MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests

Hengcheng Zhu*
Southern University of Science and Technology
Shenzhen, China
zhuhc2016@mail.sustech.edu.cn

Lili Wei*
The Hong Kong University of Science and Technology
Hong Kong, China
lweiae@cse.ust.hk

Ming Wen
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Yepang Liu†
Southern University of Science and Technology
Shenzhen, China
liuyp1@sustech.edu.cn

Shing-Chi Cheung†
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Qin Sheng
WeBank Co Ltd
Shenzhen, China
entersheng@webank.com

Cui Zhou
WeBank Co Ltd
Shenzhen, China
cherryzzhou@webank.com

ABSTRACT

In unit testing, mocking is popularly used to ease test effort, reduce test flakiness, and increase test coverage by replacing the actual dependencies with simple implementations. However, there are no clear criteria to determine which dependencies in a unit test should be mocked. Inappropriate mocking can have undesirable consequences: under-mocking could result in the inability to isolate the class under test (CUT) from its dependencies while over-mocking increases the developers' burden on maintaining the mocked objects and may lead to spurious test failures. According to existing work, various factors can determine whether a dependency should be mocked. As a result, mocking decisions are often difficult to make in practice. Studies on the evolution of mocked objects also showed that developers tend to change their mocking decisions: 17% of the studied mocked objects were introduced sometime after the test scripts were created and another 13% of the originally mocked objects eventually became unmocked. In this work, we are motivated to develop an automated technique to make mocking recommendations to facilitate unit testing. We studied 10,846 test scripts in four actively maintained open-source projects that use mocked objects, aiming to characterize the dependencies that are mocked in unit testing. Based on our observations on mocking practices, we designed and implemented a tool, *MockSniffer*, to identify and recommend mocks for unit tests. The tool is fully automated and requires only the CUT and its dependencies as input. It leverages machine learning techniques to make mocking recommendations by holistically considering multiple factors that can affect developers' mocking decisions. Our evaluation of *MockSniffer* on ten open-source projects showed that it outperformed three baseline approaches, and achieved good performance in two potential application scenarios.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software maintenance tools**; *Software testing and debugging*.

KEYWORDS

Mocking, unit testing, recommendation system, dependencies

ACM Reference Format:
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 *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416539

1 INTRODUCTION

Unit testing has been widely adopted to assure the quality of program units, namely classes, by testing them in isolation. In practice, a class under test (CUT) is commonly coupled with other classes in a program or its referenced libraries. These classes are the dependencies of the CUT and often participate in its unit tests. Mocking is a defacto mechanism to isolate the CUT from its dependencies in a test by simulating the behaviors of the dependencies using mocked objects [28]. It was reported that 23% of the Java projects with test scripts use mocking [32].

*This work was conducted when Hengcheng Zhu was a visiting student at HKUST (The Hong Kong University of Science and Technology). The first two authors contributed equally to this work.
†Yepang Liu and Shing-Chi Cheung are corresponding authors.

ASE '20, September 21–25, 2020, Virtual Event, Australia

To conduct effective unit testing using mocking, developers first need to make mocking decisions, i.e., *deciding which dependencies should be mocked*. However, it is non-trivial to make proper mocking decisions. A study showed that developers may change their initial mocking decisions during development [36]. 17% of their studied mocked objects were introduced sometime after the test scripts were created. Another 13% of the originally mocked objects eventually became unmocked. This suggests that the original mocking decisions were later considered improper by the developers. Making proper mocking decisions is challenging because: (1) Mocking decisions are unlikely to be made by considering only a single factor. In practice, developers may need to consider multiple factors holistically to make a mocking decision. (2) Mocking decisions are usually context-aware. Developers can make different mocking decisions for the same dependency when testing different CUTs according to the different usage scenarios of the dependency.

Inappropriate mocking decisions can lead to undesirable consequences. On the one hand, dependencies that should be mocked can be left unmocked in test scripts. Such *under-mocking* could result in the inability to isolate the CUT from its dependencies, which can seriously affect the efficiency of unit testing. The developers fixed this issue by mocking the real object [1]. On the other hand, dependencies that do not need mocking can be mocked by developers. Such *over-mocking* increases developers' burden on maintaining the mocked objects since they need to keep the behaviors of the mocked objects consistent with the real implementations. Inconsistencies between the mocked objects and the real implementations can cause spurious failures in testing. For example, in issue 16300 [5] of project Flink, the method `getID()` in mocked `ExecutionVertex` returns a null value and thus caused null pointer exceptions (NPEs) in test executions. However, in the real implementation of `ExecutionVertex`, the method `getID()` will never return `null`. In this case, the failure caused by NPEs does not reveal a real bug. Developers replaced the mocked `ExecutionVertex` with a real one to fix this issue. The evolution of the project code can further exacerbate the problem since developers need to update the mocked objects to catch up with the code evolution. If too many dependencies are mocked, it would be difficult for the developers to update the mocked objects in time.

Given the challenges in making proper mocking decisions, studies were conducted to find out the factors that affect mocking decisions. For example, Mostafa et al. [32] pointed out that production classes are more frequently mocked than library classes. Spadini et al. [36] categorized mocked objects and found that classes dealing with external resources are often mocked by developers. Marri et al. [31] revealed that file system APIs can be mocked to facilitate unit testing. These studies investigated the mocking practices and identified high-level and intuitive factors that can affect mocking decisions. In addition, these factors are all generic to CUTs without considering their interactions with the dependencies. With such advice, it is still difficult for developers to make proper mocking decisions when writing test scripts. Researchers have pointed out the need for automated mock recommendation techniques [31, 32]. Yet, none of the existing work has proposed such a technique. In fact, the high-level and project-generic characteristics of mocked dependencies identified by these studies cannot effectively guide the design of automated mock recommendation techniques.

This motivates us to conduct an empirical study to characterize mocked dependencies at the code level by analyzing API usages, data flows, control flows, etc. We aimed to identify those characteristics that can be automatically extracted via code analysis so that we can leverage them to build automated mock recommendation techniques. In our empirical study, we analyzed 10,846 test scripts of four large-scale open-source projects (such as Hadoop). When conducting the empirical study, we not only studied the characteristics of the dependencies themselves but also investigated their interactions with the CUTs in different test cases. We made several important observations that were not captured by existing studies. Specifically, we identified ten characteristics of mocked objects at the code level. We found that all of the ten characteristics can affect mocking decisions yet none of them is the sole determining factor. This provides evidence for the fact that mocking decisions are made by considering multiple factors, and thus we shall holistically consider different factors to recommend mocking decisions. We also observed that context-aware factors, which capture the interactions between dependencies and CUTs, are the most relevant to the mocking decisions. This indicates that an automated mocking decision recommendation technique should be context-aware, i.e., *considering the interactions between the dependencies and the CUTs*.

Based on our empirical findings, we further proposed a technique, *MockSniffer*, to recommend mocking decisions in unit testing. *MockSniffer* makes context-aware recommendations for dependencies of CUTs in unit testing. It takes a CUT and its dependencies as input and outputs a recommended mocking decision for each of the input dependencies. It also holistically combines various factors to suggest mocking decisions by leveraging machine learning techniques with features formulated from our empirical study findings. *MockSniffer* learns the knowledge of making mocking decisions from existing mocking practices and leverages the knowledge to recommend future mocking decisions.

In our evaluation, we trained and tested *MockSniffer* with 546k data entries of mocked and unmocked dependencies extracted from ten open-source projects. We compared the performance of *MockSniffer* with the generic mocking decision strategies adopted in existing studies. Our results show that *MockSniffer*, which performs context-aware mocking recommendations, can significantly outperform the baseline methods as shown by the Mann-Whitney U-Test[30]. We also evaluated *MockSniffer* under two potential application scenarios: (1) for mature projects, train *MockSniffer* with data extracted from historical releases of the same project to conduct cross-version mocking recommendation, and (2) for new projects, train *MockSniffer* with data extracted from other projects to conduct cross-project mocking recommendation. Our evaluation results showed that *MockSniffer* achieved good performance in both of these two application scenarios: it achieves an average F1-score of 69.40% and 70.77% for the two application scenarios respectively.

