Record and Replay of Online Traffic for Microservices with Automatic Mocking Point Identification

Jiangchao Liu*
Ant Group, Hangzhou, China
jiangchao.ljc@antgroup.com

Jierui Liu*
Ant Group, Hangzhou, China
liujierui.ljr@antgroup.com

Peng Di
Ant Group, Hangzhou, China
dipeng.dp@antgroup.com

Alex X. Liu
Ant Group, Hangzhou, China
alexliu@antgroup.com

Zexin Zhong
Ant Group, Hangzhou, China
zhongzexin.zzx@antgroup.com

ABSTRACT

Using recorded online traffic for the regression testing of web applications has become a common practice in industry. However, this "record and replay" on microservices is challenging because simply recorded online traffic (*i.e.*, values for variables or input/output for function calls) often cannot be successfully replayed because microservices often have various dependencies on the complicated online environment. These dependencies include the states of underlying systems, internal states (*e.g.*, caches), and external states (*e.g.*, interaction with other microservices/middleware). Considering the large size and the complexity of industrial microservices, an automatic, scalable, and precise identification of such dependencies is needed as manual identification is time-consuming. In this paper, we propose an industrial grade solution to identifying *all* dependencies, and generating mocking points automatically using static program analysis techniques. Our solution has been deployed in a large Internet company (*i.e.*, Ant Group) to handle hundreds of microservices, which consists of hundreds of millions lines of code, with high success rate in replay (99% on average). Moreover, our framework can boost the efficiency of the testing system by refining dependencies that must not affect the behavior of a microservice. Our experimental results show that our approach can filter out 73.1% system state dependency and 71.4% internal state dependency, which have no effect on the behavior of the microservice.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Automated static analysis**.

*both authors contributed equally to this research.

This research is partially supported by NSFC No.62002363.

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513029

KEYWORDS

microservice, static analysis, mocking, record and replay

ACM Reference Format:
Jiangchao Liu, Jierui Liu, Peng Di, Alex X. Liu, and Zexin Zhong. 2022. Record and Replay of Online Traffic for Microservices with Automatic Mocking Point Identification. In *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513029

1 INTRODUCTION

1.1 Background and Motivation

Using recorded online traffic for the regression testing of web applications has become a common practice in industry [2, 22, 25]. In the recording phase, it stores online traffic (*e.g.*, the input/output values) of a given online application. In the testing phase, it replays the online traffic and check whether unexpected behaviors are observed. This "record and replay" paradigm has two advantages. First, it is most efficient for testers as online traffic can be automatically recorded and modified programs can be automatically tested using the recorded traffic. Second, it is usually closer to reality than both manually written test cases and automatically generated test cases by other techniques such as fuzzing [11, 26, 29, 40].

However, record and replay on microservices is challenging because simply recorded online traffic often cannot be successfully replayed (*i.e.*, to reproduce the same output) because microservices often have various dependencies on the complicated online environment. For example, the function `C1.m1()` in code fragment in Figure 1 would get a different value each time it is executed even if the code is not modified. Because the return value of the function is calculated not only from the parameter of the function but also the underlying random seed. If the value of a variable or the result of a invocation is not just depends on the parameter of the function, we say that those variables and invocations has *state dependency*.

```
1  public class C1 {
2      public static String m1(String input) {
3          return input + toString((new Random()).nextInt());
4      }
5  }
```

Figure 1: An example function that has state dependency

We use the above example to explain why for microservices that have state dependency, straightforward record and replay of online

traffic do not work, and how the correct record and replay should be conducted. Assume that we want to record the output of the function `C1.m1()` with an empty input string (*i.e.*, ""), and the function `(new Random()).nextInt()` returns a random value 1 when we first execute the code for recording traffic on function `C1.m1()`. As a result, we would record the output of function `C1.m1()` as the string "1". Directly replaying the function `C1.m1()` makes `(new Random()).nextInt()` generate another random value (*e.g.* 2). The output of function `C1.m1()` would be "2", which is not the same with the recorded string "1", thus the replay fails even though we do not modify any code of the function `C1.m1()`. To make record and replay work, we need to record the output value 1 for the function `(new Random()).nextInt()` in the recording phase. In replay, we mock the function `(new Random()).nextInt()`, replacing its output by the recorded value 1. In this case, the output of function `C1.m1()` would be "1", which is the same with the recorded output, and the nondeterminism of the function `(new Random()).nextInt()` is eliminated.

To address state dependencies, the record and replay paradigm records not only the input and output of a microservice, but also the input and output of the functions that have state dependencies. We call a calling site of a function that has state dependencies a *mocking point*. Thus, we need to automatically identify all state dependencies and generate corresponding mocking points as manually identifying all mocking points is time-consuming. After identifying state dependencies, in the record phase, for each mocking point, we record the corresponding input and output. In the replay phase, for each mocking point, we use the recorded output to replace the real output of the corresponding function.

State dependencies can be categorized as *system state dependency*, *internal state dependency*, and *external state dependency*. For *system state dependency*, the output of a microservice may depend on the state of the underlying system, which may return different values for the same system call at different times. For example, suppose the output of a microservice relates to the current time, if we simply record the output of this microservice, even if the microservice remains unchanged, the recorded previous output (previous time) would not match the current output (current time), which means that the replay would fail. For *internal state dependency*, the output of a microservice may depend on the internal state of itself, which may keep changing. Thus, the recorded output of a continuously running microservice based on a previous internal state may not match the output of this service based on the current state. For example, suppose a microservice simply returns the number of times that it has been called, if we simply record the output of this microservice, even if the microservice remains unchanged, the replay would fail for the number of to-be-called times changes. For *external state dependency*, the output of a microservice may depend on the behavior of other microservices as in industry most microservices need to interact with other microservices via mechanisms such as Remote Procedure Call (RPC), shared databases, asynchronous messages, injected fields or functions invisible in the code, etc. Thus, the recorded output of a microservice that depends on the behavior of other microservices at a previous time may not match the output of this microservice at the current time because the behaviors of the depended microservices may become different.

