Augmenting Test Oracles with Production Observations

Deepika Tiwari

Supervised by
Professors Benoit Baudry and Martin Monperrus

School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
KTH Royal Institute of Technology
Stockholm, Sweden, 2024

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2024:87
ISBN 978-91-8106-109-3

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framläggs till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik fredagen den 13 december 2024 klockan 14.00 i Kollegiesalen, Brinellvägen 6, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Software testing is the process of verifying that a software system behaves as it is intended to behave. Significant resources are invested in creating and maintaining strong test suites to ensure software quality. However, in-house tests seldom reflect all the scenarios that may occur as a software system executes in production environments. The literature on the automated generation of tests proposes valuable techniques that assist developers with their testing activities. Yet **the gap between tested behaviors and field behaviors remains largely overlooked.** Consequently, the behaviors relevant for end users are not reflected in the test suite, and the faults that may surface for end-users in the field may remain undetected by developer-written or automatically generated tests.

This thesis proposes a novel **framework for using production observations, made as a system executes in the field, in order to generate tests**. The generated tests include test inputs that are sourced from the field, and oracles that verify behaviors exhibited by the system in response to these inputs. We instantiate our framework in three distinct ways.

First, for a target project, we focus on methods that are inadequately tested by the developer-written test suite. At runtime, we capture objects that are associated with the invocations of these methods. The captured objects are used to generate tests that recreate the observed production state and contain oracles that specify the expected behavior. Our evaluation demonstrates that **this strategy results in improved test quality for the target project.**

With the second instantiation of our framework, we observe the invocations of target methods at runtime, as well as the invocations of methods called within the target methods. Using the objects associated with these invocations, we generate tests that use mocks, stubs, and mock-based oracles. We find that **the generated oracles verify distinct aspects of the behaviors observed in the field, and also detect regressions within the system.**

Third, we adapt our framework to capture the arguments with which target methods are invoked, during the execution of the test suite and in the field. We generate a data provider using the union of captured arguments, which supplies values to a parameterized unit test that is derived from a developer-written unit test. Using this strategy, **we discover developer-written oracles that are actually generalizable to a larger input space.**

We evaluate the three instances of our proposed framework against real-world software projects exercised with production workloads. Our findings demonstrate that runtime observations can be harnessed to generate complete tests, with inputs and oracles. The generated tests are representative of real-world usage, and can augment developer-written test suites.

Keywords: Test generation, Test oracles, Production observations

Sammanfattning

Programvarutestning är processen för att verifiera att ett mjukvarusystem fungerar som det är tänkt att fungera. Betydande resurser investeras i att skapa och underhålla starka testsviter för att säkerställa mjukvarukvalitet. Interna tester återspeglar dock sällan alla scenarier som kan uppstå när ett mjukvarusystem körs i produktionsmiljöer. Litteraturen om automatiserad testgenerering föreslår värdefulla tekniker för att hjälpa utvecklare i deras testaktiviteter. Ändå förbises **gapet mellan testade beteenden och beteenden i produktionsmiljöer till stor del**. Följaktligen återspeglas inte beteenden som är relevanta för slutanvändare i testsviten, och de fel som kan visas för slutanvändare i reella situationer kan förbli oupptäckta av utvecklarskrivna eller automatiskt genererade tester.

Denna avhandling föreslår ett nytt **ramverk för att använda produktionsobservationer, gjorda när ett system exekverar i produktionsmiljö, för att generera tester**. De genererade testen inkluderar testindata som kommer från reella användare och orakel som verifierar beteenden som uppvisas av systemet som svar på dessa indata. Vi instansierar vårt ramverk på tre olika sätt.

Först, för ett målprojekt, fokuserar vi på metoder som är otillräckligt testade av den utvecklarskrivna testsviten. Vid körning registrerar vi objekt som är associerade med anropen till dessa metoder. De registrerade objekten används för att generera tester som återskapar det observerade produktionstillståndet och innehåller orakel som anger det förväntade beteendet. Vår utvärdering visar att **denna strategi resulterar i förbättrad testkvalitet för målprojektet**.

Med den andra instansieringen av vårt ramverk observerar vi anrop till målmetoder vid körning, såväl som anrop till metoder som anropas inom målmetoderna. Med hjälp av objekten som är associerade med dessa anrop genererar vi tester som använder mocks, stubs och mock-baserade orakel. Vi finner att **de genererade oraklen verifierar distinkta aspekter av beteenden som observerats i produktionsmiljöer, och även upptäcker regressioner inom systemet**.

För det tredje anpassar vi vårt ramverk för att registrera de argument med vilka målmetoder anropas, under körning av testsviter och i produktion. Vi genererar en dataleverantör med hjälp av sammansättningen av registrerade argument, som tillhandahåller värden till ett parameteriserat enhetstest härlett från ett utvecklarskrivet enhetstest. Med den här strategin **upptäcker vi utvecklarskrivna orakel som faktiskt är generaliserbara till ett större inmatningsutrymme**.

Vi utvärderar de tre fallen av vårt föreslagna ramverk mot verkliga programvaruprojekt som körs med produktionsbelastning. Våra resultat visar att körtidsobservationer kan utnyttjas för att generera kompletta tester, med indata och orakel. De genererade testerna är representativa för användning i verkligheten och kan utöka utvecklarskrivna testsviter.