To summarize, this paper makes three major contributions:

- We conducted an empirical study based on 10,846 test scripts of four large-scale open-source projects and disclosed ten code-level characteristics of the mocking practices of real-world developers. We also validated our findings in a large-scale dataset consisting of 354k mocked and unmocked dependencies. In our study, we

```
1  public int countFiles() { // Production code
2    try{
3      return fileManager.scan().length;
4    } catch(IOException e) {
5      return 0;
6    }
7  }
8
9  public void test1() { // Test script
10   FileManager mgr = mock(FileManager.class);
11   when(mgr.scan()).thenReturn(new File[500]{});
12   UnderTest cut = new UnderTest(mgr);
13   assertEquals(500, cut.countFiles());
14 }
15
16 public void test2() { // Test script
17   FileManager mgr = mock(FileManager.class);
18   when(mgr.scan()).thenThrow(new IOException());
19   UnderTest cut = new UnderTest(mgr);
20   assertEquals(0, cut.countFiles());
21 }
```

Listing 1: Example Usage of Mocked Objects

observed mocking decisions are affected holistically by various factors, among which contextual features play an important role.

- We designed and implemented *MockSniffer*, the first automated technique to recommend mocking decisions for unit testing. Our evaluation of *MockSniffer* on open-source projects showed that *MockSniffer* significantly outperformed the mocking strategies adopted by existing studies and achieved good performance in two potential application scenarios.
- In our study, we have generated a large labeled dataset consisting of 546k data entries of test cases, dependencies, and CUTs. We released this dataset for public access to facilitate future research (https://doi.org/10.5281/zenodo.3783869).

2 BACKGROUND

In unit tests, mocked objects help decouple a CUT from its dependencies. Mocked objects are usually created by leveraging a mocking framework. Take `test1()` in Listing 1 as an example. The production code at line 3 involves disk I/O. To save effort in setting up the environment for testing, at line 11, developers create a mocked `FileManager` object using Mockito [8], a popular mocking framework. Then, the mocked object `mgr` directly returns a `File` array without accessing the disk (at line 11). This also speeds up test executions as disk I/O can be slow. Similarly, the production code at line 5 is not executed unless exceptions occur at line 3. To emulate the exceptional scenarios, in `test2()`, developers construct a mocked `FileManager` object and make it throw an exception directly when the method `scan()` is invoked.

Apart from mocked objects created with mocking frameworks, we also observed that developers can construct mocked objects by creating dummy classes that extend the concerned dependencies. For example, in the test script of project HBase, developers created a class `KeyProviderForTesting`, which is a subclass of the production class `KeyProvider`. They mentioned in the document that the class is to return a fixed secret key for testing. Instances of such classes created in the test scripts serve the same purpose as those mocked objects created with mocking frameworks.

Mocking has been widely used in unit test generation techniques. For example, Arcuri et al. [13] enhanced EvoSuite [22] by leveraging mocking to increase code coverage and reduce flaky tests. Alshahwan et al. proposed AUTOMOCK [12] to improve the performance of test case generation by mocking the environment. Tillmann et al. [38] proposed a symbolic-execution-based technique to generate mocked objects for unit testing. Although these studies found that mocking can facilitate test generation, they also reported that generated test cases with mocked objects can introduce spurious test failures (i.e., false alarms). The underlying reason is that there is no reliable mechanism to help decide which dependencies to mock during test generation. When making mocking decisions, these existing techniques have to resort to simple rules (e.g., all database and file system related dependencies should be mocked). Such mocking decisions contradict with the practices of real-world developers, who often mock only a small portion of dependencies (e.g., file system related dependencies are not always mocked) [32], and may result in substantial false alarms [15, 34]. To reduce such false alarms, existing studies proposed several strategies [12, 15], such as confining the values that can be returned by method calls on mocked objects. Although these strategies can help reduce false alarms, it would be better to mock only when necessary. In the following sections, we will study the mocking practices of developers and figure out the factors that can affect mocking decisions.

3 DATA COLLECTION

In order to understand the mocking practices adopted by developers, we constructed a dataset by extracting CUTs, their dependencies, and developers' mocking decisions from open-source projects. This dataset will enable us to study whether developers share similar practices when making mocking decisions. In this section, we present its construction process in detail.

3.1 Data Representation

Each entry in our dataset is a tuple: $< T, CUT, D, L >$ where T represents the test case, CUT represents the class under test, D represents the dependency (a class used in T, but not CUT), and $L \in \{mocked, unmocked\}$ is a label that represents whether D is mocked in the test case T. For example, `<TestCachingKeyProvider.testKeyVersion, CachingKeyProvider, KeyProvider, mocked>` is a data entry extracted from Hadoop [2]. It means that developers mocked the dependency `KeyProvider` in the test case `TestCachingKeyProvider.testKeyVersion` for CUT `CachingKeyProvider`. Such a data entry not only captures the developers' mocking decision on a dependency but also links dependencies to CUTs. We include the links in our dataset because the mocking decisions for the same dependency can vary across CUTs [36].

3.2 Subjects and Data Extraction

For constructing the dataset, we selected four actively-maintained open-source projects. Table 1 shows the information about these projects. These projects all use Mockito [8] or define dummy classes to construct mocked objects. As we can see from the table, the projects are large-scale. In the following, we explain our data extraction procedure.

Existing studies leveraged static analysis to identify mocked objects in test cases. However, such identification may be imprecise. For example, developers of Hadoop created a factory method [6] to create mocked objects of `EventWriter`. Depending on the value of

Table 1: Selected Projects

Project	Version	Files	LOC	Data Entries	Mocked
Hadoop	3.2.1	10,034	1.6M	325,335	14,771
Camel	3.1.0	18,245	1.3M	12,962	1,864
HBase	2.2.3	3,976	738k	11,990	1,093
Storm	2.1.0	2,354	282k	3,648	377
Total		34,609	3.9M	353,935	18,105

the field `mockHistoryProcessing` in the test class, the test cases may or may not create a mocked object of `EventWriter`. It is difficult for static analysis to precisely infer whether such objects in each test case are mocked or not. In our work, to obtain a more precise dataset, we leveraged dynamic analysis to identify mocked objects and extract data entries. We explain the main ideas below.

For each test case T, we first infer its class under test (i.e., the *CUT*), using naming heuristics. According to commonly adopted naming conventions, the name of a test class typically contains the name of the CUT (e.g., `TestMyClass` is a test class for `MyClass`). Hence, we can analyze the class name of T to infer *CUT*. Next, to analyze whether a dependency D is mocked in T. In this paper, we regard the non-CUT objects created during test case execution and passed to the CUT directly or indirectly as test dependencies. Such objects are usually passed via method calls on the CUT and its dependencies. Therefore, we instrumented all method call sites in T to log the exact type of each reference-type argument and the type of the corresponding formal parameter. We observed that mocked objects created with popular mocking frameworks (e.g., Mockito [8], EasyMock [3]) have special type names. For example, when using Mockito [8], the type names of the mocked objects are in the form `Foo$MockitoMock$xxx`, where `xxx` is a hash code.[1] Therefore, after executing T, we can analyze the logged information to infer whether an argument is a mocked object (i.e., determine the label L) via checking its type name and obtain the dependency D, which is the type of the corresponding formal parameter.

As mentioned in Section 2, developers may construct mocked objects by themselves rather than using a mocking framework. To include such mocked objects in our dataset, we also considered objects as mocked ones if they are instances of classes that (1) are defined in test scripts, and (2) extend a class in the production code. In this case, dependency D is identified as the production class being extended (i.e., `KeyProvider` in the example in Section 2).

While the above approach may miss some test dependencies (e.g., those specified via configuration files or assigned directly to a public field), it helped us collect 354k data entries from the four open-source projects after running 50k test cases. Such collected data entries are already sufficient for our empirical study.

