MockDetector: A technique to identify mock objects created in unit tests

Qian Liang
Patrick Lam
q8liang@uwaterloo.ca
patrick.lam@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Unit testing is a widely used tool in modern software development processes. A well-known issue in writing tests is handling dependencies: creating usable objects for dependencies is often complicated. Developers must therefore often introduce mock objects to stand in for dependencies during testing.

We believe that the static analysis of test suites can enable developers to better characterize the behaviours of existing test suites, thus guiding further test suite analysis and manipulation. However, because mock objects are created using reflection, they confound existing static analysis techniques. At present, it is impossible to statically distinguish methods invoked on mock objects from methods invoked on real objects.

Researchers have started to build static analyses and developer tools for manipulating test cases. However, such tools currently cannot determine which dependencies' methods are actually tested, versus mock methods being called. As a specific example, removing confounding mock invocations from consideration as focal methods can improve the precision of analyses to detect focal methods under test, which is useful in itself and also a key prerequisite to further analysis of test cases.

In this paper, we introduce MockDetector, a technique to identify mock objects and pinpoint method invocations on mock objects. MockDetector locates common Java mock libraries' APIs for creating mock objects and propagates this information through test cases. Following our observations of tests in the wild, we have added special-case support for arrays and collections holding mock objects. We have built two implementations of MockDetector: a Soot-based imperative dataflow analysis implementation, as well as a Doop-based declarative analysis. On our suite of 8 open-source benchmarks, our imperative intraprocedural approach reported 2,095 invocations on mock objects, whereas our declarative interprocedural approach reported 5,315 invocations on mock objects (under context-insensitve base analyses), out of a total number of 63,017 method invocations in test suites; across benchmarks, mock invocations accounted for a range from 0.086% to 31.8% of the total invocations in tests.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Static Analysis, Dataflow Analysis, Declarative Program Analysis, Mock Objects, Unit Tests

ACM Reference Format:
Qian Liang and Patrick Lam. 2018. MockDetector: A technique to identify mock objects created in unit tests. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Mock objects [**?**] are commonly used in unit tests for object-oriented systems. They allow developers to test objects that rely on other objects, particularly ones that are hard to build (e.g. databases; or that come from different components).

While mock objects are an invaluable tool for developers, their use complicates the static analysis and manipulation of test case source code. Such static analyses can help IDEs provide better support to test case writers; enable better static estimation of test coverage (avoiding mocks); and detect focal methods in test cases. While researchers have proposed techniques for automatically generating mocks [**? ?**], our goal here is the opposite: we detect mocks that already exist in test cases.

Ghafari et al discussed the notion of a focal method [**?**] for a test case—the method whose behaviour is being tested—and presented a heuristic for determining focal methods. By definition, the focal method's receiver object cannot be a mock object. Ruling out mock invocations can thus improve the accuracy of focal method detection and enable better understanding of a test case's behaviour.

Mock objects are difficult to analyze statically because, at the bytecode level, a call to a mock object statically resembles a call to the real object (as intended by the designers of mock libraries). A naive static analysis attempting to be sound would have to include all of the possible behaviours of the actual object (rather than the mock) when analyzing such code. Such potential but unrealizable behaviours obscure the true behaviour of the test case.

We have designed a static analysis, MockDetector, which identifies mock objects in JUnit[1] test cases. It starts from a list of mock object creation sites (our analyses include hardcoded APIs for common mocking libraries EasyMock[2], Mockito[3], and PowerMock[4]). It then propagates mockness through the test and identifies invocation sites as (possibly) mock. Given this analysis result, a subsequent analysis can ask whether a given variable in a test case contains a mock or not, and whether a given invocation site is a call to a mock object or not. We have evaluated MockDetector on a suite of 8 benchmarks plus a microbenchmark. We have cross-checked results across the two implementations and manually inspected the results on our microbenchmark, to ensure that the results are as expected.

Taking a broader view, we believe that helper static analyses like MockDetector can aid in the development of more useful static analyses. These analyses can encode useful domain properties; for instance, in our case, properties of test cases. By taking a domain-specific approach, analyses can extract useful facts about programs that would otherwise be difficult to establish.

We make the following contributions in this paper:

- We designed and implemented two variants of a static mock detection algorithm, one as a dataflow analysis implemented imperatively (using Soot) and the other declaratively (using Doop).
- We characterize our benchmark suite (8 open-source benchmarks, 383kLOC, 184 kLOC tests) with respect to their use of mock objects, finding that 1084 out of 6310 unit tests use intraprocedurally-detectable mocks, and that there are a total of 2095 method invocations on mocks.
- We present potential applications of mock analysis: detecting focal methods, helping to understand test cases, automated refactoring, and API usage extraction.

We will make all of our artifacts and data publicly available. For the purposes of the not-very-well-defined required replication package for this submission, see: https://anonymous.4open.science/r/MockAbstraction-B0F0/ or https://figshare.com/s/285073ed5bf5b1e5e7a9.

2 MOTIVATING EXAMPLE

MockDetector finds variables containing mock objects, along with invocations on mock objects, in unit tests. It does so by identifying invocations on variables which have been assigned an object flowing from a mock creation site either using a forward dataflow may-analysis (Soot) or by solving declarative constraints (Doop).

To motivate our work, consider Listing 1, which presents a unit test from the Maven project. Line 4 calls *getRequest()*, invoking it on mock object `session`. Line 7 then calls *getToolchainsForType()*, which happens to be the focal method whose behaviour is being tested here. At the bytecode level, the two calls are indistinguishable with respect to mockness; the static call graph contains a call to the actual method in both cases, so that current static analysis tools cannot easily tell the difference between the method invocation on a mock object on line 4 and the method invocation on a real object on line 7. This uncertainty about mockness would confound a naive static analysis that attempts to identify focal methods. For instance, Ghafari et al [**?**]'s heuristic would fail on this test, as it returns the last mutator method in the object under test, and the focal method here is an accessor. Section 6 discusses applications of mock analysis to finding focal methods in more detail. As another application, mockness information can help an IDE provide better suggestions.

```
@Test public void testMisconfiguredToolchain() throws Exception {
MavenSession session = mock( MavenSession.class );
MavenExecutionRequest req = new DefaultMavenExecutionRequest();
when( session.getRequest() ).thenReturn( req );

ToolchainPrivate[] basics =
toolchainManager.getToolchainsForType("basic", session);

assertEquals( 0, basics.length );
}
```

Listing 1: This code snippet illustrates an example from maven-core, where calls to both the focal method `getToolchainsForType()` and to mock `session`'s `getRequest()` method occur in test *testMisconfiguredToolchain()*.

