
1       if r2 == null goto label1;
2       z0 = r2 instanceof java.lang.String;
3       if z0 != 0 goto label1;
4       r3 = new ContractCheckException;
5       specialinvoke r3.<ContractCheckException: void
        ↪  <init>()>();
6       throw r3;
7   label1:
8       nop;
```

Listing 1: Jimple code to detect a type confusion on a local variable r2.

reflects the complexity of the method i while remaining always above 0:

$$p_i = \frac{UCB(d_i, 95\%)}{D}$$

This way, our approach respects the following criteria: (c1) A method with observed crash density d should be chosen twice as often as a method with a density of $d/2$. (c2) If the crash density of two methods is equal, they should be chosen with equal probability. (c3) Methods with no observed crashes should be kept with a probability more than 0 in order to not kill the possibility of choosing them. (c4) Between two methods with 0 observed crashes, the one with fewer trials should be preferred over the other. We know less about the one with fewer trials and thus may find some new crashes.

The above definitions are borrowed from a technical report of Householder et al. [26]. Unlike the technical report, we do not aim at prioritising seed files but methods. We however, do not use an additional Poisson Distribution approximation because our crashing frequency per method is not as rare as in their case.

C. Vulnerability Detection Checks

To detect a type confusion we add Jimple code at the end of the current modified method by the mutation (if any) and then let the program run. We define a specific Contract [27] for Type Confusion. For each Contract we have a checker that can be added to the code. In order to differentiate between all other types of exceptions, we throw a special type of exception `ContractCheckException` that declares a true contract violation inside this program. This notion of Contract is helpful in the case we want to check other types of properties on the generated code.

Listing 1 is a Jimple implementation of a contract responsible to check that a local variable `r2` is of type `String` as expected. Such contract is added to check that the runtime types of the variables correspond to the build time ones.

D. Implementation

CONFUZZION's Java implementation represents 30 classes and 3000 lines of code. It relies on Soot 3.3.0, slf4j-simple 1.7.26, Apache Commons CLI 1.4 and Math3 3.6.1.

One major technical contribution of the proposed approach is its multi-threading capability, which ensures a relatively high fuzzing speed in terms of number of iterations per second. In fact, instead of creating a new JVM instance for every iteration, CONFUZZION operates with only one by running the generated programs in different threads.

The principal configuration possibilities of our current implementation are:

- `Binomial or Uniform`: defines the algorithm of methods selection as a binomial or uniform distribution on the crashes observed. (see IV-B3)
- `Seed` (optional): defines a starting seed program to mutate.
- `TargetClasses`: defines the JVM classes which the fuzzer will use when generating the input programs. (see IV-B1)

These parameters may impact the performance of the fuzzer, which we will discuss further in the sections V and VI.

CONFUZZION [4] source code will be published as an open-source project that can be used by the community to discover future vulnerabilities and to extend its capability.

V. EVALUATION

In this section, we evaluate our approach efficiency in finding historical confusion types in the corresponding vulnerable JVMs. We explain the best tradeoffs to detect such flaws and compare CONFUZZION with some state-of-the-art automated testing techniques.

To examine the performance of CONFUZZION we design a case study aiming at answering the following research questions:

- RQ1: How efficient is CONFUZZION at detecting Type Confusion Vulnerabilities?
- RQ2: How efficient is the Binomial Method Selection used by CONFUZZION?
- RQ3: What is the impact of small changes in the set of targeted classes by CONFUZZION?
- RQ4: How does CONFUZZION compare with state-of-the-art fuzzers?
- RQ5: How does CONFUZZION compare with Randoop in terms of type confusion revealing?

RQ1 aims at investigating the ability of CONFUZZION to detect known type confusion vulnerabilities. We simulate a scenario where developers test the JVM versions using CONFUZZION just before their release. In our analysis, we also consider the time needed to detect these vulnerabilities and highlight the importance of the used seed in shortening this time. RQ2 aims at showing that the efficiency of CONFUZZION's method selection compared to an alternative uniform selection. RQ3 shows the impact of the selection of the JVM classes to target by CONFUZZION wrt vulnerability detection efficiency. RQ4 aims at showing the strengths of our approach wrt the current state-of-the-art on JVM fuzzing and assesses CONFUZZION's speed in terms of iterations per second, while RQ5 wrt test generation tools for Java. This last comparison aims at checking whether traditional test generation techniques could work in the particular scenario we investigate.