1.2 Limitations of Prior Art

We are not aware of prior work on mocking point identification for microservices. Prior mocking point identification scheme proposed by Arcuri *et al.* [5] was designed for unit testing, and is not suitable for microservices because its mocking scheme do not handle external state dependencies, which are prevalent for microservices.

1.3 Proposed Approach

In this paper, we propose an program analysis approach to identifying all state dependencies for the record and replay paradigm. For system dependencies, to avoid too many recording of system calls, we track the flow of outputs of system calls via static taint analysis and identify the caller functions of those system calls as mocking point if necessary. In this way, we also can filter out the system calls, the return values of which do not affect the process of replaying process, to ensure as few mocking points as possible are identified. For example, the function `C1.m1()` (in Figure 1) would be identified as a mocking point but the function `C1.m2()` in Figure 2 would not, because the return value of function `(new Random()).nextInt()` in `C1.m2()` is just used for logging and not affect the output of function `C1.m2()`. For internal dependencies, we identify all functions that read the values of static fields as mocking points. We also perform a static analysis to filter out all never-modified static fields (*e.g.*, field with the *final* identifier) to reduce unnecessary recording because they would not obstruct the replaying process. For external dependencies, we perform static program analyses on Java code and XML configuration files to collect function calls to access external components, which include accessing databases, invoking functions on other microservices, and interacting with other middle-ware components. These functions are identified as mocking points. For third-party libraries, because microservices may invoke various third-party libraries, and it is hard to identify whether a third-party invocation should be mocked or not, we use a global white list and allow the tester to manually add third-party library invocations, which should be mocked, into it. All functions in the white list are also identified as mocking points. For recording and replaying, in the record phase, we instrument logging operations at the beginning and end of the mocking function to record the parameter values and return value, respectively; in the replay phase, we instrument code to make mocking function return the recorded value before executing the code in the function.

```
1  public class C1 {
2      public static long m2(long input) {
3          printLog((new Random()).nextInt());
4          return input;
5      }
6  }
```

Figure 2: An example function not to be mocked

1.4 Summary of Experimental Results

We have implemented our approach as a microservice and successfully deployed it in Ant Group, which operates one of the biggest digital payment platforms (*i.e.*, Alipay) and serves over one billion users. Instead of manually building test environment by engineers,

our approach automatically generates mocking points with high replay success rate (99% on average). Moreover, our experimental results show that our approach can filter out 73.1% system state dependency and 71.4% internal state dependency via static analysis, which have no effect on the behavior of the microservice. It means that our approach can minimize the performance overhead in online recording by filtering out dependencies that does not affect the behaviors of microservices. There is only a few third-party library mocking points added into white list manually (1.6% on average), the majority of mocking points are automatically identified (98.4% on average).

1.5 Contributions

In this paper, we make three key contributions.

- We identified *all* state dependencies that may result in different behaviors of industrial microservices between the record and replay phase.
- We proposed a soundy and scalable static analysis that can automatically identify mocking points satisfying our industry deployment requirements based on state dependencies.
- We have implemented our approach and deployed it, as a key part, into an industrial record and replay testing system. The system daily runs on hundreds of industrial microservices with high replaying success rate (99%).

2 SYSTEM ARCHITECTURE

In this section, we introduce the architecture of the whole record and replay paradigm, as shown in Figure 3.

2.1 Applying Scenario

In industry, an individual microservice, serves as a unit in many different functionalities, is almost always backward compatible, which means it must support the old traffic and keep idempotent. This makes it possible for the record and replay paradigm to be used for regression testing. For a microservice, traffic recording is performed from time to time to collect test cases. Replaying is performed when any change is committed to the code of the microservice. Mocking point analysis must be performed before traffic recording and re-performed after every new iteration to collect new mocking points.

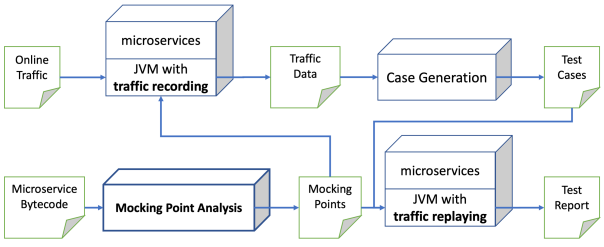

Figure 3: Record and Replay platform architecture

2.2 Mocking Point Analysis

Mocking Point Analysis module takes the bytecode of a target microservice as input and utilizes static analysis to generate the list of *mocking points*. The analysis aims to find all state dependencies of the given microservice and the output (i.e., *mocking points*) of analysis would be used in traffic recording and replaying. Our analyzer is soundy and precise to satisfy the industrial scenario. In terms of soundy, mocking points analyzing approach should be performed within acceptable time to identify all mocking points. Otherwise, a missed mocking point may incur many failures in the later replaying phase. In terms of precise, unnecessary mocking points would bring extra untestable code and heavy overhead to the online environment. Therefore the static analyzer plays an important role in the architecture shown in Figure 3 and the details of its implementation will be illustrated in the following sections.

2.3 Record

Traffic Recording module utilizes a `JVM` injector to record the online traffic [1] on all mocking points. The injector is deployed on the same machine with the target microservice. It intercepts all functions in the mocking point list and records their input and output, along with the input and output of the microservice. The recorded data is saved as *traffic data*, and will be later used to generate test cases. Recording brings extra performance overhead to online services, which is proportional to the number of mocking points. Our experiments show that each recording operation would take extra 1ms overhead in average. It is essential to keep as few mocking points as possible.