Basic dataflow analysis. The dataflow analysis-based MockDetector implementation keeps an abstraction mapping values (local variables or field references) in Soot's Jimple intermediate representation (IR) [**?**] to `MockStatus`, which holds three bits monitoring each value's status (a mock, a mock-containing array, or a mock-containing collection).

```
//        mock: ✗     mockAPI: ✗
Object object1 = new Object();

// mock: ✗
object1.foo();

//        mock: ✓     mockAPI: ✓
Object object2 = mock(Object.class);

// mock: ✓
object2.foo();
```

Figure 1: Our static analysis propagates mockness from sources (e.g. `mock(Object.class)`) to invocations.

```
java.lang.Object $r1, r2;

$r1 = new java.lang.Object;
specialinvoke $r1.<java.lang.Object: void <init>()>();
virtualinvoke $r1.<java.lang.Object: void foo()>();
r2 = staticinvoke <org.mockito.Mockito: java.lang.Object mock(java.lang.Class)> (c
virtualinvoke r2.<java.lang.Object: void foo()>();
```

Listing 2: Jimple Intermediate Representation for the code in Figure 1.

Figure 1 shows how our dataflow analysis works; Listing 2 shows the Jimple IR of the code in Figure 1. At the top of the IR, we begin with an empty abstraction (no mapping for any values, equivalent to all bits false everywhere) before line 3. For the creation of `$r1` on lines 3 and 4, since the call to the no-arg `<init>` constructor is not one of our hardcoded mock APIs, our analysis does not declare `$r1` to be a mock object. Thus, invocation `object1.foo()` on line

[1] https://junit.org
[2] https://easymock.org/
[3] https://site.mockito.org/
[4] https://github.com/powermock/powermock

5 in Figure 1 is not known to be a mock invocation. Tying back to our focal methods application, we would not exclude the call to `foo()` from being a possible focal method.

On the other hand, our imperative analysis sees mock creation API `<org.mockito.Mockito: java.lang.Object mock(java.lang.Class)>` on line 6 in the Jimple IR. It thus adds a mapping from local variable `r2` to a new `MockStatus` with mock bit set to true. Because `r2` has a mapping in the abstraction with mock bit set, MockDetector will deduce that the call on line 7 is a mock invocation. This implies that the call to method *foo()* on line 11 in Figure 1 cannot be a focal method.

Basic declarative analysis. Again referring to IR Listing 2, this time we ask whether the call on IR line 7 satisfies predicate `isMockInvocation` (facts Listing 3, line 2), which we define to hold the analysis result (all mock invocation sites in the program). It does, because of facts lines 3–6: IR line 7 contains a virtual method invocation, and receiver `r2` for the invocation on that line satisfies our predicate `isMockVar`, which holds all mock-containing variables in the program (Section 4.1 provides more details). Predicate `isMockVar` holds because of lines 7–8: `r2` satisfies `isMockVar` because IR line 6 assigns `r2` the return value from mock source method `createMock` (facts line 7), and the call to `createMock` satisfies predicate `callsMockSource` (facts line 8), which requires that the call destination `createMock` be enumerated as a constant in our 1-ary relation `MockSourceMethod` (facts line 9), and that there be a call graph edge between the method invocation at line 6 and the mock source method (facts line 10).

```
isMockInvocation(<Object:void foo()>/test/0,
<Object:void foo()>, test, _, r2).
|VirtualMethodInvocation(<Object:void foo()>/test/0,
|                        <Object:void foo()>, test).
|VMI_Base(<Object:void foo()>/test/0, r2).
|isMockVar(r2).
|-AssignReturnValue(<Mockito:Object mock(Class)>/test/0, r2).
|-callsMockSource(<Mockito:Object mock(Class)>/test/0).
|MockSourceMethod(<Mockito:Object mock(Class)>).
|CallGraphEdge(_,<Mockito:Object mock(Class)>/test/0, _,
|              <Mockito: Object mock(Class)>).
```

Listing 3: Facts about invocation `r2.foo()` in method `test`.

Extensions: arrays and collections. While designing MockDetector, we observed that developers store mock objects in arrays and collections. Listing 4 presents method *setUp()* in class `NodeListIteratorTest` from commons-collections-4.4. Line 9 puts mock `Node` objects in array field `nodes`, later used in tests. When the dataflow analysis encounters an assignment statement containing an array read or write, it first looks for values (local variables or field reference sources) on the opposite side of the assignment (the statement's destination or source) in the IR. It then checks whether any of these locals or fields are mocks. If so, then the tool would mark the local or field reference as an array mock—it propagates mockness to the array container.

```
// Node array to be filled with mock Node instances
private Node[] nodes;
@Test protected void setUp() throws Exception {
// create mock Node Instances and fill Node[] to be used by tests
final Node node1 = createMock(Element.class);
final Node node2 = createMock(Element.class);
final Node node3 = createMock(Text.class);
final Node node4 = createMock(Element.class);
nodes = new Node[] {node1, node2, node3, node4};
// ...
}
```

Listing 4: This example illustrates a field array container holding mock objects from *setUp()* in `NodeListIteratorTest`.