[4]reviewers can anonymously access the tool here: https://www.dropbox.com/s/868pc27vqblzpuy/Confuzzion.zip?dl=1. The source code of CONFUZZION will be made available once the paper is accepted.

TABLE II
TARGETED CLASSES BY CVE (CIC)

CVE-2014-0456	CVE-2015-4843	CVE-2016-3587	CVE-2017-3272	CVE-2018-2826
java.lang.System	java.nio.DirectByteBuffer$1	java.lang.invoke.MethodType	java.util.concurrent.atomic.AtomicLongFieldUpdater	java.lang.invoke.MethodType
	java.nio.DirectIntBuffer{RS,RU,S}	java.lang.invoke.MethodHandle	java.util.concurrent.atomic.AtomicIntegerFieldUpdater	java.lang.invoke.MethodHandles
	java.nio.Direct{Char,Double,Float,Long,Short}Buffer{RS,RU,S,U}	java.lang.invoke.MethodHandles	java.util.concurrent.atomic.AtomicReferenceFieldUpdater	java.lang.invoke.MethodHandleImpl
		java.lang.invoke.MethodHandles.Lookup		java.lang.invoke.MethodHandle

Experimental setup: All experiments in this section are run on a server with an Intel E5-2430@2.20GHz processor featuring 24 cores, 100 Gb of RAM and running Debian.

As we expect our approach to be continuously used, in a real world JVM developing workflow, after every code change (commit), we narrow down the fuzzing search-space by reducing the code-under-test (CUT) to the changed classes instead of exhaustively fuzzing all the classes and methods of the JVM. Thus, for every targeted CVE in our data-set, we collected the list of classes impacted by this latter's introducing commit. These classes are listed in Table II.

A. RQ1: How efficient is CONFUZZION at detecting Type Confusion Vulnerabilities?

To illustrate how CONFUZZION can detect type confusion vulnerabilities, we perform two experiments to find CVE-2016-3587 [3] and CVE-2017-3272 [4]. Despite being generic, the current implementation of CONFUZZION has some limitations preventing it to be evaluated on the other three CVEs (CVE-2014-0456 [1], CVE-2015-4843 [2] and CVE-2018-2826 [5]). This is further detailed in section VI.

For each CVE, We first launch the fuzzer using as seed the corresponding complete public Proof-of-Concept (PoC) on a vulnerable JVM to make sure the type confusion is immediately detected. Then, we launch the fuzzer on the PoC without the last line to make sure that CONFUZZION can generate a program triggering the type confusion vulnerability through mutations. Finally, we launch the fuzzer on an empty seed, to observe the time that CONFUZZION takes to trigger the vulnerability without PoC. POC, POC-1 and POC-* are the corresponding Proof-of-Concept classes when removing nothing, one line, or all lines. POC-* is equivalent to an empty seed that corresponds to an empty program. We launch 10 fuzzer instances in parallel and select the binomial method selection for mutations.

CVE-2016-3587 makes the JVM unstable. In fact, executing its revealing program (POC) on it, leads the JVM to crash. Thus, the CONFUZZION post-execution-check is not executed. In a real case scenario of JVM fuzzing, inducing a crash is a desired behaviour and the capital objective, because it reveals a bug or a vulnerability. In this experiment, we are only interested in finding a specific vulnerability. For this reason we check manually the crash stack-trace, to identify if this targeted CVE was revealed or not. We give an example of a program generated by CONFUZZION that reveals CVE-2016-3587 and explain how to go from the crash to exploit, in our anonymous reproduction package [5].

The obtained results are presented in Table III. We first observe that all fuzzers have immediately identified a type confusion in the complete PoC. We observe also that all fuzzers require less than 20 seconds to find the last line of POC-1 for CVE-2017-3272 and around 11 minutes for CVE-2016-3587. Finally, we observe that – with an empty seed – all fuzzers can generate a program to expose the type confusion vulnerability in less than 30 hours and with a median time of less than 4 hours for CVE-2017-3272, but requires longer to find CVE-2016-3587, so at least 365 hours.

