This is a repository copy of *Private API Access and Functional Mocking in Automated Unit Test Generation*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/120346/

Version: Accepted Version

Proceedings Paper:

Arcuri, A., Fraser, G. and Just, R. (2017) Private API Access and Functional Mocking in Automated Unit Test Generation. In: Proceedings of 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 13-17 Mar 2017, Tokyo, Japan. IEEE , pp. 126-137. ISBN 10.1109/ICST.2017.19

https://doi.org/10.1109/ICST.2017.19

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Private API Access and Functional Mocking in Automated Unit Test Generation

Andrea Arcuri
Westerdals Oslo ACT
Oslo, Norway,
and SnT,
University of Luxembourg

Gordon Fraser
University of Sheffield
Sheffield, UK

René Just
University of Massachusetts
Amherst, MA, USA

***Abstract*—Not all object oriented code is easily testable: Dependency objects might be difficult or even impossible to instantiate, and object-oriented encapsulation makes testing potentially simple code difficult if it cannot easily be accessed. When this happens, then developers can resort to mock objects that simulate the complex dependencies, or circumvent object-oriented encapsulation and access private APIs directly through the use of, for example, Java reflection. Can automated unit test generation benefit from these techniques as well? In this paper we investigate this question by extending the EvoSuite unit test generation tool with the ability to directly access private APIs and to create mock objects using the popular Mockito framework. However, care needs to be taken that this does not impact the usefulness of the generated tests: For example, a test accessing a private field could later fail if that field is renamed, even if that renaming is part of a semantics-preserving refactoring. Such a failure would not be revealing a true regression bug, but is a *false positive*, which wastes the developer's time for investigating and fixing the test. Our experiments on the SF110 and Defects4J benchmarks confirm the anticipated improvements in terms of code coverage and bug finding, but also confirm the existence of false positives. However, by ensuring the test generator only uses mocking and reflection if there is no other way to reach some part of the code, their number remains small.**

I. Introduction

To support developers in the task of producing test suites for object-oriented code, tests can be generated automatically. Developers can then keep these tests for regression testing and execute them during continuous integration after every new code change, or they can manually inspect the tests to check if the behavior they capture represents incorrect behavior of the class under test (CUT). In both use cases, an essential prerequisite is that the automated test generation tool achieves sufficient coverage of all parts of the CUT. Although modern test generation tools can achieve an average code coverage ratio of 70% or more [13], there remain challenging aspects in object-oriented code, where even human testers have to resort to supportive technology when writing their unit tests.

Consider the simple code example in Figure 1: The class `PAFM` has one public method `example`, which takes as input an instance of the interface `AnInterface`. This interface defines one method `isOK`, which returns a boolean. Although the example seems very simple and small, it poses two challenges for testing class `PAFM`: First, the interface has no concrete class implementing it. This could happen if none has been developed yet, or if the interface represents a web service or RMI object that is not part of the classpath. Second, the `x` parameter object is passed to the private `checkIfOK` method only if the user executing the code is the root user (which is an artificial example, of course), which may be out of control of the tester, resulting in `checkIfOK` never being called. These are challenges that many state-of-the-art unit test generation tools (e.g., all tools participating in recent editions of a unit testing tool competition for Java [8], [28]) cannot overcome. The only possible test that these tools generated for the example above consists of passing a null value to the method `example`.

```
1  public interface AnInterface {
2    boolean isOK();
3  }
4
5  public class PAFM {
6
7    public void example(AnInterface x) {
8      if(System.getProperty("user.name").equals("root")) {
9        checkIfOK(x);
10     }
11   }
12
13   private boolean checkIfOK(AnInterface x){
14     if(x.isOK()){
15       return true;
16     } else {
17       return false;
18     }
19   }
20 }
```

Fig. 1. Source code example that cannot be tested without the use of mock objects and reflection.

In order to overcome these problems, we consider two extensions to automated unit test generation, based on techniques used during manual testing as well as by commercial tools like Agitar One[1]: First, we allow the unit test generator to not only instantiate and manipulate regular objects, but also to generate *mock* objects using the Mockito framework, and to determine what values method calls on the mock objects should return. Second, we allow the unit test generator to directly access private methods and fields by using Java reflection. Figure 2

[1] http://www.agitar.com/solutions/products/automated_junit_generation.html, accessed September 2016

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
 PAFM pAFM_0 = new PAFM();
 AnInterface anInterface0 = mock(AnInterface.class, new ViolatedAssumptionAnswer());
 doReturn(true).when(anInterface0).isOK();
 Boolean boolean0 = (Boolean)PrivateAccess.callMethod((Class<PAFM>) PAFM.class, pAFM_0, "checkIfOK", (Object) anInterface0,
       (Class<?>) AnInterface.class);
 assertTrue(boolean0);
}