Nyckelord: Testgenerering, Testorakel, Produktionsobservationer

for mummy and papa

List of Papers

This thesis includes the following contributions by the author, listed in chronological order.

I. D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production Monitoring to Improve Test Suites," *IEEE Transactions on Reliability*, vol. 71, no. 3, 2021: pp. 1381-1397,
doi: 10.1109/TR.2021.3101318

II. L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry "Harvesting Production GraphQL Queries to Detect Schema Faults," *in Proceedings of the 15th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2022: pp. 365-376,
doi: 10.1109/ICST53961.2022.00014

III. D. Tiwari, Y. Gamage, M. Monperrus, and B. Baudry, "PROZE: Generating Parameterized Unit Tests Informed by Runtime Data," *in Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024,
doi: 10.48550/arXiv.2407.00768

IV. D. Tiwari, M. Monperrus, and B. Baudry, "Mimicking Production Behavior with Generated Mocks," *IEEE Transactions on Software Engineering*, 2024,
doi: 10.1109/TSE.2024.3458448

The following contributions by the author, listed in chronological order, are not included in this thesis.

I. L. Zhang, D. Tiwari, B. Morin, B. Baudry, and M. Monperrus, "Automatic Observability for Dockerized Java Applications," *arXiv preprint*, 2021, doi: 10.48550/arXiv.1912.06914
Summary: Docker images can be augmented automatically with enhanced support for observability at different levels. The author of this thesis contributed to the experiments and the writing of this paper.

II. D. Tiwari, M. Monperrus, and B. Baudry, "RICK: Generating Mocks from Production Data," *in Proceedings of the 16th IEEE International Conference on Software Testing, Verification, and Validation: Demo track (ICST)*, 2023: pp. 464-466, doi: 10.1109/ICST57152.2023.00051
Summary: The author of this thesis was responsible for the implementation of RICK for automated mock generation from production observations, as well as the writing of this short paper demonstrating the tool.

III. C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, 2023: pp. 5027-5045, doi: 10.1109/TSE.2023.3324950
Summary: Unused parts of otherwise used dependencies can be automatically identified and removed from the dependency trees of Java projects. The author of this thesis contributed to the conceptualization, experiments, analysis, and writing of this paper.

IV. B. Baudry, K. Etemadi, S. Fang, Y. Gamage, Y. Liu, Y. Liu, M. Monperrus, J. Ron, A. Silva, and D. Tiwari, "Generative AI to Generate Test Data Generators," *IEEE Software*, 2024: pp. 1-9, doi: 10.1109/MS.2024.3418570
Summary: Generative AI can be used to produce realistic test data and test data generators in diverse natural and programming languages. The author of this thesis contributed to the conceptualization, experiments, and writing of this paper.

V. Y. Liu, D. Tiwari, C. Bogdan, and B. Baudry "An Empirical Study of Bloated Dependencies in CommonJS Packages," *arXiv preprint*, 2024, doi: 10.48550/arXiv.2405.17939
Summary: Dynamic analysis can be employed to detect unused dependencies within NodeJS applications. The author of this thesis contributed to the writing of this paper.

VI. J. Wachter, D. Tiwari, M. Monperrus, and B. Baudry, "Serializing Java Objects in Plain Code," *arXiv preprint*, 2024, doi: 10.48550/arXiv.2405.11294

Summary: Runtime objects can be serialized as plain Java code for use within readable tests that are generated automatically. The author of this thesis contributed to the conceptualization and writing of this paper.

The author has it on good authority that the following overarching contribution warrants a special mention.

XLII. D. Tiwari, T. Toady, M. Monperrus, and B. Baudry, "With Great Humor Comes Great Developer Engagement," *in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2024: pp. 1-11,
doi: 10.1145/3639475.3640099
Summary: One of the earliest proponents of Software Engineering was not averse to software humor. Healthy and respectful self-expression does not detract from serious software engineering. The author of this thesis contributed to the conceptualization, implementation, writing, and live performance of this paper, in close collaboration with co-authors.

Acknowledgements

With this most crucial chapter of my thesis, I hope I can convey, even if just the tiniest bit, my deepest appreciation for those who have contributed to it.

I am indebted to the generosity and friendship offered by my stellar supervisors, *Profs.* Benoit Baudry and Martin Monperrus. I believe I struck `faker.preciousMetal()`, having done my PhD under their watchful and humorous [1] guidance. Thanks, Benoit and Martin, for your dedication to your craft, your earnest positivity, and for inviting many like myself to make what we will of the opportunities you give us.

I thank each of my co-authors, from whom I have learned about collaboration and so much more. To all the brilliant and kind members, current and past, of our appropriately named AweSome Software Engineering Research Team (ASSERT), thank you. We gather from distant shores to a little kitchen in our corridor at KTH, with so much in common, and yet such a lot to learn from each other. Belonging to this group has been a source of comfort during the (few weeks of) summers and (rather long) winters of Sweden.

I am eternally grateful to my parents, to whom I owe everything. Thanks also to my family and friends, across time zones, as well as Kaaju, the best good boy Anywhere on Earth. Thank you for listening politely when I summarized my latest experiments, despite not being asked to.

I would not have had the courage to embark on this adventure, or persevere through it, were it not for Vaibhav. I am immensely proud of the shared interests, the ever increasing number of reruns of our favorite classics, and the two doctoral theses between us, in no specific order. Thank you for our life in Uppsala, which is always worth the commute. I am excited for all that is to come!