Figure 2 illustrates a mock-containing array, and Listing 5 shows the corresponding IR. Our analysis reaches the mock API call on lines 4–6, where it records that `$r2` is a mock object—it creates a MockStatus with mock bit set for `$r2`. The tool would then handle the cast expression assigning to `r1` on line 7, giving it the same MockStatus as `$r2`. When the analysis reaches line 9, it finds an array reference on the left hand side, along with `r1` stored in the array on the right hand side of the assignment statement. At that point, it has a MockStatus associated with `r1`, with the mock bit turned on. It can now deduce that `$r3` on the LHS is an array container which may hold a mock. Therefore, MockDetector's imperative analysis associates `$r3` with a MockStatus with mock-containing array bit set. Collections work similarly—we hard-code methods from the Java Collections API.

```
java.lang.Object r1, $r2;
java.lang.Object[] $r3;

$r2 = staticinvoke <org.easymock.EasyMock:
java.lang.Object createMock(java.lang.Class)>
(class "java.lang.Object;");
r1 = (java.lang.Object) $r2;
$r3 = newarray (java.lang.Object)[1];
$r3[0] = r1;
```

Listing 5: Jimple Intermediate Representation for the array in Figure 2.

```
//       mock: ✓    mockAPI: ✓
Object object1 = createMock(Object.class);

// arrayMock: ✓ ⇐ array-write       mock: ✓
objects   = new Object[]     { object1 };
```

Figure 2: Our static analysis also finds array mocks.

The declarative analysis uses analogous reasoning, but again uses relations instead of bits in the MockStatus abstraction.

3 A SURVEY OF APPLICATIONS

Our work supports test-to-code traceability. We now sketch several applications of our work to this important problem, inspired by Ghafari et al [**?**] but specializing to our analysis.

Test case comprehension. xUnit tests are snippets of arbitrary code. In unpublished research, we have established that most tests are simple straight-line code (although there are still nontrivial incidences of conditionals and loops). However, this code by necessity interacts with the system under test in potentially complicated ways. Hence, understanding what a test case is doing can be difficult. Ghafari et al described experiments where developers are asked to identify the focal method, and finds that this is surprisingly difficult; furthermore, Daka et al [**?**] state that reasoning about tests is difficult and propose a model that enables the automatic generation of tests which are especially readable.

Our mock analysis helps segregate tests into parts that are mock-related and parts that are not mock-related. This is directly applicable to identifying focal methods, as we'll discuss in Section 6. Also, it is fairly obvious to a human reader that the part of a test that is calling mock library methods such as `thenReturns()` is setting up mock expectations, but we've seen cases where this is less obvious. In particular, when recording behaviours with EasyMock, the developer simply calls methods on the mock object, e.g. `mock.documentAdded("document");`.

Knowing what is a mock can help developers debugging test case failures get into the right mindset, in two ways: 1) when looking at mock calls, they can conclude that these mock calls are not directly causing the test failures; but also, 2) if the mock calls are now returning incorrect values (perhaps due to program evolution), then it is likely appropriate to update the recorded values. Similar considerations apply for automatic debugging and code repair.

Code recommendation. To amplify the previous point, a key application of test-to-code traceability is that when the code is updated, related tests may also need to be updated. Integrated Development Environments can find all tests referring to a particular fragment of code. Mock analysis can augment the information available to the developer (or maintainer) by giving them additional information about whether they are updating tests that depend on the changed code as a mock or whether the tests are in fact testing the changed code itself.

Automated refactoring/code generation. We originally formulated the mock analysis problem as a side problem which needed to be solved in the service of a deeper problem, which is still ongoing work: automatically generating certain useful tests. In our context, we needed to know which call was to the focal method of a test—we wanted to create additional tests based on those focal methods, but not based on mock objects.

Extracting API usage examples. Finally, mock objects can serve as even more concise API usage examples for the objects being mocked (compared to the test as a whole); that information could be extracted from the calls to the mock library. After all, mocks record invocations to methods the objects being mocked, and the expected behaviours of those objects in response to these invocations.

Our applications show that the treatment of mocks and non-mocks when editing test cases (manually or automatically) should be different. Furthermore, especially for automatic treatments of test cases, a static analysis determining which invocations are mock invocations is key to effective treatment.

4 TECHNIQUE

We implemented two complementary ways of statically computing mock information: a declarative implementation (using Doop [**?**]), along with an imperative implementation of a dataflow analysis (using Soot [**?**]). Although the reason that two implementations exist is unimportant at this stage, the result is consequential: the existence of these two implementations is a rare opportunity to cross-check static analysis results. We thus carefully compared our results to make sure that they matched.

4.1 Declarative Doop Implementation

We next describe the declarative Doop-based technique that MockDetector uses. We implemented this technique by writing Datalog rules. We propagate mockness from known mock sources, through the statements in the intermediate representation (IR), to potential mock invocation sites.

We chose to implement a may-analysis rather than a must-analysis for two reasons: 1) we did not observe any cases where a value was assigned a mock on one branch and a real object on the other branch; 2) implementing a must-analysis would not help heuristics to find focal methods, as a must-analysis would rule out fewer mock invocations.

From our perspective, including (context-insensitive) interprocedural support is almost trivial; we only need to add two rules. The Doop analysis also supports arrays, containers, and fields. Although declarative analyses are easy to define than their imperative counterparts, they are subject to the feature interaction problem: for instance, they still need to specifically handle the case where an array-typed field is assigned from a mock-containing array local.