[5]Appendix A file in: https://www.dropbox.com/s/868pc27vqblzpuy/Confuzzion.zip?dl=1

TABLE III
TIME IN SECONDS TO FIND CVE-2017-3272 AND CVE-2016-3587 WITH DIFFERENT SEEDS AND A BINOMIAL METHOD SELECTION

CVE-2016				CVE-2017			
Run	POC	POC-1	POC-*	Run	POC	POC-1	POC-*
1	0	276	365h	1	0	8	11334 (3.1h)
2	0	1068	-	2	0	8	104063 (28.9h)
3	0	2818	-	3	0	5	75530 (21h)
4	0	836	-	4	0	11	13226 (3.7h)
5	0	329	-	5	0	14	21012 (5.8h)
6	0	557	-	6	0	9	14101 (3.9h)
7	0	84	-	7	0	16	6558 (1.8h)
8	0	1212	-	8	0	18	3951 (1h)
9	0	310	-	9	0	2	7097 (2h)
10	0	1100	-	10	0	20	17883 (5h)
Median	0	696.5	-	Median	0	10	13663.5 (3.8h)

We extended this experiment for CVE-2016-3587 by launching 10 instances with each of these seeds: POC-2, POC-3, ..., POC-6. For each seed we stop when one of its 10 running instances crashes the JVM. The obtained results are presented in Table IV .

TABLE IV
TIME TO CRASH THE JVM WHEN TARGETING CVE-2016-3587

Seed	POC-2	POC-3	POC-4	POC-5	POC-6
Time	2m (0.033h)	2,16m (0.036h)	114h	6h	87h

We observe that the fuzzer is able to crash the JVM for POC, POC-1, POC-2 and POC-3 in minutes. For POC-4 onwards, the fuzzer needs hours to crash the JVM. By looking at the code of the PoC, we notice that POC-4 removes the line INV responsible for invoking the `invokeBasic` method through a `MethodHandle` with a target method as parameter. This code is quite complex to generate for the fuzzer.

So in addition to the random nature of fuzzing, the time to discovery varies not only by the number of removed lines but also by the generation complexity of these ones. This highlights the advantage of using a strong seed when running the tool. For instance, the practitioners could create a seed program calling the targeted classes, to eventually fasten the fuzzing campaign.

Furthermore, the time to discovery depends a lot on the employed computing resources and can be significantly decreased by running multiple instances of CONFUZZION in parallel. For instance, starting with an empty seed PoC-*, CVE-2017-3272 is found in 1 hour by the 8th run, which is a significantly shorter time-budget compared to the almost

29 hours spent by the 2nd run (instance 8 and 2 in Table III).

B. RQ2: How efficient is the Binomial Method Selection?

TABLE V
TIME IN SECONDS TO FIND CVE-2017-3272 WITH A BINOMIAL METHOD SELECTION AND A UNIFORM METHOD SELECTION.

Fuzzer instance	POC-* (binomial)	POC-* (normal)
1	11334 (3.1h)	24912 (6.9 h)
2	104063 (28.9h)	131469 (36.5 h)
3	75530 (21h)	108887 (30.2 h)
4	13226 (3.7h)	141210 (39.2 h)
5	21012 (5.8h)	-
6	14101 (3.9h)	-
7	6558 (1.8h)	-
8	3951 (1h)	-
9	7097 (2h)	-
10	17883 (5h)	-
Median	13663.5 (3.8 h)	120173 (33.4 h)

To understand if there is a difference in the time to find a vulnerability between the binomial method selection described in Section IV-B3 and the uniform method selection, we re-run 10 instances of the fuzzer to generate a PoC for CVE-2017-3272 with this latter selection method. Results are presented in Table V.

We observe that there is a significant reduction of the median time needed to find the type confusion with our custom binomial method selection (column 2) over the classic uniform method selection (column 3). There is also a more consistent probability of success with the binomial one. Indeed, many runs lead to effectively finding the type confusion with a median of four hours, whereas with uniform selection six runs did not find it in less than 48 hours.

C. RQ3: What is the impact of small changes in the set of targeted classes by CONFUZZION?

To answer this question, we compare the time needed for CONFUZZION to find CVE-2017-3272, with different given target classes. We run the same experiment as in RQ1 for POC-1, with an extended set of classes to fuzz, selected randomly from the JVM library. These sets of classes are compound of all the classes impacted by the commit introducing the CVE listed in Table II (not just the classes introducing the vulnerability), plus a certain number X of random classes: 1, 2, 5, 10 and so on until 50, with a step of 5 classes. We note CIC the classes impacted by the commit introducing the CVE and CIC+X the set containing these classes plus X number of extra randomly selected JVM classes. To reduce arbitrariness due to the stochastic nature of CONFUZZION, we select 10 samples of each CIC+X and run the experiment 10 times for every sample, so a total of 1200 runs. We present, in Figure 4, the box-plots of the time in seconds spent by CONFUZZION to find the CVE for every experiment.