I want to express my sincere gratitude to the members of my PhD committee, Professors Paolo Tonella, Mira Mezini, Robert Feldt, and Serge Demeyer. I am thankful to Professor Panagiotis Papadimitrados for serving as advance reviewer of this thesis, and to Professor Aristides Gionis for chairing the proceedings of my PhD defense. My PhD has been generously supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

Contents

Part I

Thesis

1. Introduction

"Pearl-fishers dive for pearls,
merchants sail in their ships,
while children gather pebbles and
scatter them again. They seek
not for hidden treasures, they
know not how to cast nets."

Rabindranath Tagore,
On the Seashore

Software has established itself as an essential actor in our lives. Our civilization is undeniably reliant on it, and for good reason. The versatility of software has helped cement its place across all enterprises [2], from finance [3] to culture [4]. We have grown to depend on increasingly more sophisticated software systems to function perfectly — much is at stake if they don't [5]. In order to guarantee their quality and reliability, it is important that software systems are tested meticulously. *Software testing* is the process of verifying that a software system behaves as it is intended to behave. A deviation from an intended behaviour unearths a problem within the system that must be fixed [6].

1.1 Software testing

Everyday, the millions of software developers around the world perform millions of *software tests* [7]. They write *automated* tests to verify that the features added to their software project work as they are required to. Additionally, whenever a bug is discovered and fixed in the system, tests are written to ensure that it is in fact corrected and that it does not reoccur as the project evolves. Depending on the testing practices of its developers, the *test suite* of even modest software projects can contain thousands of automated tests. Each time a new contribution is made by any of its developers, the test suite of a project is run in its entirety. If all tests are green, i.e., all tests verify that the new changes to the project align with expectations and do not introduce unintentional side-effects or *regressions* [8], the project is deployed to production for real-world usage. Production environments are unpredictably diverse, and the same system may behave differently depending on a multitude of factors, such as hardware, other software, and end user traffic. Insights from field behaviors, in the form of failure reports [9] and performance

```
1  public class UnitConverter {
2    ...
3    public double convertMilesToKm(double miles) {
4      if (miles <= 0)
5        return 0;
6      return miles * 1.60934;
7    }
8  }
9  ..........................................................................................
10 @Test
11 public void testConvertMilesToKm() {
12   // input
13   UnitConverter converter = new UnitConverter(...);
14   double km = converter.convertMilesToKm(10);
15   // oracle
16   assertEquals(16.0934, km, 0.0);
17 }
```

Listing 1.1: The behavior of the method `convertMilesToKm` is verified by the unit test, `testConvertMilesToKm`. The key components within the test are the test input and the test oracle.

metrics, may be incorporated within the next cycle of development and testing. This process is creative, indispensable, and perpetual.

In this thesis, we focus on the process of creating new automated software tests. An automated software test is a piece of code that runs against other code to establish its validity. Regardless of the scale of the system under test, or the granularity with which its validity is being verified, all tests are composed of two key components. The first component is the *test input*, which brings the system to an initial testable state. This includes setting up the test environment, initializing the program to the desired state, and performing the action to be tested against the system. The second component is the *oracle*, i.e., a concrete specification of the behavior expected from the system under the given test input [10,11]. The oracle is responsible for automating the comparison between the expected behavior of the system under test with its actual behavior. Developers typically draw from their expertise of the system under test, and their knowledge of the domain, to specify oracles [12]. However, with growing complexity, it becomes increasingly harder to specify the behaviors expected of the system under test [13,14]. This (in)famous problem in software testing is referred to as the *oracle problem* [15]. We illustrate with the example of Listing 1.1, which presents the automated *unit test*, `testConvertMilesToKm`, for the *focal method* `convertMilesToKm`, which is the method being tested [16]. Within the test, the developer first defines the inputs through program variables, and constructor and method calls. Next, in order to verify the behavior of the focal method, the developer specifies the oracle. The determination of a useful oracle is not a straightforward task. For example, an *implicit oracle* for `convertMilesToKm` is that its invocation should not cause

the program to exit. However, the oracle within `testConvertMilesToKm`, specified through an *assertion* statement, is arguably more meaningful [17], as it is a concrete expression of the expected behavior of `convertMilesToKm`.

The process of writing *good* software tests, which includes designing interesting and representative test inputs and expressing meaningful oracles, requires significant developer effort. It might even be more art than science [18]. Most organizations appreciate this fact, and invest serious resources for the creation and maintenance of strong test suites. However, as highlighted by Wang *et al.* [19,20], in-house developer-written test suites do not account for all the ways in which a system may be exercised by end-users in the field [21]. In fact, anticipating and testing all scenarios that can occur in the field is hardly possible. As a simple example, the interactions of an end-user with our conversion program from Listing 1.1 might trigger the invocation of `convertMilesToKm` with a value of `-6.6`, causing it to return the output `0`. This scenario within `convertMilesToKm` is not represented by any developer-written test. However, a new test with this combination of test input (`convertMilesToKm` called with `-6.6`) and oracle (output value of `0`) would serve as a useful addition to the existing test suite. Needless to say, for larger, more complex systems, the difference between test and field behaviors is more compelling, motivating the need for automatically addressing this gap.