The core of the implementation starts by declaring facts for 9 mock source methods manually gleaned from the mock libraries' documentation, as specified through method signatures (e.g. `<org.mockito.Mocki java.lang.Object mock(java.lang.Class)>`.) It then declares that a variable `v` satisfies `isMockVar(v)` if it is assigned from the return value of a mock source, or otherwise traverses the program's interprocedural control-flow graph, through assignments, which may possibly flow through fields, collections, or arrays. Finally, an invocation site is a mock invocation if the receiver object `v` satisfies `isMockVar(v)`.

```
// v = mock()
isMockVar(v) :-
AssignReturnValue(mi, v),
callsMockSource(mi).
// v = (type) from
isMockVar(v) :- isMockVar(from),
AssignCast(_/* type */, from, v, _/* inmethod */).
// v = v1
isMockVar(v) :- isMockVar(v1),
AssignLocal(v1, v, _).
```

The predicates `AssignReturnValue`, `AssignCast`, and `AssignLocal` are provided by Doop, and resemble Java bytecode instructions. Unlike Java bytecode, however, their arguments are explicit, and not stack-based. For instance, `AssignLocal(?from:Var, ?to:Var, ?inmethod:Method)` denotes an assignment statement copying from `?from` to `?to` in method `?inmethod`. (It is Datalog convention to prefix parameters with ?s).

We designed the analysis in a modular fashion, such that the interprocedural, collections, arrays, and fields support can all be disabled through the use of `#ifdefs`, which can be specified on the Doop command-line.

```
// v = callee(), where callee's return var is mock
isInterprocMockVar(v) :-
AssignReturnValue(mi, v),
mainAnalysis.CallGraphEdge(_, mi, _, callee),
ReturnVar(v_callee, callee),
```

```
isMockVar(v_callee).

// callee(v) results in formal param of callee being mock
isInterprocMockVar(v_callee) :- isMockVar(v),
ActualParam(n, mi, v),
FormalParam(n, callee, v_callee),
mainAnalysis.CallGraphEdge(_, mi, _, callee),
Method_DeclaringType(callee, callee_class),
ApplicationClass(callee_class).
```

We use Doop-provided call graph edges (relation `mainAnalysis.CallGraphEdge`) between the method invocation `mi` and its callee `callee`; the first rule propagates information from callees back to their callers, while the second rule propagates information from callers to callees through parameters. We restrict our analysis to so-called "application classes", excluding in particular the Java standard library. We chose to run our context-insensitive analysis on top of Doop's context-insensitive call graph. Mirroring Doop itself, it would also be possible to add context sensitivity to our analysis, potentially reducing the number of false positives.

Arrays and Containers. Consistent with our analysis being a may-analysis, we define predicate `isArrayLocalThatContainsMocks` to record local variables pointing to arrays that may contain mock objects. This predicate is true whenever the program under analysis stores a mock variable into an array; we also transfer array-mockness through assignments and casts. When a local variable `v` is read from a mock-containing array `c`, then `v` is marked as a mock variable, as seen in the first rule below. An store of a mock variable `mv` into an array `c` causes that array to be marked as `isArrayLocalThatContainsMocks`. Note that these predicates are mutually-recursive. Similarly with containers. We also handle `Collection.toArray` by propagating mockness from the collection to the array.

```
// v = c[idx]
isMockVar(v) :-
isArrayLocalThatContainsMocks(c),
LoadArrayIndex(c, v, _ /* idx */).

// c[idx] = mv
isArrayLocalThatContainsMocks(c) :-
StoreArrayIndex(mv, c, _ /* idx */),
isMockVar(mv).
```

We treat collections analogously. However, while there is one API for arrays—bytecode array load and store instructions—Java's Collections APIs include, by our count, 60 relevant methods. We support iterators, collection copies via constructors, and add-all methods. We use our classification of collection methods to identify collection reads and writes and handle them as we do array reads and writes.

Fields. Apart from the obvious rule stating that a field which is assigned from a mock satisfies `fieldContainsMock`, we also label fields that have the `org.mockito.Mock` or `org.easyMock.Mock` annotations as mock-containing. We declare that a given field *signature* may contain a mock, i.e. the field with a given signature belonging to all objects of a given type. We also support containers stored in fields.

Arrays and Fields. We also support not just array locals but also array fields. That is, when an array-typed field is assigned from a mock-containing array local, then it is also a mock. And when an array-typed local variable is assigned from a mock-containing array field, then that array local is a mock-containing array.

4.2 Imperative Soot Implementation

We now briefly describe the Soot-based imperative dataflow analysis to find mocks. Like the declarative analysis, we track information from the creation sites through the control-flow graph. Here, we implement a forward dataflow may-analysis—an object is declared a mock if there exists some execution path where it may receive a mock value. Our implementation also understands containers (arrays and collections), and tracks whether containers hold any mock objects. The abstraction marks all contents of collections as potential mocks if it observes any mock object being put into the array or container. It maps values in the IR to three bits: one for the value being a mock, one for it being an array containing a mock, and one for it being a collection containing a mock. Space limitations prevent us from including full details of this implementation, but we will make all implementations available, and the analysis is fully described in the first author's master's thesis[5].

Interprocedural support. The Heros framework[**?**] implements IFDS/IDE for program analysis frameworks including Soot. With some effort, it would be possible to rewrite our mock analysis with Heros; however, this would be a more involved process than in the declarative case, where we simply added two rules to obtain the interprocedural analysis. In particular, Heros uses a different API in its implementation than Soot. Conceptually, though, it should be no harder to implement an interprocedural Heros analysis than an intraprocedural Soot dataflow analysis.

4.2.1 Pre-Analyses for Field Mocks Defined in Constructors and Before Methods. A number of our benchmarks define fields as mock objects via EasyMock or Mockito `@Mock` annotations, or initialize these fields in the `<init>` constructor or `@Before` methods (*setUp()* in JUnit 3), which test runners will execute before any test methods from those classes. These mock field or mock-containing container fields are then used in tests. In the Soot implementation, we use two pre-analyses before running the main analysis, under the assumption that fields are rarely mutated in the test cases (and that it is incorrect to do so). We have validated our assumption on benchmarks. An empirical analysis of our benchmarks shows that fewer than 0.3% of fields (29/9352) are mutated in tests.