4 EMPIRICAL STUDY

To identify the factors that can affect the mocking decisions, we conducted an empirical study on the projects based on the dataset extracted (Section 3). We aimed to derive a set of rules to capture the characteristics of the mocked objects via analyzing their code patterns. Such rules can further guide us to design automated techniques to help developers make mocking decisions.

4.1 Setup

We adopted a two-stage scheme when conducting the empirical study. In the first stage, we manually inspected a small subset of the dataset to devise code-level characteristics of the mocked objects. In the second stage, we formulated the code-level characteristics into rules and conducted an automated validation of these rules with a large-scale dataset, aiming to validate the derived characteristics.

Stage 1: Characteristics identification. In the first stage, we manually inspected 100 data entries in the dataset to identify characteristics of the dependencies that are mocked by developers. Specifically, we randomly sampled 25 data entries labeled as mocked from each of the four projects. For each sampled entry, we analyzed the source code of the dependency, the CUT, and the test script from which the data entry is extracted. We inspected the data entries using the open coding method [20] to identify common code-level characteristics (e.g., data flow, control flow, and API usage) of the mocked dependencies and their interactions with the CUTs. Although the sampling size is small, our identified characteristics can cover 94.82% (on average) of the mocking cases as shown in our evaluation (see the recall of Baseline #3 in Table 5).

Stage 2: Large-scale validation. In the second stage of the empirical study, we further validated the code-level characteristics of mocked dependencies identified in stage 1 using the entire dataset. Specifically, based on the manually-identified characteristics, we formulated several rules to automatically identify the data entries that exhibit the characteristics. For each of the rules, we applied it to all 354k data entries in our dataset to obtain a subset of data entries that match this rule. For each of the subsets, we computed its mock ratio, i.e., the proportion of entries labeled as mocked. We compared the mock ratio of each subset with the mock ratio of the entire dataset, which is 5.1%. The larger the difference in the mock ratio, the more likely the corresponding code-level characteristic can affect mocking decisions.

4.2 Results

Following the process described above, we made five observations and formulated ten rules (i.e., code-level characteristics) in stage 1. In the following, we will discuss our observations and the formulated rules. We will also present the mock ratio obtained in stage 2 for each data entry subset that matches each rule. The mock ratio is presented in the brackets after each rule.

Observation 1: Classes related to environment or concurrency are often mocked. In 51 of the 100 manually-inspected entries, the dependencies invoke APIs related to concurrency, networking, disk I/O, or APIs provided by online services (e.g., Amazon AWS, Microsoft Azure). These APIs can be slow to execute or exhibit inconsistent behaviors across different test runs. We formulated the following rules based on this observation.

- **Rule 1.1: Referencing environment-dependent or concurrent classes (11.8%).** Classes matching this rule call APIs related to the environment or concurrency. We manually built a list of such APIs (at class level) in JDK, including those related to networking, disk I/O, concurrency, database, etc. We found that these

[1] We also checked the pattern in other mocking frameworks in our implementation. We skip the details due to page limit.

APIs are frequently used in mocked classes but infrequently used in unmocked ones. This rule matches the data entries where the dependency references such APIs more frequently than average of all dependencies in our entire dataset. (For succinctness, we use "more than average" to describe where a certain metric computed with a subset is higher than the average value of that metric in our entire dataset.) Here, we consider both direct and transitive references to include the cases where these APIs are not called directly (i.e., called in the callees of the dependencies).

- **Rule 1.2: Encapsulating external resources (7.4%).** We also considered classes that encapsulate external resources, which usually implement certain interfaces. For example, classes encapsulating network connections usually implement the interface `Closable`. We manually built a list of such interfaces from the classes encapsulating external resources in our inspected dataset. This rule matches the data entries, in which the dependency implements interfaces in the list.
- **Rule 1.3: Calling `synchronized` methods (13.8%).** Classes performing concurrent operations would usually call `synchronized` methods. This rule matches the data entries where the dependency calls more `synchronized` methods than average.

As the numbers in the brackets show, these rules produced subsets of data entries with higher mocking ratios than that of the entire dataset (5.1%), indicating that dependencies matching these rules are more likely to be mocked. This is natural because such dependencies often complicate unit testing [36]. For example, to prepare the test environment for a class that makes network requests, developers need to set up a running server first. Besides, a class that performs concurrent operations may produce inconsistent results across different test runs due to inherent non-determinism of scheduling. Such inconsistency would make the test flaky [27]. Therefore, developers would like to mock such classes to avoid these problems.

Observation 2: Complicated classes are often mocked. We found that in 23 of the 100 entries, the dependencies are complicated in terms of the number of fields or referenced classes. Take the class `Configuration` in Hadoop as an example. It has 113 methods and 23 fields. Also, it imports 78 classes from other packages. To create a real instance of such a class in a test, developers need to initialize all its dependencies, many of which may not be relevant to the test. As a result, developers mocked this class in test cases. We designed the following rules to find such complicated classes.

- **Rule 2.1: Excessive fields (8.8%).** Classes with lots of fields can represent complex data structures. Such classes are not easy to set up in testing and thus are likely to be mocked. This rule matches the data entries where the dependency has a larger number of fields than the average of the whole dataset.
- **Rule 2.2: A large number of dependencies (7.9%).** Classes with a large number of dependencies are also difficult to set up in testing and are likely mocked by developers. This rule matches the data entries where the dependency references (directly and transitively) more classes than average.

The two rules also produced subsets with higher mock ratios, which shows that developers usually mock complicated classes.

```
1 // === Production code ===
2 if(endpointConfig.isOverWrite()){
3   oStream.info.getFileSystemn().delete(...);
4 } else {
5   throw new RuntimeCamelException(...);
6 }
7
8 // === Mock setup ===
9 when(endpointConfig.isOverWrite()).thenReturn(false);
```

Listing 2: Example of Observation 4 in Project Camel

Observation 3: Non-concrete types are usually mocked. We observed that the dependencies in 43 of the 100 entries are non-concrete types, which motivated us to design the following rule.

- **Rule 3.1: Non-concrete types (8.3%).** The rule matches data entries where the dependency is an abstract class or an interface.

The rule increased the mocking ratio by 60.8% (from 5.1% to 8.2%), indicating that developers often mock such non-concrete types since they cannot be instantiated directly. Instead, developers have to choose an implementation or create a mocked version.

Observation 4: Dependencies affecting the runtime control flows of methods in CUTs are often mocked. We found that developers often mock a class and set different return values for its methods to cover different branches in CUT. Listing 2 shows an example in the project Camel when testing the class `HDFSOutputStream`. Here, `endpointConfig` is the dependency, and the return value of its method `isOverWrite()` is used for a branch condition in the CUT. Developers mocked the dependency, stubbed the return value with `false`, and asserted that a `RuntimeCamelException` would be thrown in the test script, to cover the branch at line 5. In stage 1, we found that 33 of the 100 entries have this characteristic, which motivated us to design the following rules.

- **Rule 4.1: Affecting CUT's runtime control flows via return values (10.9%).** Just like the example above, if the return value is used for a branch condition, developers can mock the dependency, provide different return values to cover different branches. This rule matches a data entry if the return value of a method call on the dependency is used in branch conditions in the CUT.
- **Rule 4.2: Affecting CUT's runtime control flows via exceptions (12.6%).** Similar to the case in Rule 4.1, developers can mock a dependency to throw an exception to test the exception handler. This rule matches a data entry if the CUT catches an exception thrown by the dependency.

These rules produced subsets with much higher mock ratios, indicating that classes affecting the runtime control flows of methods in CUTs are frequently mocked.