1.2 Problem statement

When writing automated software tests, developers envision typical usage scenarios for their system based on their understanding of its requirements. These assumptions are encoded into the test suite. However, behaviors expressed within the test suite might not overlap with actual behaviors that occur in the field, as the system executes under real workloads [19,22]. Consequently, real-world usage scenarios that are relevant for end-users are not represented within the test suite. The problem statement we address in this thesis is as follows.

The oracles within developer-written tests often do not reflect behaviors that are exhibited by the system in the field, exercised with real workloads.

1.3 Position within the state of the art

Owing to its necessity and its cost within software engineering, software testing has been the subject of research for decades. A large number of studies contribute techniques for automatically generating tests [13], especially focusing on the generation of test inputs that optimize for *code coverage* [23], such that more of the program is reached by the tests. For example, test inputs can be generated randomly [24], using search-based techniques [25], symbolic execution [26], and with machine learning models [27]. However, there are far fewer studies that aim to assist developers with the oracle problem. The key challenge is to propose

algorithms that can automatically determine the behaviors expected of the system under test. As a solution to this challenge, some works extract oracles from formally or informally documented requirements of the system [28–32]. However, such requirement specifications may not always be available or complete [10, 15].

Addressing the frequent unavailability of structured requirements, another body of work proposes strategies to generate oracles from the observed behaviors of the system. In the absence of specifications, the developer-written test suite can be considered an executable proxy for the requirements, and behaviors can be captured from it to generate new tests [33–37]. The key limitation of this approach is that it, by design, relies on the developer-written test suite which, as with specifications, may not always be available or complete [19]. Moreover, the oracles generated by many of these approaches tend to be simple, and not express complex functionalities of the system under test [38, 39]. Additionally, while valuable, the strategies for automated test generation proposed in the literature do not explicitly aim at bridging the gap emphasized by Wang *et al.* [19] between tested behaviors and field behaviors.

We argue that field executions are a rich source of information that can be harnessed for augmenting the test suite with behaviors that are relevant for real-world users. In this thesis, we derive a solution to the oracle problem by observing systems in the field to determine their expected behaviors.

1.4 Thesis contributions

The contributions made by this thesis are highlighted in red in Figure 1.1. Addressing our problem statement about the differences between test and field behaviors, this thesis contributes with a mechanism for capturing runtime behaviors from the field, and feeding them back into the test suite to augment it. Specifically, we monitor a system as it executes in the field, capturing data corresponding to the target units identified within the system. Next, test generation is triggered in-house, using the captured field data [21]. For the target units, the generated tests contain inputs that reflect the captured production states, and oracles that reflect observed outputs. The generated tests can be appended to the developer-written test suite and run in-house to verify functional behaviors, and detect regressions within the system as it evolves. We realize three applications of our proposed mechanism, and make the following contributions.

C1 Techniques for deriving oracles from runtime observations for improving test quality
We capture data as systems execute in the field and generate tests with explicit oracles. The oracles contribute to improved test quality, with respect to coverage and mutation analysis.

C2 A technique for deriving mock-based oracles from runtime observations for regression testing
We capture distinct aspects of runtime behaviors, and express them within

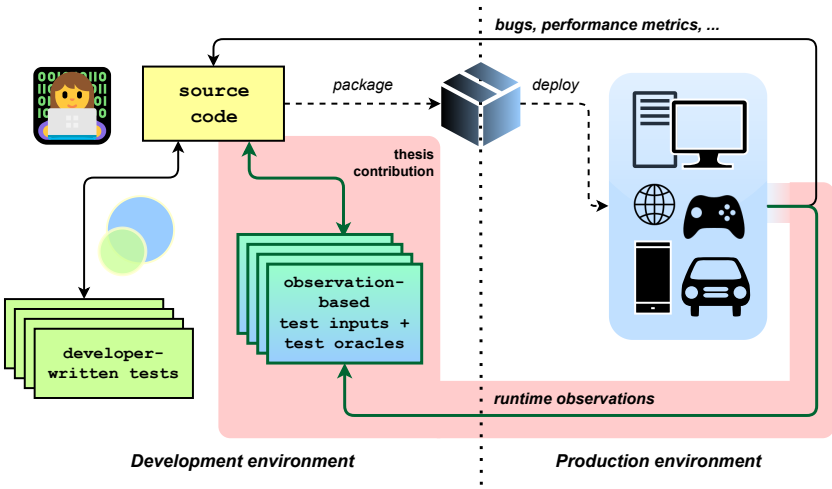


Figure 1.1: The contributions of this thesis (highlighted in red) within software testing. Within the development environment, the developer writes tests, which are run before the project is deployed to the production environment. Feedback from production in the form of bug reports and performance metrics triggers changes to the source code and associated tests. However, some behaviors that occur in production might not be represented within the test suite [19]. This thesis proposes a mechanism for making runtime observations in the production environment, and utilizing them as test inputs and oracles within generated tests. The generated tests reflect real-world usage of the system and can augment the developer-written test suite.

generated tests as mocks, stubs, and mock-based oracles. The generated mock-based oracles mimic field observations and detect regressions.

C3 A technique for generalizing developer-written oracles using runtime observations
We capture the arguments passed to methods at runtime and use them to transform existing unit tests into parameterized unit tests. The developer-written oracles within these parameterized unit tests hold over a larger range of test inputs than envisioned by developers.