MockDetector retrieves all fields in all test classes, and marks fields annotated `@org.mockito.Mock` or `@org.easymock.Mock` as mocks. Also, Listing 6 depicts an example where instance fields are initialized using field initializers. Java copies such initializers into all constructors (`<init>`). To detect such mock-containing fields, our first pre-analysis applies the dataflow analysis to all constructors in the test classes prior to running the main analysis,

[5] Citation temporarily omitted due to double-blind review.

using the same logic that we use to detect mock objects or mock-containing containers. The second pre-analysis handles field mocks defined in `@Before` methods just like the first pre-analysis handled constructors.

```
private GenerationConfig config = mock(GenerationConfig.class);
private RuleFactory ruleFactory = mock(RuleFactory.class);
```

Listing 6: Field mocks defined by field initializations from `TypeRuleTest` in jsonschema2pojo.

4.3 Common Infrastructure

We have parameterized our technique with respect to mocking libraries and have instantiated it with respect to the popular Java libraries Mockito, EasyMock, and PowerMock. We also support JUnit 3 and 4+.

Both JUnit and mocking libraries rely heavily on reflection, and would normally pose problems for static analyses: the set of reachable test methods is enumerated at run-time, and mock libraries create mock objects reflectively. Fortunately, their use of reflection is stylized, and we have designed our analyses to soundly handle these libraries.

JUnit and Driver Generation. JUnit tests are simply methods that developers write in test classes, appropriately annotated (in JUnit 3 by method name starting with "test", in 4+ by a `@Test` annotation). A JUnit test runner uses reflection to find tests. Out of the box, static analysis engines do not see tests as reachable code.

Thus, to enable static analysis over a benchmark's test suite, our tool uses Soot to generate a driver class for each Java sub-package of the suite (e.g. `org.apache.ibatis.executor.statement`). In each of these sub-package driver classes, our tool creates a *runall()* method, which invokes all methods within the sub-package that JUnit considers tests, as well as non-constructor test cases, all surrounded by calls to class-level init/setup and teardown methods. Concrete test methods are particularly easy to call from a driver, as they are specified to have no parameters and are not supposed to rely on any particular ordering. Our tool then creates a RootDriver class at the root package level, which invokes the *runall()* method in each sub-package driver class, along with the `@Test`/`@Before`/`@After` methods found in classes located at the root. The drivers that we generate also contain code to catch all checked exceptions declared to be thrown by the unit tests. Both our Soot and Doop implementations use the generated driver classes.

All static frameworks must somehow approximate the set of entry points as appropriate for their targets. For instance, the Wala framework [**?**] also creates synthetic entry points, but it does this to perform pointer analysis on a program's main code rather than to enumerate the program's test cases.

Intraprocedural Analysis. The Soot analysis is intraprocedural and the Doop analysis has an intraprocedural version. Intraprocedurally, we make the unsound (but reasonable for our anticipated use case) assumption that mockness can be dropped between callers and callees: at method entry points, no parameters are assumed to be mocks, and at method returns, the returned object is never a mock. Doop's interprocedural version drops this assumption and instead context-insensitively propagates information from callers to callees and back; we discuss the results of doing so in Section 5.

Call graphs are useful to our intraprocedural analysis because they help identify calls to mock source methods (that we identify explicitly). We assume that developers do not call inherited versions of mock creation sites (e.g. a wrapper of the mock source method), but if we have a call graph and observe calls to such methods in Doop, we include them.

4.4 Mock Analysis is not just Reaching Definitions

While our mock analysis is similar to reaching definitions, we point out some important differences which motivated our decision to design and implement a bespoke analysis. Even the base analysis requires slightly more than reaching definitions, and then we also support fields, containers and arrays.

For the base analysis, querying reaching definition information almost works. However, mock analysis also requires propagating information through casts, as seen in the `AssignCast` rule in the declarative analysis; at statement `s: x = (X)y;`, then `x` would be a mock iff `y` is. Statement `s` would be a definition for `x`, and a further query would be needed to know whether `y` was a mock or not.

Additionally, the field and array analyses each detected additional mocks in half of our benchmarks (Section 5). Fields do not fit the reaching definition paradigm and would require additional support. Arrays would be possible but not obvious.

5 EVALUATION

We now quantitatively characterize the performance of our mock analysis implementations on a suite of popular open-source applications. We also provide evidence about the importance of mock invocations to test suites. Section 6 discusses an application of our analysis to focal method detection.

We evaluated MockDetector on 8 open-source benchmarks, along with a micro-benchmark we developed to test our tool. We ran our experiments on a 32-core Intel(R) Xeon(R) CPU E5-4620 v2 at 2.60GHz with 128GB of RAM running Ubuntu 16.04.7 LTS.

Table 1 presents summary information about our benchmarks and run-times—the LOC and Soot and Doop analysis run-times for each benchmark. The 9 benchmarks include over 383 kLOC, with 184 kLOC in the test suites, per SLOCCount[6]. Our benchmarks are from different domains and created by different groups of developers. The Soot total time is the amount of time that it takes for Soot to analyze the benchmark and test suite in whole-program mode, including our analyses. The Soot intraprocedural analysis time is the sum of run-times for the main analysis plus two pre-analyses (Section 4.2). The reported Doop run-time is from the context-insensitive analysis, while the Doop analysis time for intraprocedural and interprocedural mock invocation analyses are for running the analysis alone based on recorded facts from the benchmark. The total Doop run-time is much slower than the total Soot run-time because Doop always computes a callgraph—an expensive operation. The Doop analysis-only time is also slower than the Soot time; Doop computes a solution over the entire program, while Soot works one method at a time.

[6] https://dwheeler.com/sloccount/

Table 1: Our suite of 8 open-source benchmarks (8000–117000 LOC) plus our microbenchmark. Soot and Doop analysis runtimes.