2.4 Case Generation

Case Generation module processes *traffic data* to make test cases ready for replaying. It includes data desensitization, data encryption, data cleaning and data consolidation. Here we briefly give the form of the result test cases produced by this module. Each test case is a partial execution trace of functions with inputs and outputs on them. The functions include the entry point of the microservice and the mocking points. Figure 4 shows a test case, as a the partial execution trace of a microservice.

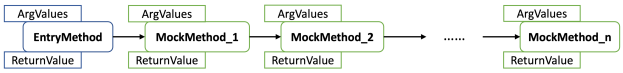

Figure 4: A case execution trace

2.5 Replay

Traffic Replaying also utilizes a `JVM` injector to replay the test cases generated from online traffic. This process involves mocking and observing. Given a test case, the traffic replaying module starts the microservice with the recorded inputs on an entry point, intercepts all mocking functions in the test case and returns the recorded outputs. The module tell us whether the replaying succeeds or not

[1] The data set is desensitized and encrypted. The data set does not contain any Personal Identifiable Information (PII).

by observing: (1) whether all mocking functions are called in the same sequence with the current test case execution; (2) whether the outputs on the entry function are exactly the same. Figure 5 shows the two failed cases because of different execution paths and unmatched output on entry function respectively. The *execution trace 1* is failed because it outputs 2 as the result, which is different from the base output (i.e., 1). The *execution trace 2* is also failed, although it outputs the same value with the base trace, because it does not execute *method3* that should be executed.

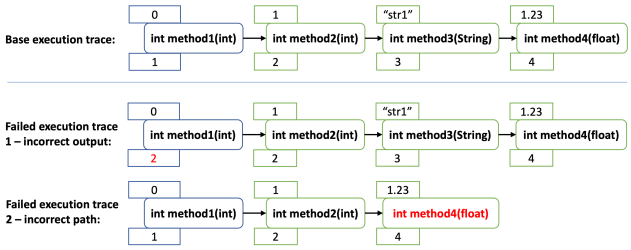

Figure 5: A failed case example

Only the recorded test cases that can be successfully replayed would be saved and used in the later regression testing. Thus, replaying is first performed on the same version of the microservice as in recording. In this case, a replay failure indicates some missed mocking points or some bugs in other parts of the system. The failed tests are manually diagnosed to locate the cause. If the cause is a missing dependency, we would add it to the dependency white list (more details in Section 3.4). The test cases that are successfully replayed are used in regression testing when any change to the microservice is committed.

From the architecture of record and replay platform, we can see that the *mocking points* is indispensable for entire record and replay process. In next section. we will introduce more technical details for the **Mocking Point Analysis** module.

3 PROPOSED APPROACH

In this section, we present the technical details on identifying mocking points for system/internal/external state dependencies. Although our techniques can be generalized for other platforms, it is currently built over SOFA/SOFABoot framework [3], which is based on Spring/SpringBoot [21].

3.1 System State Dependency

System state dependencies are library calls the outputs of which depend on the state of the underlying operating system (e.g., time, memory usage, etc.). Since the state of the underlying operating system changes from the online environment to the offline environment, system state dependencies must be mocked.

3.1.1 Source of System State Dependency. System state dependencies are in the form of function calls from runtime library, i.e. `rt.jar`, or other third-party libraries. Some common function calls are listed in Table 1.

In the table, functions No. 1 to 2 are used for getting system status. Functions No. 3 to 11 are used for getting current date or time. Functions No. 12 to 17 are used for generating random values. We classify these functions as system state dependency because their seeds are often set by system states (e.g., system time). Functions No. 18 to 20 are used for getting internet information, and the returned values depend on the system configuration. Functions No. 21 to 22 are from third party libraries. They usually wrap functions from `rt.jar` and strengthen them with more features.

Table 1: Methods relying on system state

rt.jar		
No.	**Class Name**	**Method Name**
1	java.lang.System	getenv()
2		getProperty()
3	java.util.Date	<init>()
4	java.lang.System	currentTimeMillis()
5	java.util.Calendar	get()
6	java.time.LocalDate	now()
7	java.time.LocalTime	now()
8	java.time.LocalDateTime	now()
9	java.time.Clock	system()
10		systemUTC()
11		systemDefaultZone()
12	java.util.UUID	randomUUID()
13	java.lang.Math	random()
14	java.security.SecureRandom	getInstance()
15		getInstanceStrong()
16	java.util.Random	*
17	java.util.concurrent .ThreadLocalRandom	*
18	java.net.InetAddress	getLocalHost()
19		getHostAddress()
20		getHostName()
Other Libs		
No.	**Class Name**	**Method Name**
21	org.apache.commons.lang3 .RandomStringUtils	*
22	org.apache.commons.lang .math.RandomUtils	*

For functions from third party libraries, we take them as mocking points directly. For function calls from `rt.jar`, we do not mock them directly for two reasons. First, there are too many invocations to system calls in `rt.jar`, recording traffic on them directly would bring unacceptable performance loss to the online microservices. Second, a large majority of system calls do not affect the output, so most recorded data for system calls are useless and saving those useless recorded data wastes a lot of disk resource. As a result, we perform a taint analysis to move the mocking points from these system library calls to other mockable functions.

```
1  public class C2 {
2      public int m1(AClass anObject) {
3          int random = (new Random()).nextInt();
4          if (random < 10) {
5              return 10;
6          }
7          anObject.aField = random + 1;
8          return 0;
9      }
10 }
```

Figure 6: An example of taint analysis

3.1.2 Sources and Sinks. Taint analysis is a classic information flow analysis which aims to track sensitive data by propagating the tainted data from *sources* to *sinks*. There exists a lot of work on static or dynamic taint analysis [8, 12, 38, 39]. Due to the complexity of the industrial scenario, existing approaches can hardly be applied because of unscalability. Thus, we develop a lightweight and scalable taint analysis, rather than adopting conventional taint analysis algorithms.