1.5 Summary of papers

We now summarize the four research papers included in this compilation thesis, associating them in Table 1.1 with the three contributions listed in Section 1.4.

I. D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, **"Production Monitoring to Improve Test Suites,"** *IEEE Transactions on Reliability*, 2021. doi: 10.1109/TR.2021.3101318

Table 1.1: Contributions made by the papers included in this thesis

Contribution	Paper I	Paper II	Paper III	Paper IV
C1: Oracles for improved test quality				
C2: Mock-based oracles for regression testing				
C3: Generalizing developer-written oracles				

Summary: This paper presents PANKTI, a tool that identifies target methods based on an adequacy criterion defined on the developer-written test suite, captures objects associated with the target methods in the field, and generates unit tests that encode the captured objects as test inputs and test oracles. We evaluate PANKTI against three real-world Java projects, targeting methods that are pseudo-tested by the developer-written test suite. The tests generated by PANKTI augment the developer-written test suite of the three projects we experiment with. A majority of the target methods are no longer pseudo-tested as a result of the explicit oracles derived for them from runtime observations.

Contributions: As the main author, the author of this thesis contributed to the conceptualization and implementation of PANKTI. She was also responsible for designing and conducting the experiments in collaboration with co-authors, analyzing the results, and writing the manuscript.

II. L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry **"Harvesting Production GraphQL Queries to Detect Schema Faults,"** *in Proceedings of the 15th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2022. doi: 10.1109/ICST53961.2022.00014

Summary: In Paper II, we present AutoGraphQL, a tool that generates tests using incoming HTTP requests for web applications that use GraphQL APIs, defined using a GraphQL schema. The captured GraphQL query requests are replayed within the generated tests. The assertions within the test verify the properties of the response object against the expected properties specified in the GraphQL schema. The tests generated by AutoGraphQL augment the test suites of one open-source and one industrial application, increasing their coverage and detecting faults in the implementation of the

schema.

Contributions: The first author was a Master's thesis student responsible for the implementation of AutoGraphQL, and was supervised by the author of this thesis. The author of the thesis was also responsible for designing and conducting the experiments with the open-source study subject, analyzing the results, and writing the manuscript, in collaboration with co-authors.

III. D. Tiwari, Y. Gamage, M. Monperrus, and B. Baudry, **"PROZE: Generating Parameterized Unit Tests Informed by Runtime Data,"** *in Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024. doi: 10.48550/arXiv.2407.00768

Summary: We present PROZE in Paper III as a technique for automatically transforming existing unit tests into parameterized unit tests. PROZE identifies methods that are directly invoked within developer-written unit tests, and observes their invocations at runtime. It uses the union of arguments with which a method gets invoked within the test suite and in the field, to generate an argument provider. This provider supplies values to a unit test parameterized over the target method. We experiment with five Java modules from two large projects, and find that PROZE augments their test suites with parameterized unit tests containing developer-written oracles that in fact generalize over a larger range of test inputs captured at runtime.

Contributions: As the main author, the author of this thesis contributed to the conceptualization and implementation of PROZE. She was also responsible for designing and conducting the experiments in collaboration with co-authors, analyzing the results, and writing the manuscript.

IV. D. Tiwari, M. Monperrus, and B. Baudry, **"Mimicking Production Behavior with Generated Mocks,"** *IEEE Transactions on Software Engineering*, 2024. doi: 10.1109/TSE.2024.3458448

Summary: Paper IV introduces RICK, a tool that captures objects corresponding to the invocation of a target method in the field, as well as the method calls that occur within the invocation of the target method. Using the captured data, RICK generates unit tests that mock external objects, stubbing their interactions with the target method. The three types of mock-based oracles generated by RICK verify distinct aspects of observed field behaviors, specifically the output expected from the target method, the arguments with which the mock method calls occur within the invocation of the target method, and the sequence and frequency of these mock method calls. Our evaluation of RICK against three Java applications shows that it augments their test suites with mock-based oracles that complement each other in their ability to detect regressions.

Contributions: As the main author, the author of this thesis contributed to the conceptualization and implementation of RICK. She was also responsible for designing and conducting the experiments, analyzing the results, and writing the manuscript in collaboration with co-authors.

1.6 Thesis outline

Part I presents the contributions of this thesis. Chapter 2 summarizes the literature on oracle generation and observation-based test generation. Next, Chapter 3 introduces our methodological framework, and Chapter 4 describes our experimental results. We share concluding remarks and directions for future work in Chapter 5. Part II contains the four research papers that are included in this compilation thesis.

2. State of the Art

"Boond boond sagar hai,
warna yeh sagar kya hai?"

Javed Akhtar,
Yeh Tara Woh Tara

In this chapter, we discuss the works most closely related to the contributions of this thesis. Section 2.1 summarizes the literature on test oracles, and Section 2.2 presents an overview of automated test generation techniques, focusing on the closely related body of work on test generation based on runtime observations.

2.1 Test oracles

The two key components within each software test, manual or automated, are the test input and the *test oracle*. The test inputs bring the system under test to the desired state, also performing the action to be tested. It is then the job of the oracle to ascertain if the system behaves correctly, i.e., the outcome from the test aligns with expectations.