Benchmark	Total LOC	Test LOC	Soot intraproc total (s)	Doop intraproc total (s)	Soot intraproc mock analysis (s)	Doop intraproc mock analysis (s)	Doop interproc mock analysis (s)
bootique-2.0.B1-bootique	15530	8595	58	2810	0.276	19.93	24.90
commons-collections4-4.4	65273	36318	114	694	0.386	14.20	16.64
flink-core-1.13.0-rc1	117310	49730	341	1847	0.415	27.21	62.12
jsonschema2pojo-core-1.1.1	8233	2885	313	1005	0.282	29.33	41.05
maven-core-3.8.1	38866	11104	183	588	0.276	19.49	23.42
micro-benchmark	954	883	47	387	0.130	11.73	12.92
mybatis-3.5.6	68268	46334	500	4477	0.662	59.83	192.16
quartz-core-2.3.1	35355	8423	155	736	0.231	21.06	21.92
vraptor-core-3.5.5	34244	20133	371	1469	0.455	34.95	149.38
Total	384033	184405	2082	14013	3.123	237.73	544.51

Table 2: Counts of Test-Related (Test/Before/After) methods in public concrete test classes, along with counts of mocks, mock-containing arrays, and mock-containing collections, reported by Soot intraprocedural analysis.

Benchmark	# of Test-Related Methods	# of Test-Related Methods with mocks (intra)	# of Test-Related Methods with mock-containing arrays (intra)	# of Test-Related Methods with mock-containing collections (intra)
bootique-2.0.B1-bootique	420	32	7	0
commons-collections4-4.4	1152	3	1	1
flink-core-1.13.0-rc1	1091	4	0	0
jsonschema2pojo-core-1.1.1	145	76	1	0
maven-core-3.8.1	337	24	0	0
micro-benchmark	59	43	7	25
mybatis-3.5.6	1769	330	3	0
quartz-core-2.3.1	218	7	0	0
vraptor-core-3.5.5	1119	565	15	0
Total	6310	1084	34	26

Table 3: Comparison of Number of InstanceInvokeExprs on Mock objects analyzed by Soot and Doop, and Total Number of InstanceInvokeExprs, in each benchmark's test suite.

Benchmark	Total Number of Invocations	Mock Invokes intraproc (Soot)	Context-insensitive, intraproc (Doop)	Context-insensitive, interproc (Doop)
bootique-2.0.B1-bootique	3366	99	99	122
commons-collections4-4.4	12753	11	3	23
flink-core-1.13.0-rc1	11923	40	40	1389
jsonschema2pojo-core-1.1.1	1896	276	282	604
maven-core-3.8.1	4072	23	23	39
microbenchmark	471	108	123	132
mybatis-3.5.6	19232	575	577	1345
quartz-core-2.3.1	3436	21	21	31
vraptor-core-3.5.51	5868	942	962	1630
Total	63017	2095	2130	5315

We next investigated mock prevalence. Table 2 presents the number of test-related (Test/Before/After) methods which contain local variables or which access fields that are mocks, mock-containing arrays, or mock-containing collections, as reported by our Soot-based intraprocedural analysis.

Table 4: Counts of Field Mock Objects defined via `@Mock` annotation, the constructors, and in `@Before` methods, in each benchmark's test suite.

Benchmark	# of Annotated Field Mock Objects	# of Field Mock Objects defined in the <init> constructor	# of Field Mock Objects defined in @Before methods
bootique-2.0.B1-bootique	0	0	8
commons-collections4-4.4	0	0	0
flink-core-1.13.0-rc1	0	0	0
jsonschema2pojo-core-1.1.1	26	126	0
maven-core-3.8.1	7	0	1
micro-benchmark	2	0	29
mybatis-3.5.6	41	0	0
quartz-core-2.3.1	0	0	0
vraptor-core-3.5.5	263	128	83

Table 5: Comparison of Statement Coverage and Branch Coverage with all test cases, and with test cases excluding intraprocedural mock invocations.

Benchmark	Statement Coverage		Branch Coverage	
	All Test Cases	Test Cases without Intraproc Mocks	All Test Cases	Test Cases without Intraproc Mocks
jsonschema2pojo-core-1.1.1	37%	24%	33%	19%
maven-core-3.8.1	48%	48%	39%	38%
mybatis-3.5.6	85%	81%	82%	76%
vraptor-core-3.5.5	87%	59%	81%	56%

Finding 1: Across the 8 benchmarks, test-related methods containing local/field mocks or mock-containing containers accounted for 0.35% to 51.8% of the total number of test-related methods found in public concrete test classes.

Of our benchmarks, 3 show very little mock use, while 2 show extensive use, with the remainder in between. Benchmarks VRAPTOR-CORE and JSONSCHEMA2POJO-CORE have more than half of their test-related methods containing mock objects (and mock-containing arrays); in both of these, most field mocks are created via annotations and reused in multiple test cases in the same class. The difference in mock usage reflects their different philosophies and constraints regarding the creation and usage of mock objects in tests.

Table 3 is the core result about our mock analyses. It presents the detected number of method invocations on mocks. We include numbers from the imperative intraprocedural Soot implementation, as well as intraprocedural and interprocedural versions of the declarative Doop implementation.

Finding 2: Our intraprocedural analysis finds that method calls on mock objects account for 0.086% to 16.4% of the total number of calls.

In Section 4.3 we discussed the implementation of our intraprocedural and interprocedural analyses. We can now discuss the effects of these implementation choices on the experimental results. Recall that we chose, unsoundly, to not propagate any information across method calls in the intraprocedural analysis. Thus, the intraproc columns in Table 3 show smaller numbers than the interproc columns, as expected. The minor difference in VRAPTOR-CORE is due to one method (which ought to be present) not showing up in Doop's context-insensitive callgraph. Note also that there is a sometimes drastic increase from the intraprocedural to the interprocedural result, e.g. from 40 to 1300 for FLINK. This is because mocks can propagate from tests to the methods that they call and all around the main program code. It would be desirable to be able to differentiate test helper methods, which we want to propagate mocks to, from methods in the main program, which we generally do not want to propagate mocks to; however, Doop treats test and main code identically.

Finding 3: Interprocedural analysis finds from 1.07× to 34× more mock invocations than intraprocedural analysis, but that number includes potential mock calls in the main program.