@Test(timeout = 4000)
public void test1() throws Throwable {
 PAFM pAFM_0 = new PAFM();
 AnInterface anInterface0 = mock(AnInterface.class, new ViolatedAssumptionAnswer());
 doReturn(false).when(anInterface0).isOK();
 Boolean boolean0 = (Boolean)PrivateAccess.callMethod((Class<PAFM>) PAFM.class, pAFM_0, "checkIfOK", (Object) anInterface0,
       (Class<?>) AnInterface.class);
 assertFalse(boolean0);
}
```

Fig. 2. Tests generated by EvoSuite on the target class in Figure 1: Mock objects are created for the `AnInterface` interface, and reflection is used to access the private method `checkIfOK`.

shows two resulting tests cases: `test0` covers the true branch in method `checkIfOK`, `test1` covers the false branch.

However, mock objects and reflection are known to be susceptible to creating false positives [30]; i.e., tests that fail erroneously. For example, if a private method is renamed or removed, then a test accessing that method would fail at run time as the name of the method would be encoded as a string in the reflection call. Similarly, a mock object may lead to a false positive due to an invalid or outdated assumption about the implementation, which is encoded in the mock object. In order to avoid these problems, we introduce several optimizations to reduce the number of false positives, by avoiding known causes of false positives, and by minimizing the usage of mocking and reflection.

In detail, the contributions of this paper are as follows:

- A technique to integrate mock object generation in search-based unit test generation, where the configuration of the mocks becomes part of the search problem.
- A technique to integrate reflection on private methods and fields in search-based unit test generation, while ensuring that API changes do not lead to false positives.
- An exploration study to determine the optimal parameter settings for these techniques.
- An empirical study on the performance of these techniques in terms of their effects on code coverage and fault detection, using the SF110 and Defects4J benchmarks.
- An empirical study on the effects of these techniques on false positives and code evolution.

II. Background

A. Unit Test Generation

Unit tests are an essential part of software development, and to support developers with their creation various techniques have been proposed to automatically generate tests. A common approach is to generate *random* sequences of method calls, for example implemented for Java in tools such as Randoop [24], JTExpert [29], GRT [21] or T3 [26]. To reduce the number of tests and to increase the code coverage achieved, techniques based on *search-based software testing* (SBST) [1], [17] cast test generation as an optimization problem, which then can be addressed with techniques like Genetic Algorithms (GAs). Examples of SBST tools for Java are eToc [34], EvoSuite [12] and TestFul [7]. Alternatively, approaches based on *dynamic symbolic execution* (DSE) (e.g., Pex [32]) use constraint solvers to generate test data and to explore the possible paths through a program, although they often require the tester to manually write test drivers [11].

We implemented the techniques presented in this paper as extensions to the search-based EvoSuite unit test generation tool; a similar integration into random or DSE-based tools would be possible. The search-based approach to unit test generation is very flexible and makes it easy to integrate alternative objectives, such as to limit the application of mocking and reflection to only cases that cannot be covered otherwise.

B. Private API Access

In the Java programming language, instance fields and methods have four different possible kinds of visibility modifiers: `public`, `protected`, `private` or *package-private* (the default if no modifier is specified). A test case that is located in the same package as the CUT (as is usually the case) can directly access all public, protected, and package-private methods and fields. However, it cannot access the private ones.

One goal in test data generation is to maximize coverage on the CUT, which includes also all of its private methods. Consider the *checkIfOk* method in Figure 1. If a test generation tool fails to generate a test whose execution evaluates the if-condition in Line 8 to true, then the `checkIfOk` method might never be tested. A possible solution to exercise `checkIfOK` is to use Java's *reflection* API to directly call the private method.

False Positives: Consider, for example, a refactoring (i.e., a semantics-preserving change) in which the name of `checkIfOK` is changed into `foo`. As the invocation of private methods with reflection will typically include the method name explicitly as a string (see Figure 2) it will not be renamed,

even if the developer uses an automated refactoring tool to perform the method renaming. Unless the developer manually modifies the strings representing the private method name in every single generated test, any test case that uses reflection to access `checkIfOK` would now fail, even though the CUT has no regression defect. Such reflection-related changes are out of scope of current automated refactoring tools, and hence a developer could end up spending a lot of time investigating all those failing tests.

C. Functional Mocking

Mocking is a common approach in unit testing to isolate a class from its dependencies by using a replacement of a dependency class instead of the original one. Consider the `example` and `checkIfOK` methods in Figure 1. Depending on the complexity of a class that implements `AnInterface`, it might not be possible for a test data generation tool to configure an instance of that class such that a call to its method `isOK` returns true. Even worse, it may even be the case that no concrete class for `AnInterface` is available to instantiate. This could, for example, happen when dealing with remote method invocations (RMI), or Java Enterprise Edition (JEE) code that needs to be run in a JEE container (e.g., WildFly[2] or GlassFish[3]). In such a case, a human tester can create a *mock* object to allow unit-testing of a method that expects an argument of type `AnInterface`.

When manually writing tests, it is common for developers to use a mocking framework [23] to simplify some of the input parameters. For example the mocking framework Mockito[4] is among the top-10 most used Java libraries, hosted on GitHub[5]. When open-source software has test cases, 23% of the times a mocking framework is used [23], where the JEE interface `HttpServletRequest` is the most mocked class.

There are several different approaches to mocking, depending on the functionality of the replacement class: A *stub* is a replacement with a fixed (usually default) behavior, while a *mock* not only replaces the original class, it also has some partial behavior (mimicking the intended behavior of the class) that needs to be configured, usually during the preparation of the test execution. This paper uses the term "mock" as a synonym for some common terms like *fake*, *dummy*, or *test double*. Generally, a mock `M` for a class/interface `X` can be seen as an implementation of `X`, where the return values of each method invocation can be controlled directly in the test.

Since our implementation uses the Mockito framework to create mock objects, we provide some background about this framework. Consider the following example:

```
public static void foo(X x) {
 if(x.isFoo()){
   //...
 }
}
```

[2]http://wildfly.org, accessed January 2016.

[3]https://glassfish.java.net, accessed January 2016.

[4]http://mockito.org, accessed January 2016.

[5]http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby, accessed January 2016.

To test the method `foo`, using Mockito, a tester could write:

```
X x = mock(X.class);
when(x.isFoo()).thenReturn(true);
foo(x);
```

In this example, `mock` and `when` are static methods of the class `org.mockito.Mockito`. The method `mock` creates a mocked instance of type `X`, whereas `when` specifies what should happen when its input (and latest method call on the instance `x`) is called in the test. In this particular case, it is specified that `x.isFoo()` should return the value `true`.

A mock object can be used even when a method is called several times, and each time a different value is needed. For example, consider the following code to test:

```
public static void doubleCall(X x) {
 if(x.next()==5 && x.next()==42){
   //...
 }
}
```

Using Mockito, a test case could be written as follows:

```
X x = mock(X.class);
when(x.next()).thenReturn(5,42);
doubleCall(x);
```

This specifies that the first time `next()` is called the value 5 should be returned, whereas the second time (and all successive times) the value 42 will be returned.

False Positives: Mock objects can be used to verify the order of calls performed on them. For example, Agitar's mocking framework expects calls on mock objects to be performed in the same order as recorded, and thus swapping the operands of a commutative operation (e.g., `getX()+getY()` ⇒ `getY()+getX()`) leads to a false positive [30], even though this is a semantics-preserving change. Similarly, mock objects are often used to verify that only expected methods are invoked. This can be too restrictive and cause false positives (e.g., replacing a field access with a call to a corresponding getter method).

D. False Positives in Unit Test Generation

There are at least two possible scenarios for the use of automatically generated tests: (1) manually inspecting the tests to check whether any of them is capturing an incorrect behavior of the CUTs or (2) using the tests for regression testing (e.g., to run them at every new code change in a continuous integration system such as Jenkins[6]). While there is no precise definition of when a test failure is a false positive, we can loosely distinguish two possible reasons for these scenarios: First, a test might represent an unrealistic execution or use of the CUT. For example, Gross et al. [16] showed that all 181 failures reported by a test generator on an example application violated implicit preconditions, and were thus false positives of this type. Second, a test that makes wrong or overly restrictive assumptions might fail later during regression testing, even if a CUT has only been changed in a semantics-preserving way (e.g., refactoring). Both types of false positives are harmful: software engineers

[6]https://jenkins-ci.org/, accessed September 2016.

need to manually inspect them, which represents a waste of time and resources.

Reflection and mocking can cause both types of false positives. In particular, in a prior study we showed that 46% of generated regression tests that involved reflection or mocking failed due to false positives [30] of the second type. For this paper, we further analyzed these false positives and identified that 65% were caused by unexpected method calls on mock objects, and 35% failed to changes in the private API. Our goal is to integrate private API access and mocking into unit test generation; when doing so, we address both types of false positives by minimizing the occurrences of reflection and mocking, and we provide techniques to prevent the second type by construction.

III. PAFM: Integrating Private API Access and Functional Mocking into Unit Test Generation

We integrated private API access (*PA*) and functional mocking (*FM*) into EvoSuite. This section details our implementation choices and how they reduce false positives.

A. Search-based Test Generation in EvoSuite

The integration of reflection and mocking into EvoSuite's search algorithm uses two points of access that influence which sequences of calls are constructed during the search. First, there are the search operators that insert new objects and calls into a test case, and second, there is the static analysis that informs the modification operators with the possible choices.

EvoSuite uses a GA to evolve sets of candidate unit test cases, in an optimization process that aims to maximize the code coverage. Fitness functions based on code coverage determine which individuals in a population of candidate test suites are selected for reproduction in the GA. Selected individual are susceptible to crossover, where tests are exchanged between two parent test sets, and mutation, whereby tests are added, deleted, or modified. These modifications are the first entry point for integrating mocking and reflection: Each unit test is represented as a sequence of statements, and modifications of such a test include replacing values or adding and deleting statements. EvoSuite defines different types of statements, for example for calls to constructors or methods, to generate primitive objects and arrays, or to assign values to fields.

The modifications are guided by additional information about the available calls. In particular, EvoSuite statically collects information about which method and constructor calls are part of the SUT, which calls are available to generate dependency objects, and which methods can alter the state of instances of these classes; this information is sometimes referred to as the *test cluster*. For example, when appending a new statement to a test, EvoSuite will select one of the methods of the CUT, and when inserting a call on an existing object, EvoSuite will select a modifier and insert a call somewhere before a usage of the object. Any dependencies of the new statement (e.g., parameters of the method call, receiver object of the method call, instance for the field, etc.) are resolved by either selecting existing values in the test, or by recursively adding calls that generate appropriate objects.

B. Integrating Private API Access

Using reflection might lead to undesirable boilerplate code in a test case, but this complexity can be hidden away in auxiliary libraries, such that in the tests one only gets to see calls to support methods. For our extension to EvoSuite we have created a helper class `PrivateAccess`, which is part of the runtime library that needs to be available during test execution. This class provides static methods such as `callMethod`:

```
PrivateAccess.callMethod(Bar.class, bar, "aPrivateMethod",
    42, int.class);
```

Here, the private method `aPrivateMethod` of class `Bar` is called on an instance `bar`, with 42 as input parameter. For each parameter of the called method the helper method `callMethod` also requires the parameter types (`int.class` in this case). This is in order to identify the correct method in the case of overloading. Internally, `callMethod` uses Java reflection to retrieve a method object from class `Bar.class`, and then directly invokes it. For methods with more parameters, the `PrivateAccess` class provides additional overloaded versions of `callMethod`, for example to provide all parameter objects and their declared types in arrays.

To make EvoSuite consider calls to private API, we have extended its static analysis that creates the test cluster, such that, for each private method, a call to the `callMethod` method of the `PrivateAccess` class is added, where only the parameters (receiver object and parameter objects) are susceptible to mutation. Internally, this is realized by new types of statements representing access to private methods and private fields. During regular mutation of tests, for example when deciding which new statement to add to the test, EvoSuite will as a consequence also consider all the private API contained in the test cluster.

Similarly to calling private methods, to set the CUT in a suitable state, private fields can be accessed in the same way using reflection and the helper method `setVariable`, which again takes a class, the instance object, the field name, and the value and type to assign to the field.

C. Integrating Functional Mocking

To integrate mocks into test generation, we defined a further new type of statement (*mock statement*) in EvoSuite, which represents the creation and configuration of a specific mock object. To distinguish this kind of mock object from the *environment mocks* [5] used in EvoSuite, we use the term *functional mocks* (*FM*).

During insertion of new statements, EvoSuite tries to resolve dependencies (e.g., parameter objects) either by re-using objects declared in earlier statements of the same test, or by recursively inserting new calls to generate new instances of the required dependency objects. When EvoSuite needs to instantiate an input object of type `X`, with probability P_{FM} a mock object will be instantiated. This means that a mock statement is inserted into the test before the call for which dependencies

are resolved. The mock statement assigns the mock object to a variable, which in turn is then used as the concrete parameter for the new call.

Initially, the mock statement has no parameters itself, it simply is responsible to call the `mock` method with an appropriate type. Thus, the first time a mock object is created in a test, it will have no `when` methods defined, and this means that when the test is executed as part of fitness evaluation, the mock object will return default values when its methods are called (e.g., 0 for methods returning numbers and `null` for the ones returning objects). However, each mock is run with a listener, which keeps track of which methods were called during a test case execution. After a test is executed, we add to it a `when` method for each mock invocation, with one or more random values in the `thenReturn` methods based on how many times the methods were called. To avoid very large test cases, each `thenReturn` invocation will have a maximum of n inputs (e.g., in this paper we used $n = 5$). Recall that if a mocked method is called more times than the number of inputs in the `thenReturn` method, then the last input is returned.

Although the representation as Java code consists of several lines for the call to `mock` and all calls to `when`/`thenReturn`, these are all part of the same mock statement internally in EvoSuite. Each of the `thenReturn` entries represents one additional parameter of the mock statement. Thus, during the search the search operators that modify the test cases will be applied as well on the inputs of `thenReturn` methods. However, the calls to `when` will not be modified by the search operators (e.g., delete, add new ones, or change their inputs), as they would not bring any benefit. For the same reason, no search operator will add functional calls to any mock instance, and a mock instance will not be re-used more than once as input parameter for the CUT.

During the search, it might happen that a mock object will be used differently based on how the test is modified. Some of its methods might not be called any more, and other new ones might be called. For example, consider the following case:

```
public static void foo(boolean b, X x){
 if(b){
  doSomething(x.first());
 } else {
  doSomething(x.second());
  doSomething(x.second());
 }
}
```

For this example, we might have a test like the following:

```
X x = mock(X.class);
when(x.first()).thenReturn(1);
foo(true, x);
```

Then, in successive generations EvoSuite might mutate the `true` value in `foo` into `false`. As a result, the call `when(x.first())` will be redundant, as `x.first()` would not be called any more in the test inside `foo`. In this case, *after* a test is run and evaluated, EvoSuite will remove all `when` calls that are no longer needed. Furthermore, *before* a test will be mutated in the next generations of the search, any needed `when` method (e.g., `when(x.second())`) will be added with random inputs. Similarly, if a mock method is called less or more times, the cardinalities of the `thenReturn` methods will be modified as well (i.e., remove no longer needed inputs and add new needed ones at random).

D. Avoiding False Positives

1) Private API Access: A special case is given when *PA* injects null values to private fields, which would in many cases lead to null pointer exceptions (NPEs) when the methods of the CUT are called. Because EvoSuite explicitly aims to trigger unexpected exceptions [14], the underlying fitness function would reward those tests, and so the final generated test suites could be just rigged of pointless test cases throwing NPEs. To avoid this issue, during the search we explicitly prevent the use of *PA* with null arguments.

Recall that removing an obsolete private method or renaming a private method inevitably causes a test that reflectively calls it to fail. Each such failing test represents a false positive that a developer needs to investigate. To overcome this problem, we use a simple yet effective approach: our implementation relies on JUnit's `AssumptionViolatedException`[7] (AVE). If a method in our `PrivateAccess` library tries to access a field or method that does not exist any more, then it throws an `AssumptionViolatedException`. This will not fail the test (and so it will not become an expensive false positive), but it will cause JUnit to *skip* the remainder of the test, potentially causing lower coverage. However, new tests can be automatically recreated [9].

2) Functional Mocking: In open-source software, the most used Mockito method is `verify` [23]. This is used to check if a mocked method was indeed called with a given input, and throws an exception if not (thus failing the test). For example, after `foo(true, x)` a tester could add `verify(x).first()` to check if the method `first()` was called inside `foo()`. However, we have decided to not use any `verify` calls in EvoSuite. The reason is to avoid "brittle" tests, i.e., tests that could fail in successive modifications of the CUT even though no regression bug is introduced, and so become false positives. To find bugs, we rely on assertions on the return values of the CUT methods, and not on their interactions with the input parameters, as these could change without altering the external behavior of the CUT.

Even when no `verify` is used, mocking frameworks can still lead to manually written tests that are brittle. For example, if a class X is mocked, and in an updated version of the CUT new calls to X are added/changed (without modifying the semantic of the CUT), then those will just return default values (e.g., 0 for integers and null for objects). This could well lead to the tests failing, although the new CUT is semantically equivalent (e.g., it was just a refactoring). This is a major issue in manually written tests, where care needs to be taken when using Mockito (e.g., do not use it on classes/interfaces that likely will go through few updates in the future).

Similarly to the case of *PA*, an option to avoid this problem is to change the behavior of Mockito to throw

[7]http://junit.org/junit4/javadoc/4.12/org/junit/AssumptionViolatedException.html, accessed September 2016.

a `AssumptionViolatedException` by default. This is usually not done in manually written tests, as it would force the developer to set the mocks (e.g., add `when` calls with default values) for every single mocked method that is called in the CUT, even when their return value is of no interest for the tests. This would be tedious, and thus likely not very practical. On the other hand, in automatically generated tests we can automatically add all those mock setup calls, and therefore use `AssumptionViolatedException` as default behavior (except for void methods) to guard against future changes. However, a minor side effect is that the recommended pattern `when(X.foo()).thenReturn(...)` cannot be used (as the call to `X.foo()` would throw the AVE), and we need to rather rewrite it as `doReturn(...).when(X).foo()`, which is arguably less readable.

E. Minimizing Private API Access and Functional Mocking

Both, private access and functional mocking may lead to false positives. Consequently, it is desirable to *minimize* the usage of these techniques, and only retain resulting tests if there are no alternative tests that achieve the same coverage without using reflection or mocking. We integrated this objective into the search algorithm in two complementary ways.

First, search-based test generation techniques usually maintain an archive of test cases during the search [25], [27]. This archive retains one test for each coverage goal that has been covered during the search, and it allows the search to focus on those coverage goals not yet covered. EvoSuite maintains the archive as a map, where for each testing target EvoSuite keeps track of the best test seen for this target. At the end of the search, the contents of this archive represent the final test suite, and typically some post-processing steps such as minimization are applied to improve the readability of the tests. We modified this archive such that if a new test covers a target without using *PA* or *FM*, then it will replace a previous test using *PA* or *FM*, regardless of their size (by default, EvoSuite retains the smaller of two tests that cover the same target).

Second, in order to make sure that the use of *PA* and *FM* is a last resort for the search space exploration, we integrated a number of parameters into the search operators. These parameters define how likely the search can resort to *PA* or *FM* rather than standard API calls and real objects, and when during the search *PA* and *FM* should be activated — from the very beginning of the search, or rather later on (e.g., after 50% of generations have already been evaluated)? The latter could be for example more preferable (i.e., more efficient) if most tests using *PA* will be be superseded in the archive:

- The S_{PA} parameter specifies the percentage of the search budget that should be evaluated before starting to use *PA*.
- The P_{PA} parameter decides how often reflection on a private field or method is used, rather than using a public one. It thus represents the probability of activating *PA*.
- The S_{FM} parameter specifies the percentage of the search budget that should be evaluated before functional mocking is activated.
- The P_{FM} parameter specifies the probability of using a mock object instead of a real instance. Note that if an interface has no concrete class, then a mock object will be instantiated regardless of P_{FM}.

IV. Evaluation

We aimed at evaluating how the use of Private API Access (*PA*) and Functional Mocking (*FM*) together (*PAFM*) during test generation affects the effectiveness and brittleness of the tests. In particular, the four addressed research questions are:
RQ1: What *PAFM* configuration maximizes code coverage?
RQ2: Does *PAFM* improve code coverage?
RQ3: Does *PAFM* improve fault detection?
RQ4: Does *PAFM* affect test brittleness?

A. Subject Programs