Observation 5: Dependencies capturing the internal behaviors of the CUTs are often mocked. The verification feature of mocking frameworks is widely used by developers. Developers often test whether the CUT is implemented correctly by enforcing assertions on the method invocations on the dependencies. Listing 3 shows an example in the project Storm when testing the class `RocketMqBolt`. In the CUT, the method invocation to `send()` on the dependency `producer` is dominated by two `if` conditions. Developers mocked the dependency and asserted that the method is called, to test whether the branch conditions in the method `execute()` (line 1 and 4) were designed correctly. We found that 13 of the 100 entries have this characteristic and we designed the following rules:

```
if(batch) { // Production code in method execute()
  //...
} else {
  if(async)
    producer.send(prepareMessage(input), ...);
}

// Test script
rocketMqBolt.execute(tuple);
verify(producer).send(any(Message.class), ...);
```

Listing 3: Example of Observation 5 in Project Storm

- **Rule 5.1: Conditional invocation (12.5%).** Similar to the example, dependencies whose methods are called conditionally are often mocked. Therefore, this rule matches a data entry if any method invocation to the dependency is dominated by a branch condition or an exception handler in the CFG of the CUT.
- **Rule 5.2: Capturing intermediate results (11.4%).** Arguments used to call a method of the dependency are usually the intermediate results produced by the CUT, which can be different based on different inputs for the CUT. In practice, developers may mock a dependency and capture the arguments passed to its methods. By checking the captured argument values, developers can test whether the intermediate results are correct, and thus validate the implementation of the CUT. This rule matches a data entry if an argument at a call site to the dependency is data-dependent [11] on the parameters of a CUT method.

These two rules also produced subsets with much higher mock ratios, indicating that dependencies capturing the internal behaviors of the CUTs are often mocked.

Table 2 aggregates the mock ratios in the subsets produced by applying each of the ten rules. As we can see from the table, all the mock ratios of the subsets are higher than that of the whole dataset. This indicates that all the observations we made in our manual inspection are generalizable and our identified code-level rules can characterize mocked dependencies. In particular, there are five subsets whose mock ratios are more than 100% higher than that of the whole dataset. We found that the rules that were used to extract them can be divided into two groups:

(1) **API usages.** Rule 1.1 and Rule 1.3 model the API usages of the dependencies, which reflect the behaviors of the dependencies.
(2) **Interactions.** Rule 4.1, Rule 4.2, Rule 5.1, and Rule 5.2 model the interactions between the CUT and the dependencies showing how CUTs can affect mocking decisions.

The high mock ratios indicate that these two types of factors are more likely to affect mocking decisions.

5 MOCKSNIFFER

Our empirical study yielded five observations with ten code patterns of the mocked dependencies. These findings also showed that mocking practices adopted by developers share some common characteristics, which can be leveraged to guide future mocking decisions. Therefore, based on the empirical findings, we propose *MockSniffer*, a fully automated technique that recommends mocking decisions for developers by learning from existing practices. Specifically, *MockSniffer* takes the CUT and its dependencies as input and suggests a binary mocking decision for each of the dependencies. We built *MockSniffer* based on binary classification machine learning techniques. The mechanism of binary classification is in line with the nature of our targeted problem: providing the characteristics of the dependency and the CUT, to predict whether the dependency should be mocked. We implemented *MockSniffer* on various machine learning models and trained them with the mocking practices extracted from existing test cases. *MockSniffer* can thus recommend mocking decisions by learning from existing practices.

Table 2: Mock Ratios by Applying the Rules

Rule	Matches	Mocked	Ratio	Comparison
Rule 1.1	49,789	5,874	11.8%	+131.4%
Rule 1.2	5,669	419	7.4%	+45.1%
Rule 1.3	33,356	4,611	13.8%	+170.6%
Rule 2.1	86,469	7,620	8.8%	+72.5%
Rule 2.2	104,416	8,204	7.9%	+54.9%
Rule 3.1	122,471	10,197	8.3%	+62.7%
Rule 4.1	74,510	8,119	10.9%	+113.7%
Rule 4.2	60,193	7,565	12.6%	+147.1%
Rule 5.1	64,488	8,089	12.5%	+145.1%
Rule 5.2	56,080	6,380	11.4%	+123.5%
Dataset	353,935	18,105	5.1%	

5.1 Feature Engineering

Table 3 shows the 16 features used in the machine learning model of *MockSniffer*. These features are either designed based on our observations in Section 4.2 or adopted from the empirical findings in existing studies [32, 36] that can be directly applied to the code. They fall into four categories.

Contextual information. This includes RBFA, EXPCAT, CONDCALL, AFPR, and CALLSITES. The first four are derived directly from Observation 4 and 5 in Section 4.2. These features capture the interactions between the CUT and the dependency by matching certain patterns. However, there would be some scattered patterns that lead to mock but not observed by us since each single of them doesn't appear frequently. As a complement, we designed the feature CALLSITES, which counts the call sites to the dependency in the CUT, with the heuristic that such scattered patterns would appear when there are abundant call sites.

API usage. UAPI, TUAPI, UINT, and SYNC model the API usage of the dependency and thereby reflect its behaviors (i.e., whether it is related to the environment or concurrency). They are derived from Rule 1.1 to 1.3. We split Rule 1.1 into UAPI and TUAPI, which counts the direct and transitive references to the APIs in Rule 1.1. Our heuristic is that dependencies referring to such APIs directly and transitively may have different chances to be mocked.

Complexity. DEP, TDEP, and FIELD measure the complexity of the dependency and thus fall into this category. These features count the direct and indirect references to other classes, as well as the fields of the dependency. While FIELD is derived directly from Rule 2.1, we split Rule 2.2 into DEP and TDEP with the heuristic that direct and indirect dependencies can affect mocking decisions to a different extent.

Class meta-properties. ABS, INT, ICB, and JDK are related to the meta-properties of the dependency itself. They are adapted from Rule 3.1 and existing studies [32, 36].

For features derived from the rules under each observation, we removed the thresholds and use numerical values since we want

the classifier to learn the thresholds. By using these features, *MockSniffer* can holistically consider multiple factors and make wise mocking decisions.

5.2 Classification Model

Feature extraction. To extract features from training or testing datasets, we built static analyzers for each feature on top of Soot [40]. Each static analyzer takes a specified CUT, dependency, and the byte code of the whole project as input and output the corresponding value of the feature. Table 3 shows the descriptions of our features. For indicator features like ABS, INT, and JDK, the analyzers can be implemented by loading the dependency and check the corresponding properties. For example, the analyzer of ABS simply loads the dependency and checks whether it is an abstract class. The analyzers of EXPCAT, CONDCALL, and those numerical features can be implemented by searching in the call graph by starting from the test method (i.e., the test case T) and counts the occurrences of certain patterns in the code. Take EXPPCAT as an example, the analyzer traverses the call graph from the test method and counts the call sites to the dependencies that are wrapped by a `try . . . catch` clause in the methods of the CUT.

Standardization. Training with a standardized dataset can achieve better performance on many classifiers. Data standardization is applied independently on each numerical features. Typically, data standardization subtracts the mean value and scales the values to unit variance. However, our dataset contains many outliers, which may bias the mean value and variance calculation. To address this problem, we subtracted the median from the dataset and then scale it according to the interquartile range [39].

Under-sampling. As shown in Table 1, the datasets of mocked and unmocked dependencies are often imbalanced. The entries labeled as mocked only account for a very small portion of the whole dataset (e.g., only 5.1% in Table 1). Training classification models directly with such imbalanced datasets can induce bias in the prediction results and result in poor prediction performance [41]. To address this issue, we adopted the random under-sampling technique to obtain a balanced dataset, which is simple but performs no worse than other under-sampling techniques [29]. Specifically, we randomly removed the instances labeled as unmocked, which account for the majority, until the dataset contains the same number of mocked and unmocked data entries.