We have successfully executed our Doop analysis with different base (pointer) analyses. We reported numbers for the context-insensitive base analysis here as it matches our own mock analysis. (It would also be possible, with much more effort, to adapt our analysis to carry around context.)

Validation. Our mock analysis could be subject to false positives (reported mocks that aren't) or false negatives (unreported mocks). As mentioned previously, we've cross-checked the results between the two implementations, reducing the likelihood of false positives. We observed false negatives in the Soot implementation, due to missing support for array-/collection-related functions. Both implementations are arguably subject to false negatives when subjects build their own mocks; those are more likely to be stubs or fakes than true mocks [**?**]. Other types of false negatives are hard to detect.

Field Mocks Results. We perform an evaluation on the necessity of our pre-analyses finding field mocks. Table 4 displays the number of field mock objects that are defined via `@Mock` annotations, in the constructors, and in the `@Before`/`setUp()` methods, respectively.

We focus on the 5 benchmarks that have defined field mock objects. Among these benchmarks, JSONSCHEMA2POJO-CORE, MYBATIS, and VRAPTOR-CORE have a high number (565) or a high percentage (over 50%) of test-related methods containing mock objects, with many intraprocedural mock invokes. From the results in Table 4, we can tell these benchmarks also define field mock objects instead of repetitively creating the same mock objects in each test, reducing the need for code maintenance. In addition, although BOOTIQUE and MAVEN-CORE have lower number of tests using mock objects, these benchmarks still define field mocks.

Finding 4: More than half (5/8) of our benchmarks define field mock objects in their tests.

We can deduce that our pre-analysis for field mocks described in Section 4.2.1 is required for analyzing mocks in test suites.

Application: Mocks are important. Since our mock analysis identifies mock objects, we are in a position to empirically evaluate the importance of mock objects in our benchmarks' test suites. One measure of their importance is how much they contribute to code coverage. Using jacoco with the surefire Maven plugin, we measured branch and statement coverage for 4 of our benchmarks, both with and without test cases that contain mock invocations. We excluded mocks by generating custom Maven test execution commandlines.

Table 5 presents our results. We can see that for benchmarks JSONSCHEMA2POJO-CORE and VRAPTOR-CORE, which have about 15% mock invocations, coverage drops by over 13 to 28 percentage points when excluding mocks. The fact that a correlation exists may be unsurprising, but the magnitude of the 28 point gap for VRAPTOR-CORE is striking, and points out that the mock objects are pulling more than their weight in terms of coverage. Note that our analysis, or a dynamic version thereof, is required to collect these numbers.

Finding 5: Test suites use mocks to increase statement coverage by 13–28% versus not using mocks.

6 SPECIFIC APPLICATION: FINDING FOCAL METHODS

In Section 3 we surveyed several applications of mock analysis. One useful application of mock detection is to improve the detection of focal methods in test cases, or as Ghafari et al [**?**] call them, focal methods under test (F-MUTs). In this section, we delve into one specific application of mock analysis, discussing some approaches to finding focal methods, and describing how our analysis can facilitate this search.

Ghafari's approach is the one that is most amenable to our work. Their approach is applicable to tests for classes that implement stateful objects, and assume that each test has at least one focal method, seen as a call to a mutator method for the class under test. They declare the last such call to be the focal method. However, their approach can only work on tests of mutator methods. Purely-functional methods, for instance, are beyond the scope of their approach.

Another approach is name-based, as seen in the methods2test dataset [**?**]. This approach uses two heuristics: 1) the name of the test matches the name of the focal method; or, 2) the test calls a unique method in the focal class. Rompaey and Demeyer [**?**] explore the name-based approach along with simply using the last method called—not necessarily a mutator—as the class containing the focal method. Romaey and Demeyer also explore other heuristics, as we discuss in Section 7. Ghafari et al point out that these heuristics have significant limitations, e.g. when a test case has sub-scenarios.

Precision and Recall. We further compare the static analysis and name-based approaches to detecting focal methods in terms of their precision and recall. Table 6 presents an approximation to recall—the percentage of test cases reported as having focal methods by each of the approaches. This approximates recall under the assumption that every test case has exactly one focal method. (In practice, a test case may have more than one focal method; we believe that a reasonable test should have at least one focal method.) The four projects presented under Ghafari's algorithm are from their paper [**?**], whereas we hand-picked four benchmarks from methods2test's dataset to match (as far as possible) those in Ghafari's work.

The data suggests that methods2test's name-based approach of focal method detection has low recall. This is not surprising—their heuristics have quite strict constraints, i.e. limited applicability. We reviewed some of their mock analysis results, and found that none of their reported focal method invocations were mocks, consistent with high precision.

On the other hand, Ghafari kindly shared with us recent focal method detection results. We found that there are a few types of unit test cases where they have to manually report focal methods. The first type is where their tool automatically reports a focal method that is actually a mock invocation from the developer's standpoint (an application for MOCKDETECTOR). On the other hand, their algorithm would return no focal method for test cases come with no assertion statements. Overall, Ghafari's algorithm requires more manual work to increase precision, but the percentage of test cases where they report focal methods (a proxy to recall) is acceptable.

In short, methods2test's approach finds focal methods with high precision but low recall, whereas Ghafari's algorithm has an acceptable recall but low precision. Our mock analysis can help increase precision by ruling out calls that are definitely not to focal methods.

Example: mock objects and focal methods. We continue discussing how we rule out calls by revisiting the unit test case presented in Listing 1. Figure 3 highlights the process for finding the mock object `session` and consequently the mock invocation `getRequest()` in the unit test. Both Soot and Doop implementations report this mock invocation from the test case. Thus, with the assistance of MOCKDETECTOR, `getRequest()` can be removed from consideration as a focal method. We judge `getToolchainsForType()` to be the focal method since we assume that every test has at least one focal method, and this is the only method invocation remaining after the elimination process. In addition, for the return array `basics`, its `length` attribute indeed gets checked in the assertion statement on

Table 6: Comparison of % of test cases with reported focal methods by the two automated focal method detection algorithms.