The ***oracle problem*** [15] refers to the challenge of determining the expected behaviors of the system under test, and verifying that the system conforms to this expected behavior. Finding a solution to the oracle problem is critical for the usefulness of each test [6], yet largely the responsibility of the developer [7, 11]. However, researchers have contributed techniques for the automated generation, assessment, and improvement of oracles, in an effort to alleviate the oracle problem [10, 40]. In this section, we summarize these studies with respect to two dimensions: first, the determination of the oracle, i.e., the expected behavior of the system that must be checked; and second, the concrete specification of the oracle, such that the conformance of the system with the expected behavior can be automatically verified.

Oracle generation

A large number of studies have explored strategies for the automated generation of oracles, for both developer-written and automatically generated tests. The mechanisms proposed by these studies derive oracles from different sources of

information, such as the documented requirements of a system or its observed behaviors.

Deriving oracles from requirements: Some studies mine requirement specifications, documented in diverse ways, to deduce oracles for the system under test. Li and Offut [28, 41] recommend strategies for defining oracles such that abstract, model-based tests can be transformed into concrete tests that reveal failures. They propose going beyond implicit oracles that check for the absence of runtime exceptions or crashes, and incorporating oracles that observe internal states [42] specified within assertion statements. Carzaniga *et al.* [29] produce oracles for a system using its developer-written Javadoc [43] in conjunction with specifications derived from its intrinsic redundancy, i.e., multiple ways of performing the same action. Given a pair of redundant methods, each generated *cross-checking* oracle, represented as an advice within an aspect class, compares the output of a method call with the result from an equivalent method call. Goffi *et al.* [30] discover that oracles for exceptional behaviors are rarely present in developer-written and automatically generated tests. They propose Toradocu, which uses natural language processing to parse Javadoc comments in order to generate oracles that verify the exceptions thrown by a method. The oracles are implemented as aspects that instrument existing tests, getting activated each time the method is invoked. Blasi *et al.* [31] extend Toradocu into Jdoctor. In addition to the documentation of exceptional behaviors, Jdoctor also parses pre- and post-conditions specified within the Javadoc of methods and constructors. The oracles derived from this analysis are encoded as Java expressions that can be embedded within automatically generated tests [44]. Motwani and Brun [32] present Swami, a technique that generates oracles from specifications that are documented in natural language, without the need for the source code of the project. Swami uses regular expressions for the identification of boundary conditions and exceptional behaviors from the specifications of non-trivial Java, JavaScript, and C++ projects. CrowdOracles [38] delegates the oracle problem to the human participants of a crowd-sourcing campaign. The participants rely on documentation to determine the validity of the assertions within automatically generated tests [25]. A family of tests called *theory-* [45] or *property-based* tests (PBTs) [46, 47] focus on the formally specified or developer-identified properties of a system [48]. The oracles within these tests express properties that must always be true for the system, acting as executable specifications for it [49]. The oracles within PBTs are evaluated against random input data provided by data generators [46].
Reflection: Utilizing the requirement specifications of the system under test is a meaningful strategy for the derivation of oracles that specify the behaviors expected of it. However, by design, this strategy relies on the presence of requirements documented either formally or at least in a semi-structured manner by developers. In practice, there may be limited access to such structured specifications within real-world projects, and these specifications are also subject to

evolution.

Deriving oracles from observed behaviors: The oracles for a system may also be deduced without the use of documented requirements, leveraging observability [50] for oracle generation. The Orstra tool by Xie [51] captures object states during the execution of an automatically generated test suite, which it then uses to add assertion oracles within the tests. These oracles, aptly called *regression oracles*, can detect regressions during subsequent executions of the test suite, as the system evolves [8]. Some studies focus on the values contained in program variables, which can be monitored during the execution of tests and serve as oracle data. Mutation analysis has been used to select the variables used as oracle data, which can be ranked to maximize the fault detection ability of tests for civil avionics [52] and medical device [36] systems. Addressing the generalizability and scalability of this approach, DODONA [35] employs a different strategy. It monitors the interactions of, and dependencies among, variables during test execution, and ranks them using network centrality analysis to recommend oracle data. The goal of these works on oracle data selection is to support developers in their effort to create *expected value test oracles*. Another way of approaching the oracle problem is through metamorphic testing [53], which has also received considerable attention in the literature [54,55]. The oracles used within metamorphic tests specify *metamorphic relations* (MRs), which signify the change in the output from the system with respect to the change in the input [56]. MR-Scout [37] observes the execution of a test suite to capture MRs that encode the change in the state of an object. The MRs are included as assertions within automatically generated tests [25]. Invariants, which are properties that must hold for a system in a specific state, can also serve as oracles. Daikon [33,57] is a popular tool for discovering likely [58] invariants, inferring them from the traces of a system exercised via symbolic execution. DySy [59] extends this approach, inferring invariants through dynamic symbolic execution. A large number of studies on automated test generation encapsulate the invariants detected using these approaches within assertion statements
Reflection: Oracles can be generated automatically, without relying on documented specifications, by considering the observed program behaviors as the expected behaviors. Most contributions in this space capture behaviors by observing the execution of developer-written tests. However, as with specifications, developer-written test suites may not be complete, or even present, in practice.