Model training. *MockSniffer* can be implemented on the top of various classification models and different models may achieve different performance. We compared the performance of different models with the data extracted from projects in Table 1. We trained classification models and performed 10-fold cross-validation using Decision Tree [19], Naive Bayes [42], Ada Boosting [23], Gradient Boosting [24], Random Forest [18], and Support Vector Machine [17]. When training with Random Forest and Decision Tree, we set the maximum tree depth to `sqrt` to prevent overfitting. For other models, we used the default settings provided by scikit-learn [33]. As shown in Figure 1, the accuracies of these models range from 63.33% to 78.81%, where Gradient Boosting is the best. Therefore, we used Gradient Boosting as the default classifier. Since there is no ground truth on whether a mocking decision is correct, we used the mocking decisions made by developers in our dataset as target variables when training. We expect that the number of incorrect mocking decisions made by developers is small. The chance that such incorrect decisions significantly bias the model is low.

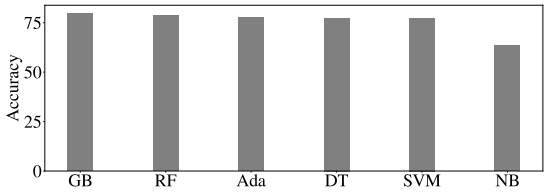

Figure 1: Accuracy of Classification Models

6 EVALUATION

In this section, we present our evaluation of *MockSniffer*. We aimed to explore the following research questions:

- **RQ1 (The effectiveness of *MockSniffer*):** *Can MockSniffer effectively recommend mocking decisions and outperform exiting mocking strategies? Does machine learning help make better mocking decisions?*
- **RQ2 (The effectiveness of a single feature):** *Which features are the most relevant to mocking decisions? Can MockSniffer effectively make mocking decisions based on every single feature?*
- **RQ3 (Potential application scenarios):** *What are the potential application scenarios of MockSniffer? How does MockSniffer perform in these scenarios?*

We conducted three studies to answer each of these research questions. We compared the performance of *MockSniffer* with three baselines, investigated the effectiveness of each feature, and evaluated *MockSniffer* in two potential application scenarios. In our experiments, we used the mocking decisions made by developers as the ground truth. Since *MockSniffer* performs binary classification to recommend mocking decisions, we adopted the metrics widely used for evaluating binary classifiers: accuracy, precision, recall, and F1-score to evaluate and compare the results of *MockSniffer*.

6.1 Evaluation Subjects

Our evaluation of *MockSniffer* is based on the four projects listed in Table 1. To further evaluate whether our findings on these four projects can be generalized to other projects, we selected six more large-scale, actively maintained projects, where mocking is often used in their test cases. Table 4 lists these projects. The methodology to collect the six projects is the same as that adopted in Section 3. Finally, we combined the six projects with the projects in Table 1 and formed a set of ten projects for evaluation. We further extracted data entries for the additional subjects by adopting the process described in Section 3 and then extracted features for each data entry leveraging the methodology described in Section 5.1. After these steps, we got an evaluation dataset containing 546k entries, where 31k of them are labeled as mocked.

6.2 Baselines

Since there are no existing automated mocking decision techniques, we compared *MockSniffer* with three baselines derived from existing studies, mocking strategies adopted by existing test generation techniques, and our empirical findings.

Table 3: Features Used by *MockSniffer*

Feature	Source	Description
UAPI	Observation 1	# Direct references to the APIs in the list mentioned in Rule 1.1
TUAPI		# Transitive references to the APIs in the list mentioned in Rule 1.1
UINT		Indicates whether the dependency implements any interfaces in the list mentioned in Rule 1.2
SYNC		# Call sites to `synchronized` methods by the dependency
DEP	Observation 2	# Classes referenced by the dependency directly
TDEP		# Classes referenced by the dependency transitively
FIELD		# Fields in the dependency
ABS	Observation 3	Indicates whether the dependency is an abstract class
INT		Indicates whether the dependency is an interface
RBFA	Observation 4	# Call sites to the dependency whose return value is used for branch conditions in CUT
EXPCAT		# Call sites to the dependency that are surrounded by a `try. . . catch` structure in the CUT
CONDCALL	Observation 5	# Call sites to the dependency in the CUT that are dominated by a branch condition or an exception handler
AFPR		# Call sites in the CUT whose arguments have data dependencies on parameters
CALLSITES	Observation 4 and 5	# Call sites to the dependency in the CUT
ICB	Existing study [32]	Indicated whether the dependency is defined in the production code
JDK	Existing study [36]	Indicates whether the dependency is a type provided by the JDK

Table 4: Extra Projects for Evaluation

Project	Version	Files	LOC	Data Entries	Mocked
Flink	1.10.0	8,828	899k	86,141	6,719
Hive	3.1.2	5,990	1.3M	23,368	1,490
CXF	3.3.5	7,293	672k	22,550	1,489
Druid	0.17.0	4,556	596k	45,332	1,869
Dubbo	2.7.6	2,210	166k	8,623	758
Oozie	5.2.0	1,388	191k	5,539	278
Total		30,125	3.8M	191,553	12,603

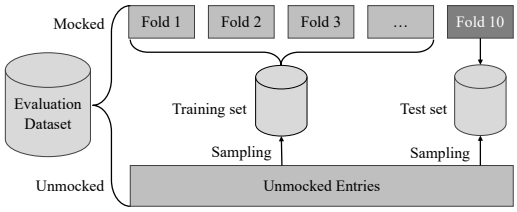

Figure 2: Cross Validation with Sampling

Baseline #1: Existing heuristics. Existing studies yielded empirical findings on the usage of mocking. We constructed a rule-based approach from the empirical findings that can be directly applied to the code. Specifically, it consists of three rules:

(1) It does not mock JDK classes since they are often not mocked [36].
(2) It mocks all the interfaces since developers prefer mocking interfaces over using their implementations [36].
(3) It mocks all the classes in the codebase since developers mock more classes in the codebase than in the libraries [32].

Specifically, Baseline #1 applies the rules as ordered above, and recommends to mock a dependency if it matches any of these rules. If none of the rules matches, it suggests not to mock the dependency.

Baseline #2: EvoSuite mock list. EvoSuite [22] contains a list [4] of dependencies to mock. As such, we built a simple rule-based strategy by mocking all the dependencies in the list. This is a conservative baseline since it only considers a limited number of JDK classes as dependencies to mock.

Baseline #3: Empirical rules. Our empirical study distilled five observations with ten rules to guide the mocking decisions. These code-level rules can be detected automatically. We designed an aggressive baseline, which would decide to mock if any of the rules in Section 4.2 match.

6.3 RQ1: The Effectiveness of *MockSniffer*

Experiment setup. To evaluate the effectiveness of *MockSniffer*, we performed intra-project prediction on the ten projects. We split the entries labeled as mocked into ten folds, chose one fold for testing, and the others for training. Figure 2 illustrates the data preparation process. By this mechanism, each of the entries labeled as mocked appeared once in the test set, which makes the performance score more objective. As mentioned before, we performed under-sampling for the unmocked entries in the dataset. To minimize the influence on the credibility of the performance scores, we repeated the sampling process for 100 times and reported the average performance scores of 1,000 (10 folds × 100 samplings) runs. After that, we applied the three baselines on the evaluation dataset and compared their performance with *MockSniffer*.

Results. As shown in Table 5, on average, *MockSniffer* correctly distinguished 85.42% of the mocking decisions. For the instances it predicted as mocked, 83.95% of them are true positives, which covers 87.89% of the instance labeled as mocked. *MockSniffer* outperformed the three baselines by large margins on the ten projects in terms of accuracy, precision, and F1-score. We also performed the Mann-Whitney U-Test [30] to evaluate the result differences. The p-values ranged from 0.00009 to 0.0086, which showed that *MockSniffer* outperformed the baselines significantly ($p < 0.01$).