In our scenario, system calls involving nondeterminism are defined as *sources*. Unlike the conventional definition of *sinks*, our *sinks* are *branch* or *return* statements. If a tainted value flows to a *branch* or a *return* statement, the behaviors of the function is affected by the nondeterminism. Thus, the entire function needs to be mocked. We do not propagate tainted values through branch statements because that needs *implicit taint analysis* [13], which is costly and brings little precision improvement to our analysis.

Example 3.1. A code segment is shown in Figure 6. At line 3, the variable *random* is tainted since its value is calculated from a system call generating random values. At line 4, the tainted variable *random* is used in the *branch* statement, which means it may affect the behavior of function `C2.m1()`. Thus the function `C2.m1()` is marked as a mocking point.

3.1.3 A Lightweight and Soundy Taint Analysis. In Algorithm 1, we first show our intra-procedural taint analysis algorithm. Later we will extend it to be inter-procedure. The algorithm starts with initializing a worklist with the sources (*i.e.*, some library calls to *rt.jar* or nondeterministic variables). For each source (*e.g.*, `(new Random()).nextInt()` in Figure 6), we collect all functions (e.g., `C2.m1()`) that read values from these sources. Mocking of a function is determined by propagating an abstract state $< T, b >$, where T represents all tainted values and b represents whether the current function should be mocked. The propagation rules inside a function are defined between line 14 and 25. For *assignment* statements, the left value is added to the tainted set T if the right expression contains tainted values in T. For *branch*/*return* statements, the whether-to-mock flag b is set to true if *branch*/*return* statements contain tainted values in T. For *loop* statements, we compute its least fixed point. For example, given a *while* block, which is a list of statements, we apply *processStmt* on them repeatedly until the output on all statements are stable. It is easy to know that this process terminates because both the size of the set T and the states of b are finite.

After the computation of *processMethod* is finished, we would get the result tuple $< T, b >$. If b is $true$, then this function is added to the mocking list. If T contains fields, they are added to $Worklist$.

Algorithm 1 Mocking points identification

T denotes a set of values (*i.e.*, variables, fields or methods) that are tainted
b denotes whether the current method is to be mocked
M denotes the set of methods to be mocked

```
1:  Worklist := system state dependencies
2:  while Worklist ≠ ∅ do
3:      pop source v off Worklist
4:      for method m that contains statement reading v do
5:          < T, b > = processMethod(m, < {v}, false >)
6:          for field f in T do
7:              add f to Worklist
8:          if b then
9:              add m to M
10: procedure processMethod(m,< T, b >)
11:     for stmt in the list of statements in m do
12:         < T, b > = processStmt(stmt, < T, b >)
13:     return < T, b >
14: procedure processStmt(stmt, < T, b >)
15:     switch stmt do
16:         case Assignment Statement:lv = expr
17:             if expr is tainted then
18:                 add lv to T
19:         case Branch Statement: if(expr)
20:         case Return Statement: return expr
21:             if expr is tainted then
22:                 b := true
23:         case While Statement: while(expr){block}
24:             apply processStmt on expr/block until a fixed point
25:     return < T, b >
```

This generalize the taint on the field of an object to the field of that class. It is different with conventional taint analyses, which are usually object-sensitive. The reason for this choice is that, it avoids costly pointer analysis and usually does not increase the size of mocking points. Because the tainted field usually results in its getter function as a mocking point.

Example 3.2. At line 7 in Figure 6, a field `aField` is tainted because it reads a tainted variable *random*. We propagate the taint on field `aField` of an Object *anObject* to the field `aField` of the class *AClass*. This means any places that read the field `aField` will be tainted.

Now, we extend our algorithm to be inter-procedure. We assume that function invoking expressions only exist in assignment statement in Algorithm 1. It is a valid assumption, since any statements involving function calls can be transformed into this form. Given such an assignment `lv = invoke(m)`, our algorithm computes $< T', b' >= processMethod(m, < T, b >)$. If b' is $true$, then `lv` is tainted. But the function `m` is not added to the mocking list. This indicates that if the tainted values affect the behaviors of the function `m`, it just propagates the taint, instead of taking `m` as a mocking function. The tainted values set T' (probably with new tainted fields) is propagated to the current method. We do not limit the depth of inter-procedure analysis on application methods, but

```java
public class AppClass {
    private static Integer counter = 0;
    private static Map<String, String> cache = new HashMap<>();
    public static Integer getCounter() {
        return counter;
    }
    public static String getValue(String key) {
        ++counter;
        return cache.getOrDefault(key, "");
    }
    public static void putValue(String key, String value) {
        cache.put(key, value);
    }
}
```

Figure 7: An example of static fields

```java
// JAVA code in an application
public class BeanClassImpl {
    private String info = "someInfo";
    public String getInfo() {
        return info;
    }
    public void setInfo(String info) {
        this.info = info;
    }
}

public class AppClass {
    @Autowired
    private BeanClassImpl bean;
}

// XML configuration in application
<bean id="bean" class="BeanClassImpl"></bean>
```

Figure 8: An example of bean usage

restrict that on library methods to be 2. This is unsound, but we did not observe any missing mocking functions from this limitation.

Example 3.3. Applying *processMethod* on method call `printLog()` in Figure 2 results in $< \emptyset, false >$, which means that the propagation of the taints from `(new Random()).nextInt()` does not affect the output of `printLog()` and no new tainted fields. In contrast, applying *processMethod* on method call `toString()` in Figure 1 results in $< \emptyset, true >$, which means that the taint is propagated through the method call `toString()` and the return value of `toString()` is tainted.

3.2 Internal State Dependency

Internal state dependencies refer to the internal states of an application that may incur different behaviors in online and offline environments. The internal states are usually stored in objects that can be accessed by multiple threads (e.g., static fields, Spring beans). They are often used as caches or to record the status of the running microservice. These objects can be written in one thread and be read in another. In our recording phase, we deploy the online injector when microservices are running. This means that these objects may have been modified many times before we start recording. Thus we have to record the internal states (i.e., values stored in these objects) to reproduce the online traffic.