To answer our research questions, we used two different datasets: the SF110 corpus [13] and Defects4J [19]. SF110 is a collection of 100 open source projects randomly chosen from SourceForge, plus the top 10 most popular ones at the time in which the corpus was defined. In total, the SF110 corpus consists of 23,886 Java CUTs, totalling more than 6.6 million lines of code. The Defects4J [19] dataset consists of 357 real faults from five open source projects: *JFreeChart* (26 faults), *Google Closure compiler* (133 faults), *Apache Commons Lang* (65 faults), *Apache Commons Math* (106 faults), and *Joda Time* (27 faults). For each fault, Defects4J provides a buggy and fixed program version with a minimized change that represents the isolated bug fix. It further provides information about the classes relevant to the fault (e.g., classes modified by the bug fix, classes loaded by the fault-revealing test, etc.).

B. Experimental Setup

For the experiments carried out in this paper, we used the default configuration of EvoSuite, which is supposed to show good results on average [4]. In each experiment, the search phase for EvoSuite was executed until either a timeout of two minutes or 100% code coverage was reached. For each run we collected data on the achieved branch coverage as reported by EvoSuite. Statistics on the number of mock objects and reflection calls were collected by parsing and analyzing the generated tests. For experiments on Defects4J we collected bug detection data using Defects4J's infrastructure.

Because EvoSuite is based on randomized algorithms, each experiment was repeated several times with different random seeds, to obtain reliable results from which to draw sound conclusions. The results were then analyzed following standard guidelines [3]. In particular, to assess statistical difference we used the non-parametric Mann–Whitney–Wilcoxon U-test, and the Vargha-Delaney $\hat{A}_{12}$ effect size.

C. RQ1: What PAFM configuration maximizes code coverage?

To answer **RQ1**, we investigated different settings for the probability P_{PA} of applying private access, and when starting to apply it (S_{PA}; recall Section III-E). We did the same for the probability P_{FM} of using functional mocking, and its starting time S_{FM}. This gave us four different parameters to tune.

TABLE I
BRANCH COVERAGE COMPARISON OF *Base* WITH BEST CONFIGURATION FOR *PA*, *FM* AND *PAFM* ON 110 CLASSES.

Name	Configuration	Coverage
Base	$P_{FM}=0, P_{PA}=0$	72.1%
PA	$P_{FM}=0, P_{PA}=0.50, S_{PA}=80\%$	74.1%
FM	$P_{FM}=0.50, S_{FM}=30\%, P_{PA}=0$	74.8%
PAFM	$P_{FM}=0.80, S_{FM}=50\%, P_{PA}=0.50, S_{PA}=80\%$	76.8%

For the two probabilities P_{PA} and P_{FM}, we considered the four different values $[0.0, 0.3, 0.5, 0.8]$. For the starting point percentages S_{PA} and S_{FM}, we used $[0.3, 0.5, 0.8, 1]$, where 1 means "never start", i.e., deactivate *PA* or *FM*. The combination of four different parameters with four different values did not lead to $4^4 = 256$ combinations, as some of them are redundant. For example, if $P_{PA} = 0$, then the value of S_{PA} becomes irrelevant, and vice-versa if $S_{PA} = 1$. So, to study *PA* we had all $3^2 = 9$ combinations of $P_{PA}, S_{PA} \in [\ 0.3, 0.5, 0.8]$, with a further $P_{PA} = 0, S_{PA} = 1$ to study when *PA* is not used. Similarly, for *FM* we had 9 combinations in which it is used and 1 in which it is not. All together, this led to $(9+1)\times(9+1) = 100$ different configuration settings.

Due to the large number of configurations to tune, we did not use the whole SF110 as case study. For each of the 110 projects in SF110, we selected one CUT at random. Then, on each CUT we executed EvoSuite with the 100 different configurations, and collected data on the resulting branch coverage. Each experiments was repeated 3 times, giving $110 \times 3 = 330$ data points per configuration. In total, this led to $330 \times 100 = 33{,}000$ runs of EvoSuite.

Table I shows the results for the *Base* configuration (no *PA* and no *FM*), the best configuration in which *PA* is used but not *FM*, then the other way around, and finally the best configuration for when both are used at the same time (*PAFM*). The results in Table I show that both *PA* and *FM* improve upon the *Base* configuration, and this improvement is even higher when they are combined.

The best configuration improved branch coverage from 72.1% to 76.8%, i.e., a +4.7% improvement. It is interesting to analyze which configuration values maximized the coverage: For example, *PA* is best used only near the end of the search, as $S_{PA} = 80\%$. This is not unexpected: Adding private fields and methods to the search increases the search space considerably. The search might end up spending a lot time modifying private fields, when for a given CUT it could be easier to just do it through its public methods. However, after 80% of the search, it might well be that EvoSuite covered everything possible with the public methods, and so introducing *PA* does not hinder the coverage. The best probability of applying *PA* at that point is $P_{PA} = 0.5$; higher values inhibit the search, as focusing on only private methods would prevent coverage on any non-private methods in the CUT. *FM* might have side effects on the search landscape, and the creation of mock objects with Mockito causes a considerable overhead, and as such its starting percentage is $S_{FM} = 0.5$ and not the lowest 0.3.

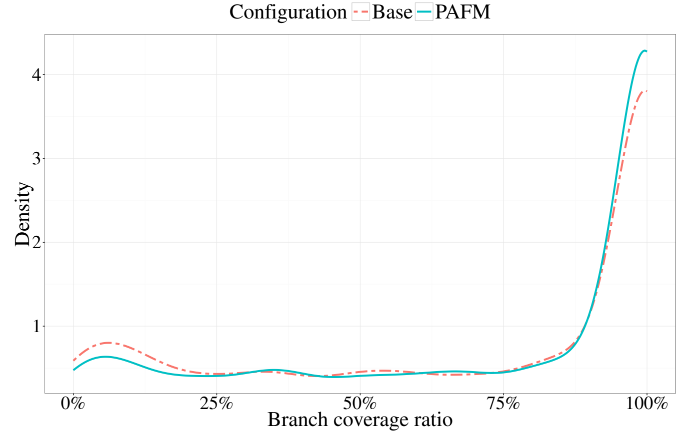

Fig. 3. Branch coverage comparison of *Base* configuration with *PAFM*.

> *The best configuration for* **PAFM** *is:* $P_{PA} = 0.5$, $S_{PA} = 80\%$, $P_{FM} = 0.8$ *and* $S_{FM} = 50\%$.

These settings are the ones used for all successive experiments with *PAFM*.

D. *RQ2: Does* PAFM *improve code coverage?*

The results of **RQ1** identify the best configuration for *PAFM* in terms of coverage. However, the +4.7% coverage improvement needs to interpreted with care, as it might well be an overestimation. Even though we used repetitions to minimize the threat of noise in the data influencing the result, the experiment focused more on a breadth of different configurations rather than reducing noise on individual classes/configurations. To reduce this threat to validity, when answering **RQ2** we therefore conducted a new experiment on a stratified sample of 1,000 classes from SF110, trying to sample uniformly from each project (i.e., about 9 classes per project were used). On these 1,000 CUTs, we ran both *Base* and the best *PAFM* configuration 10 times. This led to $1{,}000 \times 2 \times 10 = 20{,}000$ runs of EvoSuite.

The results of this experiment confirm the results of **RQ1**: the *Base* configuration obtained a 70.5% average branch coverage, whereas *PAFM* obtained 73.8%. This is a +3.3% improvement. This improvement on the total means that *PAFM* achieves coverage of +11.2% of the coverage goals missed by *Base*.

Figure 3 compares the distribution of coverage values for the *Base* and *PAFM* configurations. There is a clear spike in density to the far right of the plots, showing that EvoSuite obtains very high branch coverage already without *PAFM*. In these cases *PAFM* cannot make any further improvement, and this influences the magnitude of the observed increase in coverage. However, the plot shows a clear reduction of the cases where coverage is low (0%–25%), and a clear increase in the number of cases where coverage is very high (90%–100%) when using *PAFM*. A paired U-test (1,000 average coverage values for *Base* paired with 1,000 values of *PAFM* on the same CUTs) confirms this observation with a p-value very close to

TABLE II
NUMBER OF FAILURE-TRIGGERING TEST SUITES FOR THE BUGS IN DEFECTS4J FOR *Base* AND *PAFM*. *Triggering total* GIVES THE TOTAL NUMBER OF BUGS FOR WHICH AT LEAST ONE FAILURE-TRIGGERING TEST SUITE WAS GENERATED ACROSS ALL 30 RUNS. *Triggering average* GIVES THE AVERAGE NUMBER OF BUGS FOR WHICH A FAILURE-TRIGGERING TEST SUITE WAS GENERATED IN A SINGLE RUN.

Project	Bugs	Triggering total		Triggering average		p-value
		Base	*PAFM*	*Base*	*PAFM*	
Chart	26	23	25	15.4	14.7	0.346
Closure	133	25	32	10.7	12.2	0.030
Lang	65	42	46	23.7	26.4	0.286
Math	106	65	81	42.9	50.5	0.002