Generating oracles using learning-based approaches: An increasing number of studies perform oracle synthesis using learning-based techniques. ATLAS [17] uses neural machine translation to generate an assertion statement for a test method, given its focal method. The generated assert statements resemble developer-written assertions in their meaningfulness and complexity. Yu *et al.* [60] extend ATLAS by incorporating information retrieval to fetch an assertion for a focal test that is most similar to the test within the training data, and adapting the retrieved assertion for the focal test. Molina *et al.* [61,62] obtain valid pre- and

post-states for a method under test by executing it within generated tests, and then guide a genetic algorithm to learn the postconditions of the method for use as oracles. The inferred oracles, expressed as assertions, encode expected object states. TOGA [63] generates oracles for a focal method, and ranks them using a transformer-based approach. It does not rely on the implementation of the focal method, but rather its signature and documentation, and the prefix of the test generated automatically for it [64]. The generated oracles are expressed as assertions that check the value returned from the method or an exception thrown from it. The works of Hossain *et al.* [65] and Liu *et al.* [66] extend the evaluation [67] of TOGA, highlighting areas of improvement for neural oracle generation. Without the need for an assertion oracle, SEER [68] predicts whether a unit test passes or fails, using neural representations of correct versions of methods embedded with their passing tests, and incorrect versions with their failing tests. The use of Large language models (LLMs) for test oracle synthesis is also being investigated actively [69]. TOGLL by Hossain and Dwyer [70] is a technique for producing diverse assertion and exception oracles by prompting LLMs. The generated oracles have improved fault-detection capabilities, compared to the oracles from TOGA. *Reflection:* As LLMs become more skilled at handling software development tasks, their use for oracle generation is also gaining attention in the literature. This is welcome news for developers who prefer to use LLMs as assistants when writing tests. However, the responsibility of ensuring the validity of their outputs lies largely with developers.

Oracle augmentation

Oracles created by human developers, or those generated automatically, are not guaranteed to be perfect [68]. In their survey, Danglot *et al.* [71] refer to *test amplification* as the process of enriching existing tests, of which oracle improvement is a cornerstone. Several studies assess the effectiveness of oracles, and propose strategies for their augmentation.

Existing oracles differ in their ability to detect faults. Staats *et al.* [72] argue that tests can be prioritized based on their oracles, for quicker fault detection, and thus more effective testing. Their approach for prioritization is based on analyzing the proportion of the program that is actually exercised by the oracle. Schuler and Zeller [73, 74] introduce checked coverage as an indicator for oracle quality. Unlike the traditional coverage metric, which represents the statements that are reached during the execution of the test suite, checked coverage represents covered statements that actually contribute to the results evaluated by the oracles contained in the test suite. Mutation analysis can also help discover inadequate oracles [75], and provide more evidence that coverage does not guarantee fault detection. An extreme example is that of pseudo-tested methods [76]. Despite these methods being covered by existing tests, no oracle detects the replacement of the entire body of these methods with a default return statement. Vera-Pérez *et al.* [77] demonstrate the prevalence of pseudo-tested methods within projects that have test suites with high coverage. OraclePolish by Huo and Clause [78] detects

oracles that over-specify by checking too much, or under-specify by checking too little. An erroneous oracle can raise false alarms (false positives) or fail to detect bugs (false negatives), as discussed by Guo *et al.* [79], who recommend debugging the oracle itself. The OASIs tool proposed Jahangirova *et al.* [80, 81] identifies such deficient oracles, using a search-based approach for the false positive cases, and mutation analysis for the false negative ones. Through an evaluation with a human study [82], the authors also discover that OASIs identifies false positives and negatives more accurately, relative to manual assessment of such cases. Furthermore, GAssert [83, 84] uses an evolutionary algorithm to automatically improve the oracle deficiencies identified by OASIs.
Reflection: The literature on oracle augmentation highlights the inherent complexity of the oracle problem. Oracles are the essence of testing, yet it is challenging to determine meaningful oracles manually, and more so to generate them automatically.

Domain-specific oracles

The oracle problem has been studied in more specific contexts, highlighting its pervasiveness across domains. For example, AGORA [85] adapts the Daikon system [33] to learn oracles that express the invariants of RESTful web APIs, based on their OpenAPI specification and past requests. For the social networking platforms of Meta, the ALPACAS tool [86] generates integrity tests that keep users safe from harmful online content and behavior. Using logs simulated from production-like scenarios, ALPACAS infers oracles which are responsible for decisions regarding user-reported content. Across systems with rich Graphical User Interfaces (GUIs), Augusto [87] can generate oracles that express the outcomes of commonly-used operations, such as those that represent the creation and deletion of entities, or filling up of registration forms. On the other hand, for graphical systems such as digital TVs, the ADVISOR framework [88] facilitates the configuration of computer vision models in order to produce oracles that can visually compare expected and actual outputs.

Many studies focus on automated oracles for mobile applications. For instance, Lin *et al.* [89] argue that taking screenshots of the application under test for use within oracles can overload an already busy Android device. They propose replaying recorded GUI events and photographing the application GUI with an external camera, so that it can be tested using oracles that are automated and device-agnostic. GUI-based oracles that are reusable across applications can also be derived from common interactions with UI elements and their corresponding response [90, 91]. Jabbarvand and colleagues [92] approach the oracle problem for Android applications from the perspective of their energy consumption, employing a deep learning model to detect unusual patterns of energy usage.