3.2.1 Identification of Internal Dependency. In J2EE applications, static fields and Spring beans are usually used to store internal states.

Static fields can be easily identified. They always come with a specifier `static`. Fields with this specifier are allocated at *Metaspace* and are destroyed when the class is unloaded. A static field can be accessed by multiple threads during its lifetime. Figure 7 shows two typical usages of static fields. Static field `cache` is used as a cache, which supports faster access than database. Static field `counter` is used to record the time that `getValue()` is called.

Spring beans are the objects that form the backbone of applications and are managed by the Spring IoC container [21] (this feature is inherited by SOFA/SOFABoot). If a *bean* is declared with the scope "singleton", it can be accessed by multiple threads. There are multiple ways to declare a bean in SOFA or SOFABoot. Here we just show a conventional way which relies on XML files as shown in Figure 8. It declares a java class `BeanClassImpl` as a bean with id "bean". Other classes can use it with this id, as in `AppClass`. The default scope of a bean is "singleton", which means, one single instance of `BeanClassImpl` is managed by Spring IoC container and is shared by all `AppClass` instances (even if in different threads). Thus the non-static field `info` can also store internal states.

3.2.2 Mocking Points for Internal Dependency. We need to find a way to isolate internal state dependencies from the rest of a microservice. In [5], the authors proposed to reset the static fields before every read on them. This method is not suitable in our scenario because we are not allowed to reset the internal states in the online environment. We can only utilize mocking functions to achieve this. Actually, we rely on the same algorithm in Section 3.1.3 to find mocking points, by taking each static/bean field as "sources" in the taint analysis. In this way, the task left for us is to find all static/bean fields.

Static analysis can identify static fields via the identifier "static". For bean fields, we need to find all bean classes and then extract all their fields. SOFA/SOFABoot support various ways to define bean classes, *e.g.*, with tags like "Bean" or with XML files as in Figure 8. They can be all identified with static analysis via pattern matching.

Example 3.4. Take the code segment in Figure 7 as an example. Fields `cache` and `counter` are identified as sources of internal state dependency because of their specifier "static". Via the taint analysis in Section 3.1.3, our algorithm automatically generates the mocking points `getCounter()` and `getValue()`.

3.2.3 Refinement. Mocking points generated from Section 3.2.2 are actually not all necessary. Our algorithm for generating mocking points on internal state dependencies can be refined. The refinement is achieved via revision on the process of identifying static/bean fields to rule out the static/bean fields that are never modified. These fields can be seen as constants and do not incur different behaviors in online and offline environments. It is easy to know that the immutable fields (with specifier `final`) of immutable types (e.g., of primate types) are never modified. These fields can be easily recognized. The tricky part is that many mutable static/bean fields without the specifier `final` are actually never modified. Even if there exists code segments that modify a static/bean field, those code segments can be dead code. One immediate idea is to find out that for a static/bean field, whether there exists a path from an entry point of the microservice, on which it is modified. This task is challenging.

Due to reflection and other complex features of JAVA and the frameworks, a sound call graph can hardly be computed with current techniques [31]. With an unsound call graph, our analysis may

infer that the code segments that modify a static/bean field are not reachable, even if they are reachable actually. This can make a static/bean field wrongly ruled out, and result in a missing mocking point. A conservative way is to keep all static/bean fields that there exists code modifying them. However, it is common that every field comes with a setter function (that may never be called) which modifies it, especially for bean fields. Thus our analysis follows the strategy below to rule out static/bean fields.

- If there does not exist code that modify a static/bean field, then it is ruled out.
- if all functions (called modifiers) that modify a static/bean field are never called, then it is ruled out.

Example 3.5. In Figure 8, the field `info` will be ruled out if its setter function `setInfo()` is never called in the microservice.

3.3 External State Dependency

External state dependencies are functions the outputs of which depend on the states of other microservices or other computers. These dependencies are often in the form of middleware.

3.3.1 DB Accesses. Most online microservices access databases. The behaviors of such microservices rely on the states of databases. Since it is not practical to move the whole database from the online environment to the offline environment for each test case, we need to mock all accesses to databases.

In SOFA/SOFABoot applications, two common middleware that help access databases are Ibatis [20] and Mybatis [14]. In the Ibatis framework, a Dataset Access Object (DAO) will be declared, which inherits from a framework class `SqlMapClientDaoSupport`, to perform database operations. So we just pick all sub-classes from class `SqlMapClientDaoSupport`), and put them into the mocking list. This can be achieved with a class hierarchy analysis [36]. In the Mybatis framework, programmer only need to define an interface to declare the operations on database, and the implemented code is generated dynamically via proxy class. Since an interface method should not be a mocking point, we pick all methods, which invoke the method of Mybatis mapper interfaces, into the mocking list.

3.3.2 RPC. RPC (Remote Procedure Call) [10] is a protocol for a program to call a procedure on another computer. It's one of the infrastructures for distributed systems. With RPC, a microservice can call another microservice on other computers. RPCs are external state dependencies.

There are many implementations of RPC. In SOfA/SOFABoot, RPC is implemented via SOFARPC [3]. In SOFARPC, the references to RPC entry point is defined in XML configuration files via the tags `sofa:reference` and `sofa:binding.tr`. RPC calls can be collected by parsing XML configuration files and analyzing the microservices' code. Thus, we put those RPC calls into the mocking list.