Time	27	18	18	11.3	11.2	0.913
Total:	357	173	202	103.9	115.0	0.002

TABLE III
PAFM STATISTICS ON THE FAILURE-TRIGGERING TESTS: PERCENTAGE OF TESTS THAT USED *PAFM*, AND AVERAGE NUMBER OF *PA* OR *FM* RELATED STATEMENTS.

Project	Test %	*PA* Methods	*PA* Fields	*FM* Objects	*FM* Calls
Chart	11.8%	0.0	0.0	0.1	0.1
Closure	11.0%	0.1	0.0	0.1	0.1
Lang	9.0%	0.1	0.0	0.1	0.1
Math	14.8%	0.0	0.0	0.2	0.2
Time	10.6%	0.0	0.0	0.1	0.1
All	12.8%	0.0	0.0	0.1	0.2

0, which means there are more cases in which *PAFM* leads to an improvement than the other way round.

*Using **PAFM** covers +11.2% of the uncovered branches, thus increasing branch coverage by +3.3%.*

E. RQ3: Does PAFM *improve fault detection?*

We used Defects4J to analyze the fault detection capability of *PAFM*. Although mutation analysis is a viable alternative for this kind of experiment [20], we preferred to use the 357 real faults provided by the Defects4J benchmark. For each of the 357 faults in Defects4J, we executed EvoSuite with the configurations *Base* and *PAFM* 30 times on the bug-free CUTs. Once the test suites were generated, we executed each of them on the buggy version of the CUT to check if any tests fail. Table II shows the results of this analysis. Because EvoSuite uses a randomized algorithm it may happen that, for a particular CUT, a failure-triggering test suite was generated only in a subset of the 30 runs. Therefore, we make a distinction between the total number of bugs for which a failure-triggering test suite was generated in all 30 runs and the average number of bugs for which a failure-triggering test suite was generated in a single run.

The results in Table II show that *PAFM* led to 11 more failure-triggering test suites on average—a $\frac{115-103.9}{103.9} = 10.6\%$ improvement. However, this does not mean that these tests are revealing the bugs, as they could be false positives.

Table III summarizes statistics on the amount of *PAFM* in the failure-triggering tests: On average, 12.8% of the failing tests use some kind of mocking or reflection. The *PA* Methods and *PA* Fields columns show the average number of reflection statements per test, and only Closure and Lang use at least some reflection on methods. Mock objects are used in all projects; i.e., on average there are 0.1 mock objects per failing test, and 0.2 `doReturn` calls per failing test. These numbers suggest that the mechanisms to reduce *PAFM* to only the absolutely necessary cases are effective. We also note that the AVE mechanism is effective, as on average over all bugs and random seeds there are 0.4 AVEs per test suite; these AVEs would otherwise lead to false positives.

To investigate the effects of *PAFM* on false positives, we thus focus on the 36 bugs for which only *PAFM* generated a failure-triggering test suite. (Note that the 202 − 173 = 29 in Table II is due to *Base* finding 7 bugs that *PAFM* did not.) In order to determine whether *PAFM* leads to false positives, we manually investigated the test suites that triggered these failure. As manually investigating 36 program versions of a complex application with 30 test suites each is a time-consuming task, we sampled the bugs whose detection is most likely related to *PAFM* as follows: To avoid looking at spurious results, we only analyzed the bugs that, being revealed by chance at least once in the 30 runs of *PAFM*, have less than 1% probability of never being triggered in any of the 30 runs by *Base*. Given p being the most likely estimation of fault detection based on s successful runs with *PAFM* (i.e., $p = \frac{s}{n}$), we looked at the probability $k < 0.01$ that no run in *Base* found the bug while p is independent from whether *PAFM* is used or not. In other words, we look at the lowest value for s for which we can have enough confidence that the bug was found due to *PAFM* and not just by chance. This means solving $min_s(1-(\frac{s}{30}))^{30} < 0.01$, which leads to $s = 5$. Overall, 15 out of 36 bugs were found at least 5 out of 30 times (Chart-25, Closure-1, Closure-68, Closure-76, Lang-17, Math-9, Math-34, Math-39, Math-64, Math-67, Math-71, Math-76, Math-84, Math-86, Math-100).

Each author of the paper independently analyzed all failing tests for all 15 and classified them as either fault revealing or false positive. Each discrepancy in the classification was then analyzed and discussed until a consensus was reached.

There were 4 cases that were clearly true positives: Chart-25, Closure-68, Closure-76 and Math-100. In these cases a numerical value was computed, and then an assertion on the returned value failed in the JUnit tests. On the other hand, there were 4 bugs for which we identified both true and false positives among the failing tests, and another 4 bugs only had false positives. For these 8 bugs, there were several different reasons for which a test resulted in a false positive.

Invalid mock objects. In two cases (Math-76 and Math-86), the false positive was due to a mocked object with inconsistent state, e.g., a matrix/vector where the *size* method returns an invalid negative value, which could never happen on the original non-mocked class. Because the object is invalid, the generated tests throw an exception of type X (e.g., a null pointer or array out of bounds exception), and that is caught in a try/catch. However, when those tests are run on the buggy version of the CUT, a different exception is thrown of type Y due to different

code being executed, making the test fail (the tests generated by EvoSuite check for the type of expected exceptions that should be thrown). A possible mitigation for this type of false positive might be to not check for exception types when a test is using any *FM* calls.

Loop counters and timeouts. To avoid that test cases execute too long or end up in an infinite loop, EvoSuite uses (a) bytecode instrumentation to count loop executions, and (b) timeouts on unit tests. If a loop is executed too many times (e.g., 10,000 times), EvoSuite throws a dedicated exception. In the cases of Math-39 and Math-71, the bug made the CUT execute fewer or more iterations, causing either an unexpected exception of this type, or the absence of an expected exception, both of which cause the JUnit test to fail. This, however, is not directly related to the bug, and is thus a false positive. A possible mitigation for this issue would be to never fail a test when an exception is thrown by the loop check, which can be implemented with a JUnit `@Rule`. Timeouts, implemented with the `@Test(timeout=4000)` annotation seen in Figure 2), caused a false positive for Lang-17, where the failure was due to a non-functional property and not on the actual bug we investigated. Neither of these 3 cases (Lang-17, Math-39 and Math-71) is directly related to the use of *PAFM*, as they could happen for *Base* as well. However, *PAFM* leads to higher code coverage, and so it is more likely that these cases occur.

Null values in reflection. For Math-9 and Math-34, *PA* set some private fields to null values, and thus put the CUT in an inconsistent state. These particular tests threw an exception that is of a different type than the one thrown in the buggy version, as different code is executed. Similarly to Math-76 and Math-86, this leads to false positives. However, this kind of problem should have been prevented in the first place, as EvoSuite does not use null values in *PA* for fields during the test generation (see Section III-D1). However, in this case it happened in the successive phases that can modify the generated tests, like test minimization. This problem can be fixed by enforcing such constraints throughout all the phases of test generation.

Bugs in EvoSuite. The last out of these 8 false positives was due to an AVE, which should have caused the test to be ignored rather than counting it as a failure. However, the try/catch block in the generated test caught this exception, then compared it with an expected exception type, and declared it a failure because the type did not match. This problem could be easily solved by never catching an AVE in a generated JUnit test (or re-throwing it if caught).