Interestingly, we observe that the fuzzer could in all cases find the CVE. The results also show that the more classes we target, the bigger the space to search, hence the slower the detection. Nevertheless, we can see that up-to 15 additional classes, the detection time remains relatively low, compared to bigger sets. The time-budget increases considerably, from an average of 143s for CIC+15 to 433s for CIC+20, then remains relatively unchanged or grows slowly with the following additions of classes. These results confirm the importance of reducing the search space of such fuzzing campaigns and that CONFUZZION is well suited for a continuous use, such as every 1 or more commits.

D. RQ4: How does CONFUZZION compare with state-of-the-art fuzzers?

1) Efficiency to detect Type Confusion vulnerabilities:: Recently Chen et al. proposed fuzzing tools to detect JVM anomalies, Classfuzz [18] and Classming [19]. Both tools are based on a differential testing approach that compares a JVM to test with a reference one which is considered stable (another JVM or a different version of the same JVM). While Classfuzz fuzzing effort is focused on finding errors in the JVM's startup process, Classming is designed to target deeper ones, that can only be detected by a valid Java program. As far as we know, Classfuzz compares only the files loading and not the execution of the Java programs. Therefore, Classming is the only JVM fuzzer able to find complex corner-case faults such as type confusions.

Classming error finding efficiency, same as Classfuzz, depends on the initialization Java program (seed) which is the origin of all the testing inputs (Java programs generated by mutation) generated by the fuzzer. Considering also the fact that its set of mutation operators limits the input generation to Java programs having only the same set of methods invoked in the initialisation program, its chances are limited to detect type confusion vulnerabilities since these would probably require creating/invoking new classes and making several method invocations. In fact, Classming mutates the given seed by inserting or removing a Jimple instruction and its required labels from the following 5 ones, called Hooking Instructions (HI): (1) goto, (2) return, (3) throw, (4) lookupswitch, and (5) tableswitch. So, even in a perfect case scenario, with a seed that is very close to a type confusion revealing program, which contains a call to a method exposing the vulnerability, such tool would fail to automatically infer a type confusion revealing program, at all or at least in a reasonable amount of time.

Furthermore, our study of the type confusions exposed by JVMs in the 5 last years showed that the same type confusion persists from a version to another through the JDK updates. For instance, the vulnerability CVE-2014-0456 [1] was introduced in the Java version 1.6.0-14, persisted in the following 44 publicly released versions of the JDK, to be finally discovered and fixed in the version 1.7.0-55. Both ClassFuzz and ClassMing's algorithms detect faults by differential testing operated under the strong assumption that the chosen JVM of reference is stable. Therefore, they would only discover such a persistently hidden flaw if they test a vulnerable JVM version (i.e. 1.7.0-51) against a JVM version prior to the vulnerability introduction one, the version 1.6.0-14. While in a real case scenario, before discovering

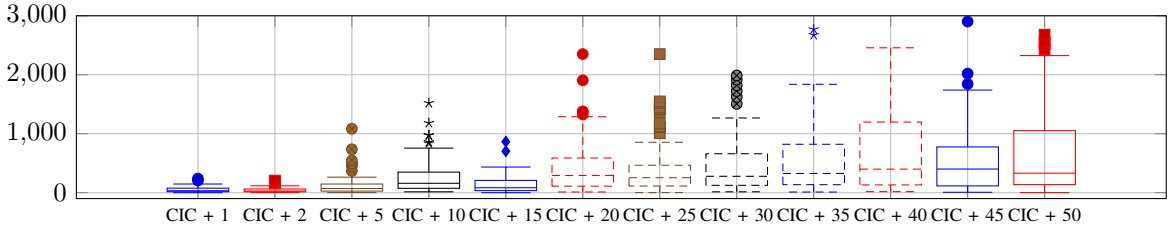

Fig. 4. Impact of small changes in the size of the target classes set on the time (in s) to find CVE-2017-3272 from POC-1

a vulnerability, it is unknown which JVM is vulnerable and which one can be used as reference to reveal it. Thus, it is unlikely for a differential testing approach to discover such flaw, unless exhaustively executing each generated program with all previous JVM versions one by one, as the reference JVM which would considerably slow down the approach. Our approach does not make such an assumption since it bases its detection on confusion type post-check conditions.