3.3.3 Proximal Calls. *Proximal calls* are calls to the functions of other proximal microservices running on the same JVM. Proximal microservices are usually taken as a blackbox and we do not record traffic inside it. Thus, proximal calls are external state dependencies. To execute a proximal function in a proximal microservice, we need to do the following two things. First, in the proximal microservice, we define an interface API and implementation classes, and declare their mapping relation with `sofa:service` and `bean` tags in a XML configuration file. Second, in the current microservice, we declare the proximal interface that we want to call in a XML configuration file with tag `sofa:reference`. Thus, we need to perform static analysis on both the current microservice and the proximal microservice to find implementation classes for proximal interface. First, our analysis collects interface information (with tag `sofa:reference`) from the configuration files in the current microservice. Since we do not know which microservice each interface comes from, we have to traverse all dependent packages and collect interface information (with tag `sofa:service`) and class information (with tag `bean`). Combining these information, we can figure out the real implementation classes for each proximal call. Finally, we put all those implementation classes and their methods into the mocking list.

3.4 Dependency White List

In previous subsections, we show how to generate mocking points on system state dependencies from *rt.jar* and external state dependencies from DB/DRM/proximal-call/RPC. Apart from them, some system/external state dependencies are from third-party libraries (*e.g.*, HTTP APIs). Such dependencies can be taken as mocking points directly. However, we can hardly enumerate all such dependencies. Of course, we can choose to mock all third-party library calls. But this would bring unacceptable performance loss to the online services. Our strategy is to establish a white list. Every time we find such dependencies, we put them into the white list. To implement this strategy, we utilize a failure-feedback approach to update mocking white list. After the replaying phase, the failed cases (if there are any) are manually investigated. If it is caused by dependencies in third-party libraries, then we add those dependencies to the white list. We may never enumerate all such dependencies, but as we collect more and more such dependencies, less and less manual effort would be needed.

3.5 Discussion on Soundness

The soundness of our algorithm is defined by whether it can isolate all dependencies. Our algorithm is soundy with two assumptions: (1) All of unknown third-party library calls with system/external dependencies are manually labeled into a white list; (2) No field that is assigned in deeper-than-two library functions flows to the microservice (see Section 3.1.3). It is worth mentioning that, even if our algorithm is soundy, the replaying may not be 100% successful. The reason is that the side-effect of mocking functions. Figure 9 shows such an example. The function `C3.m1()` should be mocked because of system state dependency `(new Random()).nextInt()`, but the side-effect (line 3) to field *someObject.someField* is not handled. So in the replaying phase, the field *someObject.someField* would not be assigned with 1 and it may cause an inconsistency in somewhere uses the field. The side-effect limitation is caused by the operation of mocking functions rather than the unsoundiness of our analysis. Although such limitation exists, our approach still obtains very high success rates in the replaying phase on our industrial microservices for this scenario rarely happens.

Table 2: Experimental Results on 10 Microservices

Microservice	Scalability		Success Rate			Automation						Proportion	
	#LOC	Time	#Failed	#All	Success%	#I	#E	#S	#W	#T	White%	#C	#T/#C
M1	814K	110s	599	15818	96.2%	449	231	43	21	744	2.8%	22729	3.2%
M2	141K	78s	0	19864	100%	28	33	8	1	70	1.4%	3729	1.9%
M3	384K	86s	60	36515	99.8%	189	194	67	3	453	0.7%	11586	3.9%
M4	392K	147s	34	1895	98.2%	112	128	50	2	292	0.7%	14317	2.0%
M5	175K	59s	46	10405	99.5%	119	107	18	3	247	1.2%	4166	5.9%
M6	166K	70s	0	40818	100%	144	36	26	6	212	2.8%	4723	4.5%
M7	351K	68s	1019	93837	98.9%	50	170	13	3	236	1.2%	10597	2.2%
M8	567K	156s	486	10898	95.5%	86	227	79	3	395	0.8%	17471	2.3%
M9	94K	89s	214	13268	98.4%	199	38	14	0	251	0%	2597	9.6%
M10	88K	163s	111	10110	98.9%	27	84	30	6	147	4.1%	2516	5.8%
total	**3172K**	**1026s**	**2569**	**253428**	**99.0%**	**1403**	**1248**	**348**	**48**	**3047**	**1.6%**	**94431**	**3.2%**

```
1  public class C3 {
2      public int m1(SomeClass someObject) {
3          someObject.someField = 1;
4          return (new Random()).nextInt();
5      }
6  }
```

Figure 9: Side-effect of mocking function

4 EVALUATION

4.1 Implementation

We have implemented our analysis as a microservice based on GraalVM [28]. When our implementation receives a request from the record and replay platform, it analyzes the bytecode of the target microservice and returns the mocking points synchronously.

Our implementation is deployed on a group of elastic cloud clusters with eight 2.5GHz cores and a RAM of 32GB. It has analyzed hundreds of production microservices. We pick the results on the first 10 analyzed production microservices [2] to evaluate the effectiveness of our algorithm.

4.2 Experimental Results

The evaluation is set to answer the following research questions.

- How scalable is our approach when it is applied on industrial microservices?
- What is the success rate in replaying with our approach?
- What the level of automation does the "record and replay" platform achieve with our approach?
- How well does our approach perform in minimizing the scope of mocking functions?

4.2.1 Scalability. Our experiments show that our approach can efficiently works on large scale industry-level microservices. In Table 2, the first column *Microservice* denotes 10 anonymous microservices implemented with SOFA/SOFABoot framework. The second column *#LOC* denotes the total number of lines of code for all Java file in the microservices. We can see that these microservices are quite large, with 814 thousands lines of code at maximum and 88 thousands lines of code at minimum. The third column *Time* denotes the total time cost to finish the entire analysis. From the table, we can see that our analysis can be finished in a matter of minutes on all the microservices. For the largest microservice *M1* in the list, the time cost is just within 2 minutes. Our mocking point analysis works offline and a couple of minutes is acceptable. Actually, our framework keeps daily recording and replaying hundreds of industrial microservices under elastic cloud environment, our mocking process only occupies very small proportion of the execution time of the whole record and replay process.