Controversial cases. While the 12 bugs discussed so far were relatively easy to analyse, the remaining 3 were not. In Math-67, there was one failing test that reveals an actual bug, but not the specific Defects4J bug. Still, we count it as a true positive. The case of Math-84 was similar to the one of Math-39 and Math-71, i.e., EvoSuite throws an exception due to a loop that is executed too often. However, here the bug was indeed related to performance, and the original manual tests for that bug also checked for loop executions. As such, we did not consider it as a false positive. Finally, Closure-1 calls a method with a null parameter using *PA*, which causes an exception only in the buggy version of the CUT. However, in practice this method cannot be executed with a null value through its public API, so even though the test reveals the difference, it could arguably be counted as a false positive.

In summary, out of the 15 analysed bugs, we identified 10 types of true failures, and 9 types of false positives. However, in most cases the manual analysis led to the identification of ways to avoid these types of false positives to some extent. The degree to which this mitigates the problem of false positives and the degree to which this counters the coverage benefits remains to be investigated as future work.

*On average, test suites generated using **PAFM** triggered 10.6% more failures in Defects4J, but there is a trade-off with false positives.*

F. RQ4: Does PAFM *affect test brittleness?*

While *PAFM* improves test suite effectiveness in terms of code coverage and number of failure-triggering tests, it might also increase test brittleness—that is, it might increase the likelihood of a test failing in the future, and thus increasing the effort for test maintainability. We therefore conducted an experiment on code evolution, using the programs version control systems, to answer the question of whether *PA* and *FM* affect the test maintainability of the generated tests. In particular, we applied the following methodology for each generated test suite:

1) Determine the commit of the program version for which the test suite was generated.
2) Determine and enumerate all future commits in the version conrol system that affect the source code of the program.
3) For each of the future commits, evolve the source code one commit at a time (i.e., incrementally apply the committed changes to the source code). We call such an incremental step an evolution step.
4) For each evolution step, determine whether the test suite still compiles and passes, and if so, determine the number of `AssumptionViolatedException`s.
5) If the test suite fails or passes all future commits, determine the number of successful evolution steps.

In addition to evaluating the generated test suites, we applied the same methodology to the developer-written test suites, provided by Defects4J. Note that to ensure comparability, we only consider the developer-written tests that are related to the classes for which the generated test suites were created. Figure 4 shows the code evolution results for *Base*, *PAFM*, and the developer-written test suites.

The results of this experiment show that, on average, *PAFM* test suites pass on fewer evolution steps than *Base* test suites, but the differences in terms of the number of evolution steps are small and not significant ($p = 0.63$). *PAFM* and *Base* test suites pass on all evolution steps for 7 program versions and fail after the same number of evolution steps for 162 program versions. *PAFM* test suites fail earlier for 131 and later for 51 program versions. Moreover, both *PAFM* and *Base* test suites

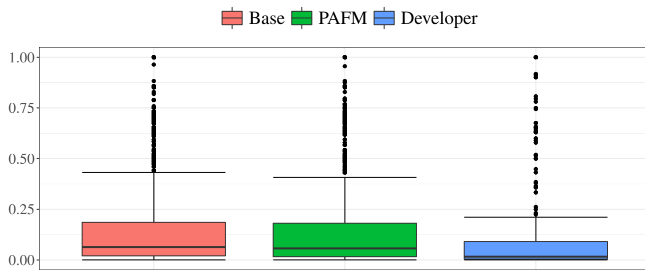

Fig. 4. Ratio of successful code evolution steps for *Base*, *PAFM*, and the developer-written test suites.

pass on more evolution steps than the developer-written test suites on average.

The question of whether these earlier failures are true or false positives cannot be answered without further manual analysis. However, interestingly we did not observe any `AssumptionViolatedExceptions` for the *PAFM* test suites throughout the entire evolution experiment. The likely explanation for this is that the amount of *PAFM* is very moderate, by design: Table IV shows statistics about the average amount of mocking and reflection related statements per test suite. On average, the test suites have 74.1 tests, but only 2.9 calls to private methods and 1 access of a private field. There are 5.7 mock objects on average per test suite, and 7.6 `doReturn` calls. To put these numbers into perspective, we calculated the same statistics on the test suites generated with Agitar One for a previous study [30] by counting invocations to Agitar's reflection and mocking helper functions. Table V shows that there is substantially more use of *PAFM*, which explains the high number of false positives observed for the Agitar test suites in that study.

> ***PAFM*** *test suites tend to fail slightly earlier than their* ***Base*** *counterparts, but the absence of* `AVE`*s suggests that the reason is increased code coverage and not* ***PAFM****.*

G. Threats to Validity

Internal: The techniques presented in this paper have all been implemented as part of the EvoSuite tool, which is used by many practitioners, but may still contain bugs. Because EvoSuite is based on randomized algorithms, each experiment was repeated several times, and the results have been evaluated with rigorous statistical methods. To ensure reproducibility [10], we released the implementation of all the techniques presented in this paper as open-source (LGPL license), and we made it available on a public repository[8]. Similarly, SF110 and Defects4J are freely available.

Construct: We used branch coverage, which is a common coverage criterion in the software testing literature, and we also considered fault detection based on real faults. However, it is hard to quantify the tradeoff between improved coverage/fault detection and the presence of false positives. If *PAFM* improves coverage by X%, but at the same time it increases the number of false positives by Y%, how to determine if the X% increase is worthwhile? As the effects of false positives on software development is a little investigated topic in the literature, more studies are necessary to shed light on this issue.

[8] www.github.com/EvoSuite/evosuite

TABLE IV
AVERAGE NUMBER OF *PA* AND *FM* USES IN EVOSUITE TEST SUITES.

Project	Tests	*PA*		*FM*	
		Methods	Fields	Objects	Calls
Chart	100.8	0.8	4.4	15.3	15.6
Closure	43.1	4.8	0.3	2.6	2.3
Lang	145.0	3.2	0.3	1.9	2.3
Math	61.8	1.5	1.5	6.2	9.4
Time	95.4	1.4	1.4	16.3	27.2
All	74.1	2.9	1.0	5.7	7.6

TABLE V
AVERAGE NUMBER OF *PA* AND *FM* USES IN AGITAR TEST SUITES [30].

Project	Tests	*PA*		*FM*	
		Methods	Fields	Objects	Calls
Chart	117.4	32.5	317.8	1.1	430.3
Closure	142.0	69.8	303.4	9.0	633.8
Lang	164.5	71.5	47.1	12.0	154.5
Math	90.5	13.6	62.0	0.7	117.4
Time	157.6	78.6	122.6	3.2	303.8
All	129.0	51.5	188.9	6.0	385.3

External: We used the SF110 corpus, which is a statistically valid random sample of 100 projects from SourceForge, plus its 10 most downloaded ones. Although we selected stratified samples (110 and 1,000 CUTs), this still led to a large variety among the employed classes on which EvoSuite was applied, which increases our confidence that the results generalise. Regarding fault detection, we based our experiments on 357 real faults from Defects4J. However, those faults come only from five different systems.

V. RELATED WORK

A. Private API Access