We excluded Java *application* fuzzers from this experiment because of the following two reasons. First, they are designed to test Java programs instead of generating Java programs to test JVMs. Second, they crash, hang or fail to execute normally when they target the JVM classes.

2) Fuzzing speed:: To answer this research question, we show that the speed of CONFUZZION is relatively similar to the state of the art java application fuzzers: Java-AFL [16], Kelinci [15] and JQF. Being not able to fuzz a JVM, we run these reference fuzzers against a Java program that reads a given file as an image with the `javax.imageio.ImageIO` class. In every iteration, the fuzzers execute the program with an input file, aiming to reveal an unexpected behaviour.

In Table VI, we illustrate the speed of the above mentioned fuzzers as well as the speed of CONFUZZION under two different configurations of its multi-threading feature: `disabled` and `enabled` with 2 threads, which we note correspondingly as Conf-JVM and Conf-2thr.

TABLE VI
COMPARISON OF CONFUZZION'S JVM-FUZZING SPEED (ITERATIONS PER SECOND) WITH JAVA FUZZERS SPEED IN FUZZING AN IMAGE LOADING PROGRAM.

Fuzzer	Conf-JVM	Conf-2thr.	Java-AFL	Kelinci	JQF
Speed	10	100	10	10	1000

We observe that JQF is by far the fastest among all compared tools with a speed achieving 1000 iterations per second. This is mainly due to the fact that the CUT and the test to fuzz remain unchanged during the fuzzing campaign. Thus, it loads and instruments the java program on startup, then executes the same JUnit test with different inputs in every iteration. Although the JVM-fuzzing task is more resources-consuming (code generation, compilation and execution) compared to the image loading fuzzing one, CONFUZZION-JVM has a similar speed as conventional fuzzers such as Java-AFL and Kelinci. Enabling the multi-threading feature makes CONFUZZION-2thr. 10 times faster than those fuzzers.

E. RQ5: How does CONFUZZION *compare with Randoop in terms of type confusion revealing?*

Research in automated test generation lead to the creation of a solid state-of-the-art collection of generators, such as Randoop [28], Evosuite [29], JTExpert [30] and T3 [31]. These tools generate executable sequences of instructions followed by optional validation assertions targeting specific methods or classes to test. Operating in a white-box manner, they are able to capture the behaviour of the internal methods of the given code under test (CUT), and not just of the software as a whole. Thus, we think that such tools might be able to discover confusion types by generating a sequence of instructions that contains the right order and combination of JVM core method calls with the adequate parameter values.

Randoop [28] and Evosuite [29] are commonly used in research as the continuously maintained mature state-of-the-art automated test generators [32] [33]. Unlike Randoop, Evosuite operates only under the assumption that the given CUT is stable (does not present any fault) thus creates only valid tests that are useful for regression and does not output any error revealing tests. Thus, as we aim at comparing CONFUZZION against automated test generators in terms of type confusion finding, we only select Randoop as our baseline to answer this research question.

Randoop is a feedback guided generator operating with an Incremental Random Search algorithm to construct valid sequences that can cover the most of the code under test. In every iteration, a sequence of instructions is generated then executed, in order to be classified as either an error-revealing or regression one to output, or as redundant or invalid to discard. Only regression sequences are kept as valid sources of arguments for the following generation iterations. Once the given time-budget is consumed, the test generator outputs the regression and error-revealing sequences in well formatted executable JUnit [34] test suites, one for each type. Confusion Type flaws are not detectable by default in the post-check execution of Randoop, thus sequences revealing confusion types execute normally on a vulnerable JVM and are classified as valid regression ones.

To enable Randoop to detect the targeted JVM flaws, we added a dedicated post-checking predicate that compares the result type of every instruction with the expected one. Because of Randoop's post-check customisation limitation, we had to modify its source-code responsible for checking the generated sequences execution, in order to add an extra predicate that prints a message in the terminal, every time a confusion type

is revealed. Our changes have been applied on the current master branch [6], then tested and validated with a manually crafted sequence that reveals a type confusion.

Randoop is not able to find CVE-2014-0456 as it fails to run on JVMs of version 7 and prior. We execute it for 24 and 48 hours on the 4 left CVEs corresponding vulnerable JVM versions 10 times with different seeds in parallel on the same environment as the experiments driven in V-A, V-B and V-C, with different configuration parameters of: time-budget, classes to generate tests for and enabling/disabling multi-threading. In all cases, Randoop fails either to find any confusion type or to execute properly. To exclude the possibility that our confusion type post-check is responsible for these failures, we execute these configurations with the original current released version of Randoop, 4.2.4.