4.2.2 Success Rate in Replaying. Our experiments show that our approach leads a very high success rate in case replaying. In Table 2 (columns 4-6), the column `#Failed` and `#All` denote the number of failed and all test cases in replaying respectively. The Column `Success%` denotes the success rates in replaying and is calculated by the formula $\frac{(\#All - \#Failed)}{\#All}$. As shown in the Table 2, the average replaying success rate reaches 99.0%, with 100% in the best case and 95.5% in the worst case. After manual investigation, the failed cases are caused by side-effects (Section 3.5). As a comparison, the success rates without mocking points are all 0 because most industrial microservices call other microservices or access databases. Replaying without mocking is almost sure to fail.

4.2.3 Automation. Although our static analysis is fully automatic, manually adding extra mocking points to a whitelist is needed because of unknown libraries as mentioned in Section 3.4. Our experiments show that there is a very high level of automation of our technique. In Table 2 (columns 7-12), the column *#I/#E/#S* denote the number of automatically generated mocking points for internal/external/system dependencies respectively. The column *#W* denotes the number of mocking points from the white list. The column *#T* denotes the total number of mocking points, where

[2] Adequate data protection was carried out during the experiment to prevent the risk of data copy leakage, and the data set was destroyed after the experiment. The data set is only used for academic research, it does not represent any real business situation.

$\#T = \#I + \#E + \#S + \#W$. The column *White%* denotes the proportion of white list mocking points and equals $\frac{\#W}{\#T}$. From the table, we can see that, in different microservices, the ratios of mocking points for different dependencies vary. In M1, the dependency type resulting in most mocking points is internal state dependency, while in M7, that dependency type is external state dependency. However, in all the 10 microservices, mocking points from while-list remains the fewest. Actually, there is only a very small portion (1.6% on average) and a very small number (fewer than 10 in most cases) of white list mocking points. Moreover, the white list is designed to be reused because the middleware and external libraries invoked in microservices in the same industrial scenario are probably the same. Thus, the manual effort is usually one-off service. With more microservices analyzed, more white-list mocking points will be added and reused, and fewer and fewer mocking points needed to be manually added.

4.2.4 Minimizing the Mocking Scope. We answer this research question in the following two dimensions: (1) the proportion of mocking points in all invoked methods; (2) the revision result on system/internal state dependencies.

Mocking Point Proportion: Our experiments show that the amount of mocking points takes a very small portion in all method invocations. In Table 2, The second last column *#C* denotes the number of distinct methods invoking (exclude those in `rt.jar`). The last column denotes the mocking point proportion (*#T/#C*). From the table, we can find that our mocking points only takes 3.2% on average. This means the mocking points will not affect code coverage in replaying much.

Mocking Point Revision: Our experiments show that our analysis on system and internal state dependencies can filter out a large amount of unnecessary mocking points. The diagram in Figure 10 shows the experimental result of our revision on system state dependency. The left bar (*All*) denotes the number of all system state dependencies collected according to Table 1. The right bar (*FromIntra +FromIner*) denotes the number of actual system state dependencies that need to be mocked after our revision. It can be seen that more than half of the system state dependencies are filtered out by our analysis. To be more specific, there are 1427 system state dependencies in all 10 microservices and only 384 need to be mocked in total, and our analysis can reduces 73.1% unnecessary system state dependencies on average. We divide the mocked system state dependencies into *FromIntra* and *FromInter*, to represent ones that need intra-procedural and inter-procedural analysis, respectively. The diagram shows that quite a few system state dependencies need to be identified with inter-procedure analysis in every microservice. To be more specific, there are 55.2% (212/384) system state dependencies identified with inter-procedure analysis. This shows that it is necessary for us to perform inter-procedure analysis to identify system state dependencies, otherwise a lot of system state dependencies would be missed.

The diagram in Figure 11 shows the experimental result of our revision on internal state dependency. The left bar (*All*) denotes the number of all internal state dependencies (*i.e.*, all static fields without `final` identifier and all bean fields). The right bar (*OurRevision*) denotes the number of actual internal state dependencies that need to be mocked after our revision. The diagram shows

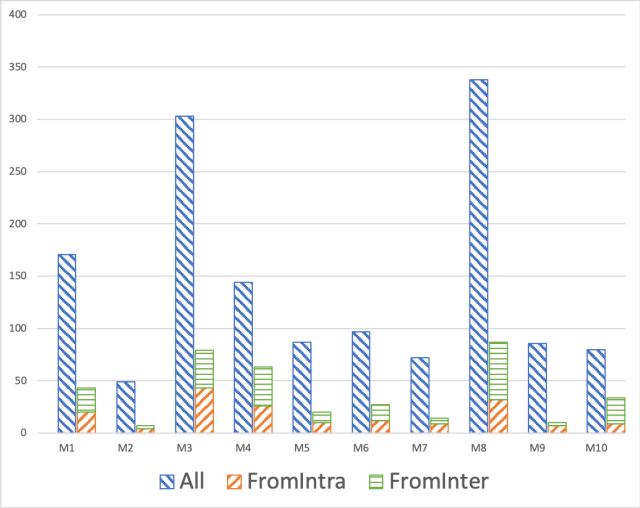

Figure 10: Revision on system state dependencies

that more that half of the internal state dependencies are filtered out by our analysis. To be more detailed, there are 3531 internal state dependencies in all 10 microservices and only 1010 need to be mocked in total. Our analysis can reduces 71.4% unnecessary internal state dependencies on average. Besides, the middle bar (*WithModifier*) denotes the number of internal state dependencies with modifiers (*i.e.*, functions that modify them). To be more detailed, 66.5% (2349/3531) internal state dependencies have modifiers. This shows that it is necessary to check whether the modifiers for an internal state dependency are invoked rather than only checking whether an internal state dependency has modifiers. Otherwise, our revision would only be able to reduce 33.5% unnecessary internal state dependencies.