A study of Ma et al. [22] on the effects of private methods on the code coverage of unit tests showed that, while developers seem to be able to cope well, automated tools (in this study Randoop and EvoSuite) are negatively affected. In the same study, Ma et al. demonstrated that a customized version of Randoop, in which private methods were called through reflection, achieved higher code coverage than without this extension. However, they also acknowledged that accessing private methods, without knowing if their preconditions are valid, could have undesired side-effects.

In a study of the state-of-the-art in unit test generation [30], we observed that the commercial Agitar One makes use of reflection to access private members. However, we also observed a substantial number of false positives related to these tests, which motivated the integration as described in this paper, with the aim to minimize the number of false positives.

B. Functional Mocking

In the context of automated test generation, the most common use of mocking is to handle interactions of the CUT with its *environment*. Interactions could be, for example, reading/writing files, opening TCP connections to remote servers, etc. To have a full, deterministic control over the environment, an approach is to use *environment mocks*, which are classes that mimic the behavior of the environment. In the code under test, all calls to classes dealing with the environment can be replaced with mocks, which then can be configured directly in the tests. This approach was used to deal with the file system [5] and networking [6], and similar approaches have been used to deal with interactions with databases [31] and cloud services [35].

There have been some discussions about automatically generating more generic mock objects to improve test generation ([2], [15], [33]), and Islam and Csallner [18] presented a technique where mock objects were generated for classes that depend on interfaces with no concrete implementations. For those interfaces, mock objects were generated, where the return values of method calls on these mock objects were determined with a constraint solver. Promising results were achieved on 34 static methods, for a total of 320 lines of code. However, the presented technique only worked on the testing of static methods, and was limited by the type of inputs the constraint solver could handle (e.g, integers but not objects).

VI. Conclusions

In this paper, we introduced techniques to integrate private API access (*PA*) and functional mocking (*FM*) together (*PAFM*) into search-based unit test generation. A large empirical study showed that *PAFM* improves not only branch coverage (on average by +3.3%), but also fault detection on the real faults of the Defects4J data set.

The use of *PAFM* leads to *false positives*, i.e., tests that misleadingly fail although they detect no actual fault. We presented techniques to reduce these negative effects, showing that the number of false positives is low. A manual analysis nevertheless revealed that false positives may still happen, even with these counter measures.

The use of *PAFM* is thus a trade-off between increased coverage and increased risk of false positives. However, note that the problem of false positives does not go away by deactivating *PAFM*: We observed several cases of false positives independent of *PAFM*. As the role of false positives is a little investigated topic in automated test generation, future work will need to focus on studying those false positives and developing techniques to detect or avoid them.

All techniques discussed in this paper have been implemented as part of the EvoSuite test data generation tool. EvoSuite is open-source (LGPL license) and freely available to download. To learn more about EvoSuite and to access all artefacts of the experiments in this paper please visit our website at: `http://www.evosuite.org`.

Acknowledgments

This project has been funded by the EPSRC project "GREAT-EST" (EP/N023978/1), and by the National Research Fund, Luxembourg (FNR/P10/03).

References

[1] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)*, 36(6):742–762, 2010.

[2] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Schuler, P. Tonella, M. Harman, H. Muccini, W. Schulte, and T. Xie. Automock: Automated synthesis of a mock environment for test case generation. *Practical Software Testing: Tool Automation and Human Factors*, 2010.

[3] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[4] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[5] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–90, 2014.

[6] A. Arcuri, G. Fraser, and J. P. Galeotti. Generating TCP/UDP network data for automated unit test generation. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 155–165. ACM, 2015.

[7] L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: an evolutionary test approach for Java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.

[8] S. Bauersfeld, T. Vos, K. Lakhotia, S. Poulding, and N. Condori. Unit testing tool competition. In *International Workshop on Search-Based Software Testing (SBST)*, pages 414–420, 2013.

[9] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 55–66. ACM, 2014.

[10] C. Collberg and T. A. Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016.

[11] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[12] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[13] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[14] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[15] S. J. Galler, A. Maller, and F. Wotawa. Automatically extracting mock object behavior from design by contract™ specification for test data generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 43–50. ACM, 2010.

[16] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2012.

[17] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[18] M. Islam and C. Csallner. Generating test cases for programs that are coded against interfaces and annotations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):21, 2014.

[19] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 654–665, Hong Kong, November 18–20 2014.

[21] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: Program-analysis-guided random testing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[22] L. Ma, C. Zhang, B. Yu, and H. Sato. An empirical study on effects of code visibility on code coverage of software testing. In *Automation of Software Test (AST), 2015 IEEE/ACM 10th International Workshop on*, pages 80–84. IEEE, 2015.

[23] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *14th International Conference on Quality Software (QSIC)*, pages 127–132. IEEE, 2014.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[25] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[26] I. S. W. B. Prasetya. T3i: A Tool for Generating and Querying Test Suites for Java. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2015.

[27] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 2016.

[28] U. Rueda, R. Just, J. P. Galeotti, and T. E. Vos. Unit testing tool competition - round four. In *International Workshop on Search-Based Software Testing (SBST)*, 2016.

[29] A. Sakti, G. Pesant, and Y.-G. Gueheneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering (TSE)*, 2015.

[30] 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 (t). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, pages 201–211. IEEE, 2015.

[31] K. Taneja, Y. Zhang, and T. Xie. Moda: Automated test generation for database applications via mock objects. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 289–292. ACM, 2010.

[32] N. Tillmann and J. N. de Halleux. Pex — white box test generation for .NET. In *Int. Conference on Tests And Proofs (TAP)*, volume 4966 of *LNCS*, pages 134 – 253. Springer, 2008.

[33] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 365–368. IEEE, 2006.

[34] P. Tonella. Evolutionary testing of classes. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[35] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. De Halleux. Environmental modeling for automated cloud application testing. *IEEE Software*, 29(2):30–35, 2012.