MockSniffer performed better than the first two baselines due to the following reasons. First, *MockSniffer* takes into account the contextual information, namely the interactions between the CUT and the dependency while neither of the baselines captures such information. Therefore, it can distinguish between different mocking decisions on the same dependency with different CUTs. Besides, *MockSniffer* can cover the scenarios where the dependency does not match the rules in the first two baselines but are mocked for other reasons (e.g., mock to increase the branch coverage just like

Table 5: Performance of *MockSniffer* and the Baselines

Metric (%)	Accuracy				Precision				Recall				F1-Score			
Approach*	**MS**	B#1	B#2	B#3	**MS**	B#1	B#2	B#3	**MS**	B#1	B#2	B#3	**MS**	B#1	B#2	B#3
Hadoop	82.90	64.46	48.71	63.30	80.36	60.01	13.93	57.86	87.09	86.71	0.50	97.92	83.59	70.93	0.96	72.74
Flink	82.92	68.58	48.90	54.64	81.32	62.54	8.89	52.46	85.51	92.68	0.24	98.88	83.35	74.68	0.46	68.55
Hive	84.36	58.52	49.90	63.94	84.46	58.52	47.06	58.82	84.32	58.52	1.61	92.95	84.33	58.52	3.11	72.05
Camel	79.52	64.91	48.52	53.40	76.19	62.28	6.35	51.80	86.12	75.58	0.22	97.74	80.80	68.29	0.42	67.71
CXF	83.05	59.37	48.82	54.73	84.67	59.48	15.69	52.60	80.83	58.83	0.54	95.84	82.65	59.15	1.04	67.92
Druid	83.41	64.50	49.06	52.70	83.53	66.75	2.70	51.50	83.38	57.78	0.05	92.78	83.39	61.94	0.10	66.23
HBase	86.58	75.28	49.30	54.61	84.39	68.87	20.00	52.47	89.89	92.26	0.47	98.03	87.00	78.87	0.91	68.35
Dubbo	88.55	65.45	49.14	54.55	85.75	69.07	23.96	52.49	92.71	55.94	0.79	95.78	89.03	61.81	1.53	67.82
Oozie	91.14	64.24	46.58	52.34	91.12	61.35	0.00	51.40	91.57	76.98	0.00	85.61	91.17	68.28	0.00	64.24
Storm	91.77	52.74	49.32	52.83	87.75	53.12	0.00	51.58	97.48	46.61	0.00	92.68	92.27	49.65	0.00	66.27
Average	85.42	63.81	48.82	55.70	83.95	62.20	13.86	53.30	87.89	70.19	0.44	94.82	85.76	65.21	0.85	68.19

* MS, B#1, B#2, and B#3 stands for *MockSniffer* and the three baselines respectively.

the example in Listing 2). Second, *MockSniffer* considers the behavior of the dependencies by analyzing their invoked methods. Baseline #2 mocked all the classes in the EvoSuite mock list, which contains the classes that are related to the environment or concurrency. However, the list contains only classes in JDK. Non-JDK classes invoking APIs related to environment or concurrency are not included. As a result, Baseline #2 suffered from a low recall. In comparison, *MockSniffer* can identify such cases by also analyzing the methods invoked by the dependencies and identifying transitive references to APIs related to the environment or concurrency.

Baseline #3 outperformed *MockSniffer* in terms of recall. This is because baseline #3 decided to mock the dependency when any of the rules in Section 4.2 match. Although such an aggressive approach can identify most of the mocked instances (94.82% recall on average), it would mispredict the dependencies that should not be mocked as mocked, and thus suffered from low precision (53.30% on average). The comparison between *MockSniffer* and Baseline #3 shows that it is not enough to simply apply the rules in Section 4.2 to recommend mocking decisions. To make better mocking decisions, we need to combine these rules holistically.

Answer to RQ1: *MockSniffer* outperformed existing mocking strategies. Machine learning can help make better mocking decisions.

6.4 RQ2: The Effectiveness of a Single Feature

MockSniffer uses multiple features together to make mocking decisions. As such, we want to investigate how relevant is each feature to mocking decisions, as well as the effectiveness of *MockSniffer* by using only that feature.

Experiment setup. The experiment consisted of two steps. First, we studied the relevance of each feature to the mocking decisions by performing the Chi-squared test [25]. Then, we ran modified versions of *MockSniffer* trained with each single feature to investigate how each feature contributes to the performance of *MockSniffer* with a process similar to that in Section 6.3.

Results. Table 6 shows the results of the Chi-squared test. For most of the features, the Chi-squared statistic values are larger than 100, indicating that these features are relevant to the mocking decisions. Also, features describing the interactions between the CUT and the dependency (e.g., EXPCAT, CONDCALL) and features reflecting the API usages of the dependency (e.g., TUAPI, SYNC) achieved the highest Chi-squared statistic values, indicating that these two factors are more relevant to mocking decisions.

Table 6: Chi^2 Statistics and Performance of Intra-project Prediction Using Single Feature

Metric (%)	Chi^2	Accuracy	Precision	Recall	F1-Score
ABS	2394.05	60.56	59.71	68.82	62.80
AFPR	4835.57	56.76	63.99	39.54	43.31
CALLSITES	18996.25	57.72	63.94	49.31	49.11
CONDCALL	20728.24	58.53	64.55	45.56	49.15
DEP	1490.05	71.31	70.90	73.43	71.57
EXPCAT	30968.33	58.42	70.40	34.27	41.84
FIELD	8.46	67.77	64.56	80.74	71.19
ICB	1281.75	60.14	62.34	55.70	56.57
INT	2506.90	59.05	61.41	55.13	56.14
JDK	3103.92	59.68	55.72	96.30	70.52
RBFA	3455.96	58.80	65.39	47.27	50.47
SYNC	18120.28	54.17	72.80	40.33	36.62
TDEP	2349.46	68.91	70.69	65.67	67.37
TUAPI	117270.05	65.73	71.53	66.33	63.41
UAPI	1035.86	62.92	73.84	52.66	55.36
UINT	60.01	51.15	59.41	61.71	43.68
Average		60.73	65.70	58.30	55.57

In addition, Table 6 presents the average performance of *MockSniffer* on the ten projects by using each of the features. By using each of the features, *MockSniffer* achieved an average accuracy, precision, recall, and F1-score of 60.73%, 65.70%, 59.30%, and 55.57%, respectively. The performance is much lower than that using all the features, which shows that, by taking multiple factors into account, *MockSniffer* can suggest better mocking decision than considering a single one. This is because there is more than one factor that drives developers to mock, but each of these features took just one of the factors into account. Instead, *MockSniffer* combines these scattered factors together and considers them holistically, and thus, can suggest better mocking decisions.

Answer to RQ2: Context-aware and API usage related features are the most relevant to mocking decisions, but only considering each single of them is not enough.

Table 7: Performance of *MockSniffer* in Potential Application Scenarios

Metric (%)	Accuracy		Precision		Recall		F1-Score	
Settings*	CVP	CPP	CVP	CPP	CVP	CPP	CVP	CPP
Hadoop	75.20	75.84	75.08	73.06	73.74	79.46	74.68	76.13
Flink	79.79	74.12	75.39	69.76	78.95	89.64	79.37	78.46
Hive	69.57	70.41	65.90	74.60	60.96	48.21	65.50	58.57
Camel	72.33	66.72	67.11	66.51	73.31	68.94	72.49	67.70
CXF	61.59	71.84	71.56	73.46	40.00	67.55	51.01	70.37
Druid	68.75	66.39	66.50	68.37	56.90	61.43	64.33	64.71
HBase	83.82	77.11	76.42	75.00	87.02	79.26	84.25	77.07
Dubbo	68.33	71.24	66.68	72.22	44.76	54.32	56.93	61.99
Oozie	64.11	69.74	73.92	80.72	30.67	62.93	44.61	70.68
Storm	73.29	70.27	71.16	72.86	60.02	67.48	66.99	70.06
Average	71.68	70.97	78.82	72.65	60.63	67.92	66.02	69.57

* CVP and CPP stands for cross-version and cross-project prediction, respectively.

6.5 RQ3: Potential Application Scenarios

The ultimate goal of *MockSniffer* is to recommend mocking decisions in real-world projects during software development. As such, we designed experiments to investigate the effectiveness of *MockSniffer* in potential application scenarios.