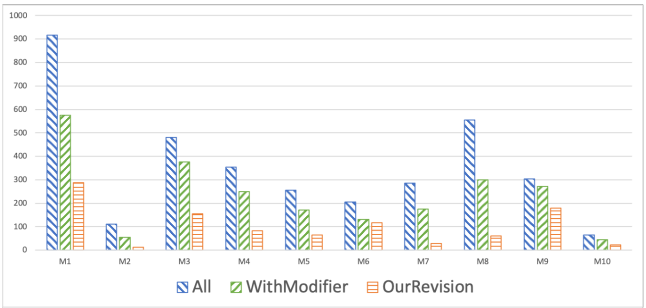

Figure 11: Revision on internal state dependencies

5 RELATED WORK

We review the work related to record and replay, mocking recommendation and static taint analysis for enterprise application.

Record and replay, as a popular way of debugging and testing, has been well studied and implemented on various platforms, including Web [2, 22, 25], Mobiles [16, 19, 30], Windows [17], Linux [15], etc. Most of these work share the same insight with this paper, that

is, to isolate nondeterministic behaviors to reproduce the same results in record and replay. We are inspired by them, since some nondeterministic behaviors are common across different platforms, like system calls that return the current time. But we face unique challenges (e.g, the performance requirement), since our target is online microservices, which is different from all of them. There are some works [7, 24, 33] applying record and replay on microservices. However, these works aim at reproducing bugs and only track the interaction between microservices, and do not guarantee the identical behaviors in online and offline environments.

Mocking is used to isolate nondeterminism or irrelevant code in testing. Mocking recommendation techniques [5, 41] try to find mocking points automatically. Some works [27, 34, 35] perform empirical studies to find the insights of mocking decisions by collecting the characteristics of mocking points. Beyond providing insights, MockSniffer [41] recommends mocking points automatically with the help of AI. These works target at general principles in mocking decision. Other works [5, 6, 19, 37], like ours, are designed for isolating dependencies. These works share the same challenge, which is to identify all dependencies. However, our scenario (*i.e.*, online microservices) brings additional challenges, which include that the number of mocking points should be restricted and system calls would not be mocked.

Static analysis for industry [4, 9, 18, 23, 32] varies from simple pattern matching [12] to AI based methods [1]. Specifically, this paper proposes a lightweight and scalable taint analyzer for industrial applications, since existing taint analyzers [8, 12, 38, 39] are heavy, even with on-demand alias analysis [8] to improve efficiency.

6 CONCLUSION

This paper proposes a new approach on automatically generating mocking points for deterministic "record and replay" on industrial online microservices. To do that, we first identified system/internal/external state dependencies that may result in different behaviors of microservices in offline and online environments. Then we developed a static analysis that can generate mocking points based on the dependencies. We successfully deployed our work in Ant Group to daily handle hundreds of industrial microservices. With the generated mocking points by our analysis, the replaying achieves high success rate as well as high code coverage in testing.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *PLDI'20*. 91–105.

[2] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *DSN'11*. 403–410.

[3] Ant Group. 2021. SOFASTACK. https://github.com/sofastack

[4] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: Frameworks and caches, the elephants in the room. In *PLDI'20*. 794–807.

[5] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *ASE'14*. 79–90.

[6] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *FSE'15*. 155–165.

[7] Nipun Arora, Jonathan Bell, Franjo Ivančić, Gail Kaiser, and Baishakhi Ray. 2018. Replay without recording of production bugs for service oriented applications. In *ASE'18*. 452–463.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[9] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *SANER'16*, Vol. 1. 470–481.

[10] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *CCS'17*. 2329–2344.

[12] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. 2009. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL'09*. 114–126.

[13] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *ISSTA'07*. 196–206.

[14] et al. Clinton Begin. 2021. Mybatis. http://www.mybatis.org/

[15] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. *Replay debugging for distributed applications*. Ph. D. Dissertation. University of California, Berkeley.

[16] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *ICSE'13*. 72–81.

[17] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay.. In *OSDI'08*, Vol. 8. 193–208.

[18] Roman Haas, Rainer Niedermayr, Tobias Röhm, and Sven Apel. 2019. Poster: Recommending Unnecessary Source Code Based on Static Analysis. In *ICSE'19 Companion*. 178.

[19] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for android. In *OOPSLA'15*. 349–366.

[20] et al. Jeremy Landis. 2021. Ibatis. https://ibatis.apache.org/

[21] Rod Johnson. 2021. Spring. https://spring.io/

[22] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *FSE'20*. 1498–1508.

[23] Samer AL Masri, Sarah Nadi, Matthew Gaudet, Xiaoli Liang, and Robert W Young. 2018. Using static analysis to support variability implementation decisions in C++. In *SPLC'18*. 236–245.

[24] Mihir Mathur. 2020. *Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices*. Ph. D. Dissertation. UCLA.

[25] James W Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications.. In *NSDI'10*, Vol. 10. 159–174.

[26] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[27] Shaikh Mostafa and Xiaoyin Wang. 2014. An empirical study on the usage of mocking frameworks in software testing. In *SQIC'14*. 127–132.

[28] Oracle. 2021. GraalVM. https://www.graalvm.org/

[29] Chao Peng and Ajitha Rajan. 2020. Automated test generation for OpenCL kernels using fuzzing and constraint solving. In *GPGPU'20*. 61–70.

[30] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *ICSE'16*. 571–582.

[31] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. 107–112.

[32] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *ICSE'15*, Vol. 1. 598–608.

[33] João Pedro Gomes Silva. 2019. Debugging Microservices. (2019).

[34] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock? An empirical study on mocking practices. In *MSR'17*. 402–412.

[35] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems. *Empirical Software Engineering* 24, 3 (2019), 1461–1498.

[36] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices* 35, 10 (2000), 264–280.

[37] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated test generation for database applications via mock objects. In *ASE'10*. 289–292.

[38] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *FASE'13*. 210–225.

[39] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[40] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *ICSE'18 Companion*. 61–64.

[41] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *ASE'20*. 436–447.