For a fair comparison with CONFUZZION, we started by passing only the CIC classes as code-under-test (CUT), same as in V-A and in V-B. Because of Randoop's disability to generate tests for abstract classes, it quits directly the test-generation runs for CVE-2017-3272. This is not the case for the other CVEs, where only few or none of the changed classes are abstract. We decide then to pass all the JCL classes as CUT. This made Randoop crash or fail because of memory issues.

Another attempt to overcome the limitations of instrumentation and abstract classes testing was to enlarge the input CUT to the classes exposing the vulnerability and their dependent classes (collected using jdeps [35]). This lead Randoop to crash with similar issues, as when passing the whole JVM as CUT.

The multi-threading feature caused Randoop to fail before the expiration of the allocated budget-time, so we decided to disable it.

With the only working configuration, passing only the CIC as CUT and disabling the multi-threading, Randoop run successfully until time-budget expiration, on the vulnerable JVMs 1.8.0u25, 1.8.0u92 and 9. However, it did not find any type confusion.

To guide Randoop even further towards such a flaw discovery, in addition to limiting the targeted CUT to the CIC, we added to its initialization String variables the value "invokeBasic" so that it could use it directly as a parameter in its method calls and eventually create a sequence similar to the PoC. Even in such a perfect case scenario and after 48h of execution, Randoop failed to reveal any type confusion.

VI. DISCUSSION

Unfortunately, CONFUZZION has limitations which prevents it from detecting all type confusion vulnerabilities. These limitations are not only present in CONFUZZION but are, for the most of them, well known limitations of testing tools targeting Java applications or the JVM.

Handling Exceptions. As far as we know, none of the current fuzzers for the JVM generate try/catch blocks. CONFUZZION has some basics mutations such as calling a method or adding a new field, but does not currently support try/catch blocks either. Efficiently handling try/catch block is a challenging open problem. Indeed, for a given Java program with n instructions, there are $\frac{n(n+1)}{2}$ possibilities to adding a *single* try block. Thus, the search space, which is already huge, explodes. Unfortunately, this prevents current tools including CONFUZZION from detecting certain vulnerabilities such as CVE-2014-0456.

Loops. Loops are, similarly to try/catch blocks, increasing significantly the search space. Vulnerabilities which require to execute some instuction blocks multiple times *might* not be detected by confusion. However, at least theoretically, CONFUZZION is able to generate a code equivalent to the unrolled version of the code to detect the vulnerability.

Java version 9 and above. CONFUZZION currently only targets Java version 8 and below and cannot generate programs with features from Java 9 or above. Thus, it currently cannot find type confusions in features that have been added to the Java API since version 9. For instance, the vulnerability of CVE-2018-2826 relies on a new API method introduced in Java 9 and cannot, therefore, be detected by CONFUZZION.

Handling JIT. The JVM optimizes code that runs often using different compilers such as the C1 or C2 compilers [36] [37]. Our approach does not aim at triggering compiler C1 or C2. CONFUZZION might thus not be unable to discover type confusion vulnerabilities which rely on bugs in these compilers. As far as we know, none of the other JVM fuzzers target these compilers either.

Compiling Input Programs. Our implementation currently relies on Soot to generate `.class` files. Improving speed by enabling faster program compilation is a key matter for future use of the fuzzer.

Intermediate Representation. CONFUZZION relies on Jimple to generate input programs. While this intermediate representation allows to generate any `.class` file that can be generated from Java source code, it cannot generate *all* `.class` files that a tool based on mutation at the bytecode level could generate. However, bytecode level mutation have a high chance of producing an invalid class file. Their execution on the JVM would thus mainly test the code responsible from class loading. With confuzzion we avoid generating invalid class files as we aim at testing the code behind class loading.

Incremental compilation. CONFUZZION compiles the intermediate representation mutant into an executable class file. One can think of incremental compilation [38] as a more effective way of adding code mutations to the mutant. For now, CONFUZZION does not use incremental compilation but it could be a performance improvement (number of mutants compiled per second). However, as the generated mutants are rather small, it may not bring a decent improvement over a standard complete compilation.