Experiment setup. When using *MockSniffer* to recommend mocking decisions on a project, there are two potential scenarios.

(1) **Cross-version prediction (CVP).** For mature projects, developers can train *MockSniffer* with the data extracted from the historical releases of the project, and recommend mocking decisions for the new test cases in subsequent releases. To evaluate *MockSniffer* in this scenario, we collected five consecutive releases of the ten subject projects. For each release, we extracted data entries and their corresponding feature leveraging the same process as that described in Section 3 and 5. After that, we performed incremental predictions on each of the projects. For each release, we trained *MockSniffer* with the dataset extracted from its prior releases, and predict on those newly added instances in the current release (i.e., data entries that are not in the dataset extracted from prior releases). Since the number of newly added instances between each pair of releases varies, we reported the weighted average of the performance scores:
$$\text{Score} = \sum_i \text{Score}_i \times \#\text{new instances}_i / \sum_i \#\text{new instances}_i$$
in which i refers to the i^{th} release of the project used in the experiment.

(2) **Cross-project prediction (CPP).** For new projects, their historical mocking practices are insufficient to train *MockSniffer*. As such, developers can train *MockSniffer* with the data extracted from other projects (e.g., the vast open-source code repositories). To evaluate *MockSniffer* in this scenario, with the evaluation dataset, we picked the data entries from one project for testing and used the remaining for training. We repeated this procedure on each of the ten projects.

As mentioned in Section 5, we performed under-sampling when preparing the dataset, to maintain the reliability of the performance scores. We repeated the sampling process for 100 times and reported the average scores of the 100 runs.

Results. Table 7 shows the performance scores of *MockSniffer* in the two scenarios. The accuracy achieved by *MockSniffer* ranged from 61.59% to 83.82%, and on average, it made 71.68% and 70.97% correct decisions under the two settings, respectively. The precision in cross-version prediction (78.82% on average) is higher than that in cross-project prediction (72.65%). This is because the mocking decisions within the same project share more similarities. Existing mocking practices in the historical releases of a project are likely to be adopted by the test cases in its subsequent releases.

However, the recall of cross-version prediction (60.63%) is lower than that in cross-project prediction (67.92%). This is because developers may adopt new mocking practices that do not exist in the current project, but such practices may be adopted by other projects. In this case, cross-project prediction can transfer the knowledge from one project to another, and thus help developers make proper mocking decisions. For example, *MockSniffer* suffered from extremely low recall when performing cross-version prediction on the project Oozie (30.67%). We manually inspected the prediction process and found that low recall happened when predicting on Oozie 5.0 by learning from Oozie 4.2 and 4.3. The reason is that significant changes took place from Oozie 4.x to 5.x, for example, the workflow graph generator was completely rewritten and the Oozie launcher was moved from MapReduceMapper to YARN application master [21]. Developers also adopted new mocking practices in the newer versions. Due to such great changes, the knowledge learned from Oozie 4.x could not capture mocked object usages in Oozie 5.x In this case, cross-project can help increase the recall by leveraging the knowledge learned from other projects. Specifically, by performing cross-project prediction, *MockSniffer* achieved a recall of 62.93% on project Oozie, which is 105.18% higher than that in cross-version prediction.

Answer to RQ3: *MockSniffer* can be applied in two application scenarios (cross-version prediction and cross-project prediction). In these application scenarios, *MockSniffer* can effectively recommend proper mocking decisions by learning from historical mocking practices or mocking practices of other projects.

7 DISCUSSION

7.1 Threats to Validity

In the data collection process, we inferred the name of CUTs from the class name of the test cases according to a widely adopted naming convention. We may fail to identify CUTs for those test cases that do not follow such a naming convention. In addition, we focused on method arguments when extracting dependencies. This may miss some dependencies that are not method arguments (e.g., those specified via test configuration files and those assigned directly to a public field). Although our dataset does not include all the dependencies used in unit tests, it is large enough for our study.

The selected subject projects use only two mocking frameworks Mockito [8] and EasyMock [3]. There are also other mocking frameworks such as jMock [7]. Since there are differences in the functionalities provided by these frameworks, developers may have slightly different practices when using them. Such differences could not be covered by our study. However, according to a prior study [32], Mockito and EasyMock are the two most popular mocking frameworks. They are used by 70% and 20% of 5,000 randomly sampled

projects, respectively. As such, our study results can be applied to most of the Java projects.

When learning from existing mocking practices, *MockSniffer* assumes that the mocking decisions made by the developers are correct. However, developers can also make improper mocking decisions and change their decisions. To address this threat, we trained *MockSniffer* with 546k data entries extracted from ten different projects. The large number of training data entries can mitigate the problem induced by the small portion of improper mocking decisions in the dataset.

7.2 Future Work

Unit tests generation with mock recommendation. Some existing studies can generate unit tests with synthesized mocked objects [12, 13, 26]. However, these studies cannot select which dependencies to mock. Directly using such techniques can induce problems of over-mocking since they choose to mock all the dependencies of classes under test. In the future, we plan to combine our technique with these existing studies to generate unit tests with properly-selected mocked dependencies.

Generalizing *MockSniffer* to other languages. In this paper, we studied the mocking practices in Java and proposed *MockSniffer* to recommend mocking decisions. Besides Java, mocking is also widely used in projects written in other programming languages. In the future, we plan to further extend *MockSniffer* to recommend mocking decisions for projects in other languages such as C# and Javascript. Mocking frameworks like Moq [9] and Sinon.js [10] are frequently used in C# and JavaScript projects. Recommending mocking decisions for C# can be similar to Java since C# and Java share similar programming paradigms. In comparison, the language features of Javascript are different from that of Java. Such different language features can induce different mocking practices. In the future, we plan to also study the mocking practices of projects in different languages and generalize *MockSniffer* to recommend mocking decisions for them.

8 RELATED WORK

Empirical studies on mocking. Studies were conducted to analyze the usage of mocking as well as understanding the intentions behind the mocking decisions. Marri et al. [31] were among the first to analyze the use of mocked objects in testing file-system-dependent software and highlighted the need for automated identification of APIs that need to be mocked (i.e., automated techniques to recommend mocking decisions). Mostafa et al. [32] analyzed the use of four popular mocking frameworks in 5,000 open-source projects. They found that mocking frameworks are used in 23% of the projects that have test code, and developers usually mock a small portion of the dependencies. Spadini et al. [35] studied the usage of mocked objects in three open-source projects and one industrial system and distilled that developers usually mock dependencies that make testing difficult. More recently, Spadini et al. studied the evolution of mocked objects [36]. Even though the need for automated techniques to recommend mocking decisions was proposed early by Marri et al. [31], none of these existing studies produce such an automated technique. The existing studies either provided statistical evidence to show the popularity of mocking or identified general characteristics of mocking practices. In comparison, our study identified specific code-level characteristics of mocked objects. Based on such code-level characteristics, we were able to propose the first automated technique *MockSniffer* to recommend mocking decisions by leveraging machine learning techniques.

Automatic test generation with mocking. Another line of related work aimed to improve test generation techniques with the help of mocking [12, 13, 15, 26]. For example, Arcuri et al. [13] extended EvoSuite [22], a Java test generation tool by adding mocked objects to its generated test cases. By mocking the environmental interactions in the test cases, their technique improved the code coverage and reduced the number of unstable tests by more than 50%. Other studies used mocking to simulate the environment operations to control the environment-related dependencies. Taneja et al. proposed MODA [37] to generate tests for database applications by mocking database operations. This approach is also adopted to handle interactions with networking [14] and web services [16, 43]. While these work leveraged mocking to improve test generation techniques, they selected the dependencies to mock merely based on single factors without considering the interactions between the dependency and different CUTs. As a result, these approaches can select the same dependencies to mock even for different projects. However, as shown in our empirical study(Section 4.2), contextual information is a key factor that affects mocking decisions. Without taking such contextual information into account, these techniques can generate test cases with the problem of over-mocking or under-mocking. In comparison, our technique combines several different code-level features of the dependencies and the classes under test to recommend context-aware mocking decisions. As shown in our evaluation, our technique can outperform the baselines that adopt single factors to make mocking decisions.