Ghafari's algorithm					methods2test				
Benchmark	Source Code KLoC	Reported Focal Methods	Test Cases	% of test cases with focal methods detected	Benchmark	Source Code KLoC	Reported Focal Methods	Test Cases	% of test cases with focal methods detected
commons-email-1.3.3	8.78	90	130	69%	goja-0.1.14/goja-core	11.52	27	80	34%
PureMVC-1.0.8	19.46	34	43	79%	mock-socket-0.9.0	1.09	4	34	12%
XStream-1.4.4	54.93	513	968	53%	project-sunbird-4.3.0/sunbird-lms-service	45.36	310	984	31%
JGAP-3.4.4	73.96	1015	1390	73%	optiq-0.8/core	93.94	26	1346	2%
Geometric Mean				68%					12%

Line 13, which means this test case indeed tests the behaviour of `toolchainManager.getToolchainsForType()`.

For this unit test, since Ghafari's algorithm does not consider accessors ("inspector methods" in their paper) as focal methods, they will presumably report no focal method for this unit test case. Ghafari's algorithm drops recall here. Incidentally, since Ghafari's heuristic requires the unit test case to have at least one assertion statement, the algorithm will presumably fail to return any focal methods when analyzing unit test cases without assertions.

As for the heuristic for methods2test, method `getToolChainsForType()` does not appear to match under the name-based heuristic from methods2test. The other heuristic finds calls to the focal class, so methods2test should identify the call to `getToolChainsForType()` with that heuristic. Our mock analysis can help with that heuristic by removing mocks from the set of eligible focal classes.

```
@Test public void testMisconfiguredToolchain() throws Exception {
  //                mock:✓    mockAPI:✓
  MavenSession session = mock ( MavenSession.class );
  MavenExecutionRequest req =
      new DefaultMavenExecutionRequest();
  //     mock invocation:✓ ⇒ focal method:✗
  when( session.getRequest() ).thenReturn( req );

  ToolchainPrivate[] basics =
    //                      focal method:✓
    toolchainManager.getToolchainsForType("basic", session);

  assertEquals( 0, basics.length );
}
```

Figure 3: Example: removing mock invocation from focal method consideration.

7 RELATED WORK

We discuss related work in the areas of focal method detection, treatment of containers, and taint analysis.

Focal methods and classes. To situate Ghafari et al's work [**?**], a number of previous works have studied test-to-code traceability by identifying focal *classes* for a test case—the classes which are tested by a test case. The focal *methods* we discuss in this paper belong to focal classes. Ghafari et al were the first to extend the study of traceability to focal methods. Before that, Qusef et al proposed techniques which identify focal classes. In [**?**], they propose a two-stage approach, which relies on the assumption that the last assertion statement in a test case is the key assertion. The first stage uses dynamic slicing to find all classes that contribute to the values tested in the assertion (possibly including mocks), while the second stage filters classes and keep only those that are textually closest to the test class. An additional mock object filter would help remove classes that are certainly not focal classes but might look like them. Earlier work by Qusef et al [**?**] uses dataflow analysis instead of dynamic slicing.

Rompaey and Demeyer [**?**] evaluate six other heuristics for finding focal classes: naming conventions, types referred to in tests ("fixture element types"), the static call graph, the last call before the assert, lexical analysis of the code and the test, and co-evolution of the test and the main code. No heuristic dominates: different heuristics succeed differently for different codebases. Our approach adds another way to rule out unwanted results both for focal methods and focal classes.

Ying and Tarr [**?**] also propose heuristics to filter out unwanted methods during code inspection. Their heuristics are based on characteristics of the call graph, i.e. they filter out methods closer to the bottom of a call graph as well as small methods. Both of their heuristics depend on tuneable parameters. It turns out that these heuristics empirically eliminate mock calls in their benchmarks, but there is no principled reason for that to be the case, and indeed, the static call graph that they depend on should not interact well with mock calls.

Treatment of containers. In this work, we use coarse-grained abstractions for containers, consistent with the approach from Chu et al [**?**]. In our experience, test cases do not perform sophisticated container manipulations where it would be necessary to track exactly which elements of a container are mocks. Were such an analysis necessary, the fine-grained container client analysis by Dillig et al [**?**] would work.

Taint analysis. Like many other static analyses, our mock analysis can be seen as a variant of a static taint analysis: sources are mock creation methods, while sinks are method invocations. There are no sanitizers in our case. However, for a taint analysis, there is usually a small set of sink methods, while in our case, every method invocation in a test method is a potential sink. In some ways, our analysis resembles an information flow analysis like that by Clark et al [**?**]. However, the goal of our analysis (detecting possible mocks) is different from taint and information flow analyses in that it is not security-sensitive, so the balance between false positives and false negatives is different—it is less critical to not miss any

potential important mock invocations, whereas missing a whole class of tainted methods would often be unacceptable.

8 CONCLUSION

Our thesis is that mock objects are an important technique that developers use when creating test suites. However, common mock object libraries use reflection and cause developers to write tests that appear to have different behaviours than they actually do—method invocations on mocks look like normal method calls, but instead record behaviour. Because of the prevalence of mocks, tools that work with tests (including static analyses, IDEs, and automatic repair tools) need to correctly handle mock objects.

We have described our MockDetector static analysis, which we intend to use for further static analyses of test cases. We have implemented MockDetector imperatively in Soot and declaratively in Doop, and characterized its performance and behaviour on a set of 8 open-source benchmarks. Our results show that mocks play an important role in achieving test coverage for some real-world benchmarks. We believe that mock analysis is a useful prerequisite for test case analysis and development, enabling numerous subsequent analyses.

ACKNOWLEDGEMENTS

We thank the Doop developers for their timely and helpful answers to our questions; developer or community support is necessary to successfully use Doop (or any research-grade program analysis framework, including Soot.) We also thank M. Ghafari for his timely response and for access to his focal method results.