VII. RELATED WORK

A. Fuzzing tools

CONFUZZION is a fuzzer as it tries many inputs into a target program, the JVM. These inputs are obtained through high level Jimple mutations compared to AFL bit-level mutations.

[6] commit 4400200f7f0f4321610a8d042da52fffedbaa74c

It also uses the seed concept as it can take a predefined input before execution and then applies mutations on it.

The genetic aspect of fuzzers is not used by CONFUZZION because combining programs is not the same task as combining binary files. Also, because CONFUZZION is mainly blinded over the current score of a mutant, it cannot rank the different mutants to select and apply a genetic algorithm as AFL does.

While AFL [7] and CONFUZZION are based on mutations regarding a set of seed files, there exist generation-based fuzzers that are not using mutations but instead are constantly generating a new input. JQF [14] is an effective Java fuzzer that uses a generator-based structured fuzzing approach that can be guided by AFL or Zest [17]. This means the seed that feeds the generator will mutate according to the guidance and its sensors (coverage, exceptions, ...). In the end the seed controls the random behaviour of the generator but not directly the end result that is created according to generator rules. Kelinci [15] is another Java fuzzer but uses only AFL on an instrumented Java program. The instrumentation is similar to the AFL one, except that it is on Java bytecode. Here an intermediate program called Driver is used to link between AFL and Kelinci (C and Java) in order to transfer coverages and inputs. JAVA-AFL [16] is another mutation-based fuzzer for Java. It has many caveats according to its creators but it is one of the few fuzzers available for Java. Like Kelinci, JAVA-AFL brings AFL-like instrumentation to Java programs. CONFUZZION is faster than JAVA-AFL and Kelinci because of the absence of coverage that means no instrumentation needed.

B. Random test generation

Randoop [39] uses a list of public methods available for each class. When generating sequences, Randoop reuses previous sequences that are valid. CONFUZZION also uses a list of targets that corresponds to some classes, but it does not use the Randoop concept of sequences. In fact, sequences are defined within a simple method body, but CONFUZZION is working in a multi-level context where a value may come from a complex previous sequence like a field, a return value or a specific parameter. Inside CONFUZZION a sequence cannot be used independently as Randoop does.

Thanks to our approach that does not use sequences, CONFUZZION is less dependent on the seed for the result thus the Feedback-Controlled [40] approach is not used as-is by CONFUZZION. However, the fuzzer uses a similar approch to select the next method to call based on the number of previous crashes.

C. JVM implementation testing

Research on testing JVM implementations is at an advanced level with ClassFuzz [18] and ClassMing [19] being quite powerful. These tools aim at general JVM bugs, which can be found efficiently when targeting only the language features and not the related API. In contrast CONFUZZION aims at a specific class of bugs that is outside the scope of ClassFuzz and ClassMing. At the same time, CONFUZZION follows the work line of ClassFuzz and ClassMing and uses successive mutations on class files with Soot, but with different mutations. Another difference is that CONFUZZION does not require an initialisation seed and does not need a reference JVM enabling higher flexibility when testing. Interestingly, these approaches could be combined providing an interesting avenue for future research.

D. Test-program generation

In addition to JVM implementation testing, test-program generation is commonly used in compiler testing literature [41], [42], [43], [44], [45], [46], [47], [48], [49], [50]. The main goal of this line of work is to stress the compiler and reveal corner cases, typically in compiler optimization or parsing features of the compilers. To do so, most of the studies focus on the use of the language features, syntax and supported structures, while CONFUZZION focuses on the core language API and its semantics at the JVM implementation level. Another difference is that CONFUZZION aims at the checking of the program's semantic at run-time, after compilation.

VIII. CONCLUSION

Thanks to Confuzzion we now have a tool to generate complex programs at decent speed in order to find flaws inside the JVM such as type confusions or other object related issues. We measured a significant improvement in binomial method selection based on the failure rate over the classic uniform method selection. This leads to a faster finding of type confusions by concentrating efforts on those complex methods.

We believe that Confuzzion is the beginning of a global thinking about how type confusion can be detected inside the Java Virtual Machine. Although it seems primitive, it is also quite efficient when targeting specific classes. As such, to find unknown type confusions, we can list all modified classes since previous release and target each one (or a combination of some of them) in order to reduce the search space.

Also, the current median time needed to find a type confusion (4 hours for CVE-2017-3272) is small enough (< 24 hours) to be useful in practice. This is a direct consequence of the mean speed we achieved at more than 100 execs/s where *exec* represents the whole association of: applying a new mutation, adding a type confusion detector, compiling the entire program and finally executing it.