9 CONCLUSION

In this work, we conducted an empirical study on four large-scale, actively-maintained open-source projects and identified ten code-level characteristics of the mocked dependencies. Our identified characteristics captured both the feature of the dependencies themselves and the interactions between the dependencies and the CUTs. Based on our identified characteristics, we further proposed *MockSniffer*, an automated technique that recommends mocking decisions. *MockSniffer* leverages machine learning techniques and recommends context-aware mocking decisions by learning from existing mocking practices. Our evaluation shows that *MockSniffer* can effectively recommend mocking decisions and can outperform the generic mocking strategies adopted by existing studies.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of ASE'20 for their constructive comments. This work is supported by the National Natural Science Foundation of China (No. 61932021), the WeBank-HKUST Joint Laboratory, the General Research Fund (No. 16211919) by the Hong Kong Research Grant Council, and the Guangdong Provincial Key Laboratory (No. 2020B121201001). Lili Wei was supported by the Postdoctoral Fellowship Scheme by the Hong Kong Research Grant Council.

REFERENCES

[1] 2013. *CAMEL-6826 apache/camel@dae6366.* Retrieved May 2020 from https://github.com/apache/camel/commit/dae6366

[2] 2013. *hadoop/hadoopTestCachingKeyProvider.java at branch-3.2.1 apache/hadoop.* Retrieved May 2020 from https://github.com/apache/hadoop/blob/branch-3.2.1/hadoop-common-project/hadoop-common/src/test/java/org/apache/hadoop/crypto/key/TestCachingKeyProvider.java#L59

[3] 2020. *EasyMock.* Retrieved May 2020 from https://easymock.org/

[4] 2020. *EvoSuite MockList.* Retrieved May 2020 from https://github.com/EvoSuite/evosuite/blob/master/runtime/src/main/java/org/evosuite/runtime/mock/MockList.java

[5] 2020. *[FLINK-16300] ASF JIRA.* Retrieved May 2020 from https://issues.apache.org/jira/browse/FLINK-16300

[6] 2020. *hadoop/TestJobHistoryEventHandler.java at branch-3.2.1 apache/hadoop.* Retrieved May 2020 from https://github.com/apache/hadoop/blob/branch-3.2.1/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-app/src/test/java/org/apache/hadoop/mapreduce/jobhistory/TestJobHistoryEventHandler.java#L1084

[7] 2020. *jMock - An Expressive Mock Object Library for Java.* Retrieved May 2020 from http://jmock.org/

[8] 2020. *Mickito frmework site.* Retrieved May 2020 from https://site.mockito.org/

[9] 2020. *moq/moq4: Repo for managing Moq 4.x - GitHub.* Retrieved May 2020 from https://github.com/moq/moq4

[10] 2020. *Sinon.JS - Standalone test fakes, spies, stubs and mocks for JavaScript. Works with any unit testing framework.* Retrieved May 2020 from https://sinonjs.org/

[11] Alfred V Aho. 1988. *Compilers : principles, techniques, and tools.* Addison-Wesley Pub. Co., Reading, Mass.

[12] Nadia Alshahwan, Yue Jia, Kiran Lakhotia, Gordon Fraser, David Shuler, and Paolo Tonella. 2010. AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation. In *Practical Software Testing: Tool Automation and Human Factors, 14.03. - 19.03.2010 (Dagstuhl Seminar Proceedings)*, Vol. 10111. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2010/2618/

[13] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.* ACM, 79–90. https://doi.org/10.1145/2642937.2642986

[14] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* ACM, 155–165. https://doi.org/10.1145/2786805.2786828

[15] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API Access and Functional Mocking in Automated Unit Test Generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017.* IEEE Computer Society, 126–137. https://doi.org/10.1109/ICST.2017.19

[16] Thilini Bhagya, Jens Dietrich, and Hans W. Guesgen. 2019. Generating Mock Skeletons for Lightweight Web-Service Testing. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019.* IEEE, 181–188. https://doi.org/10.1109/APSEC48747.2019.00033

[17] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. 1992. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992.* ACM, 144–152. https://doi.org/10.1145/130385.130401

[18] Leo Breiman. 1996. *Bias, variance, and arcing classifiers.* Technical Report. Tech. Rep. 460, Statistics Department, University of California, Berkeley ....

[19] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth.

[20] John W Creswell and Cheryl N Poth. 2016. *Qualitative inquiry and research design: Choosing among five approaches.* Sage publications.

[21] The Apache Software Foundation. 2018. *The Apache Software Foundation Announces Apache® Oozie(TM) v5.0.0.* Retrieved May 2020 from http://www.globenewswire.com/news-release/2018/04/18/1481007/0/en/The-Apache-Software-Foundation-Announces-Apache-Oozie-TM-v5-0-0.html

[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011.* ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[23] Yoav Freund and Robert E. Schapire. 1995. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 904. Springer, 23–37. https://doi.org/10.1007/3-540-59119-2_166

[24] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Ann. Statist.* 29, 5 (2001), 1189–1232.

[25] Karl Pearson F.R.S. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175. https://doi.org/10.1080/14786440009463897

[26] Stefan J. Galler, Andreas Maller, and Franz Wotawa. 2010. Automatically extracting mock object behavior from Design by Contract™ specification for test data generation. In *The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa.* ACM, 43–50. https://doi.org/10.1145/1808266.1808273

[27] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 643–653. https://doi.org/10.1145/2635868.2635920

[28] Tim Mackinnon, Steve Freeman, and Philip Craig. 2001. *Endo-Testing: Unit Testing with Mock Objects.* Addison-Wesley Longman Publishing Co., Inc., USA, 287–301.

[29] Inderjeet Mani and I Zhang. 2003. kNN approach to unbalanced data distributions: a case study involving information extraction. In *Proceedings of workshop on learning from imbalanced datasets*, Vol. 126.

[30] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[31] Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada, May 18-19, 2009.* IEEE Computer Society, 149–153. https://doi.org/10.1109/IWAST.2009.5069054

[32] Shaikh Mostafa and Xiaoyin Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014.* 127–132. https://doi.org/10.1109/QSIC.2014.19

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[34] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* IEEE Computer Society, 201–211. https://doi.org/10.1109/ASE.2015.86

[35] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To mock or not to mock?: an empirical study on mocking practices. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 402–412. https://doi.org/10.1109/MSR.2017.61

[36] Davide Spadini, Maurício Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems - Why and how developers use them, and how they evolve. *Empirical Software Engineering* 24, 3 (2019), 1461–1498. https://doi.org/10.1007/s10664-018-9663-0

[37] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: automated test generation for database applications via mock objects. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 289–292. https://doi.org/10.1145/1858996.1859053

[38] Nikolai Tillmann and Wolfram Schulte. 2006. Mock-object generation with behavior. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan.* IEEE Computer Society, 365–368. https://doi.org/10.1109/ASE.2006.51

[39] Graham Upton and Ian Cook. 1996. *Understanding statistics.* Oxford University Press.

[40] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada.* 13. https://dl.acm.org/citation.cfm?id=782008

[41] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. *Empir. Softw. Eng.* 23, 5 (2018), 2866–2900. https://doi.org/10.1007/s10664-017-9567-4

[42] Harry Zhang. 2004. The Optimality of Naive Bayes. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, Valerie Barr and Zdravko Markov (Eds.). AAAI Press, 562–567. http://www.aaai.org/Library/FLAIRS/2004/flairs04-097.php

[43] Linghao Zhang, Xiaoxing Ma, Jian Lu, Tao Xie, Nikolai Tillmann, and Peli de Halleux. 2012. Environmental Modeling for Automated Cloud Application Testing. *IEEE Software* 29, 2 (2012), 30–35. https://doi.org/10.1109/MS.2011.158