ACKNOWLEDGMENT

REFERENCES

[1] "CVE-2014-0456." Available from MITRE, CVE-ID CVE-2014-0456, Dec. 12 2013, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0456.

[2] "CVE-2015-4843." Available from MITRE, CVE-ID CVE-2015-4843, Jun. 24 2015, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4843.

[3] "CVE-2016-3587." Available from MITRE, CVE-ID CVE-2016-3587, Mar. 17 2016, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3587.

[4] "CVE-2017-3272." Available from MITRE, CVE-ID CVE-2017-3272, Dec. 06 2016, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3272.

[5] "CVE-2018-2826." Available from MITRE, CVE-ID CVE-2018-2826, Dec. 15 2017, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2826.

[6] I. Eauvidoum and d. noise, "Twenty years of Escaping the Java Sandbox," Phrack 2018, http://www.phrack.org/papers/escaping_the_java_sandbox.html.

[7] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2017.

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

[9] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy*, 2018.

[10] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[11] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534772

[12] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, vol. abs/1709.07101, 2017. [Online]. Available: http://arxiv.org/abs/1709.07101

[13] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Master's thesis, Karlsruhe Institute of Technology, 2018.

[14] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[15] R. Kersten, K. S. Luckow, and C. S. Pasareanu, "Poster: Afl-based fuzzing for java with kelinci," in *ACM Conference on Computer and Communications Security*, 2017.

[16] J. Judin, "Java-afl," https://github.com/Barro/java-afl, 2019.

[17] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Zest: Validity fuzzing and parametric generators for effective random testing," *CoRR*, vol. abs/1812.00078, 2018.

[18] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[19] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[20] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 4:1–4:36, 2020. [Online]. Available: https://doi.org/10.1145/3363562

[21] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. L. Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 695–705. [Online]. Available: https://doi.org/10.1145/3338906.3338941

[22] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," 03 2011, pp. 427–430.

[23] G. Fraser and A. Arcuri, "Handling test length bloat," *Softw. Test. Verification Reliab.*, vol. 23, no. 7, pp. 553–582, 2013. [Online]. Available: https://doi.org/10.1002/stvr.1495

[24] R. Vallée-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[25] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[26] A. Householder and J. Foote, "Probability-based parameter selection for black-box fuzz testing," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2012-TN-019, 2012. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28047

[27] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," *Proceedings - International Conference on Software Engineering*, pp. 75–84, 2007.

[29] G. Fraser and A. Arcuri, "EvoSuite," *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, no. September 2011, p. 416, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2025113.2025179

[30] A. Sakti, G. Pesant, and Y. G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, 2015.

[31] I. S. Prasetya, "Budget-aware random testing with T3: Benchmarking at the SBST2016 testing tool contest," *Proceedings - 9th International Workshop on Search-Based Software Testing, SBST 2016*, pp. 29–32, 2016.

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

[33] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp. 201–211, 2016.

[34] "Junit."

[35] "Oracle jdeps."

[36] B. Evans, "Understanding java jit compilation with jitwatch, part 1," https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html, 2014.

[37] OpenJDK, "Hotspot glossary of terms," http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html, 2006.

[38] Y. Liang and L. Yansheng, "An incremental compilation algorithm for the java programming language," in *2012 7th International Conference on Computer Science Education (ICCSE)*, 2012, pp. 1121–1124.

[39] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[40] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden, "Feedback-controlled random test generation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 316–326. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771805

[41] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for java jit compiler test system," 12 2003, pp. 20 – 23.

[42] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," 11 2019, pp. 305–316.

[43] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," 05 2016, pp. 180–190.

[44] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," *ACM SIGPLAN Notices*, vol. 47, pp. 335–346, 08 2012.

[45] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *ACM SIGPLAN Notices*, vol. 47, p. 283, 08 2012.

[46] A. Boujarwah and K. Saleh, "Compiler test case generation methods: A survey and assessment," *Information and Software Technology*, vol. 39, pp. 617–625, 09 1997.

[47] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," 06 2017, pp. 347–361.

[48] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. XIE, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 12 2018.

[49] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," 08 2019, pp. 223–234.

[50] C. Lidbury, A. Lascu, N. Chong, and A. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, pp. 65–76, 06 2015.

APPENDIX