Several studies address the oracle problem for cyber-physical systems, big and small [36, 52]. For example, Gay *et al.* [93] propose steering the models used to test real-time embedded systems to allow for more flexible oracles. These oracles can accommodate the non-deterministic behaviors that are characteristic of such

systems, while successfully verifying acceptable behaviors. Mithra by Afzal *et al.* [94] uses simulated telemetry data for cyber-physical systems to learn oracles for them that detect anomalous behavior. Oracles for elevators can be generated and specified in domain-specific languages so that they can be continually evolved and tested [95], and machine learning models can also provide oracles that help identify bugs in their functionalities [96]. Jahangirova *et al.* [97] propose oracles for self-driving cars, deriving them from metrics that represent the quality of driving of human drivers.

Reflection: Finding a solution to the oracle problem is a challenge across all domains. Much research has been done on automatically determining the expected behaviors for diverse kinds of systems. The strategy used is typically fine-tuned for the system under test, and there is no silver bullet for automatic oracle determination.

2.2 Automated test generation

There are several studies that seek to aid developers in the arduous and essential task of writing software tests [13]. Section 2.1 has summarized the literature on the automated generation and augmentation of oracles. We now present studies that propose mechanisms for the automated generation of whole tests, including their inputs and their oracles.

Test inputs can be generated randomly, such as with well-known Randoop [24, 44] that produces a sequence of method calls, each call appended after feedback from the preceding call sequence. Using the execution of this sequence, in conjunction with specifications for the classes under test, Randoop produces oracles that express expected and erroneous behaviors. Several tools combine symbolic and concrete executions, into a technique called dynamic symbolic execution or *concolic* testing [13]. Examples include DART [98] and CUTE [26] for C programs, the latter's counterpart jCUTE for Java programs [99], and PEX for .NET [100]. These tools use constraint solvers to generate inputs by assigning concrete values to symbolic variables, while maximizing code coverage. The oracles used within the generated tests are implicit, they check for crashes or violations of existing assertions. Instead of relying solely on random or symbolic execution approaches for the generation of complex objects to be used as test inputs, Thummalapenta and colleagues [101] mine sequences of method calls within the source code of the input project. Their approach, called MSeqGen, complements dynamic symbolic execution and random approaches to generate complex objects to be used as test inputs. They achieve higher coverage compared to Randoop and PEX. Fraser and Zeller propose μTEST [75], which relies on mutation analysis of an existing test suite, to generate new unit tests containing oracles that kill surviving mutants. This approach has since been incorporated into EvoSuite [25,64], a popular automated test generation technique. EvoSuite uses an evolutionary search algorithm to generate test inputs, optimizing for code coverage [102]. The generated oracles are further minimized, while killing the maximum number of mutants.

Not just oracles (cf. Section 2.1) but complete tests can also be derived from specifications. For embedded software in safety-critical systems, Wang *et al.* [103] exploit use case specifications and domain models to automatically generate system test cases. In doing so, they address the scalability issues that come with deriving tests from behavioral models such as activity and sequence diagrams. Their approach, based on natural language processing and constraint solving, generates automated tests with inputs based on the specified preconditions. The oracle in each generated test expresses the postconditions specified in the domain model. Liu and Nakajima [104] generate tests that verify the conformance of the implemented versions of information systems with their formal specifications. The goal of their approach is to cover all the documented functional scenarios of the target system, with oracles that express their formally specified post-conditions.

There are several approaches for the automated generation of tests for projects with rich user interfaces, such as desktop [105], mobile [106], and web [107] applications. However, many of these approaches use implicit oracles, such as crashes. FragGen by Yandrapally and Mesbah [108] generates tests for web applications, treating a page not as a single frame but fragmenting it into smaller entities, in order to detect redundant components among pages. Their approach is based on the development of a model for the application, from which the tests are derived. Assertion generation is based on the behavior inferred from the model and that observed during test execution, resulting in test failures in case there are severe differences between them.

Reflection: The techniques discussed here are useful for automatically generating inputs that result in high test coverage. However, several studies [18, 23, 39, 109] advocate for the representativeness [22, 104] of complex objects used as test inputs [110, 111], explicit oracles that are sensitive to faults, as well as better overall understandability of automatically generated tests [112–114]. Moreover, the test generation effort in these studies targets high structural coverage, and does not address the behavioral differences between test and field executions [19].

Learning-based test generation: Owing to their success in a large number of software engineering tasks [115], there is a growing body of work on the use of learning-based approaches for testing activities [27, 116, 117]. Many techniques have been proposed for the generation of automated tests using neural networks [118] and lately, Large Language Models (LLMs). Tufano *et al.* [119] present AthenaTest, which employs a transformer model trained on source code to produce unit tests. The authors also contribute a large corpus of tests and their corresponding focal methods, called Methods2Test, on which their model is fine-tuned. For projects in the Defects4J [120] benchmark, AthenaTest generates accurate and understandable unit tests with code coverage that is comparable with EvoSuite tests. These results are challenged by Alagarsamy *et al.* [121], who highlight the syntactical and assertion-related issues of the tests generated by AthenaTest. They implement A3Test, which is pre-trained on assertions, and ensures naming and syntactical consistencies when generating unit

tests with meaningful assertions. TestPilot [122] is an adaptation of Copilot by GitHub [123] that, when prompted with a JavaScript function and its documentation, proposes regression test cases that contain assertions, achieve high code coverage, and resemble human-written tests. Their approach requires an off-the-shelf LLM, without having to fine-tune it on code and test examples. For generated tests that fail, TestPilot can be re-prompted with details of the failure, for refined results. Vikram *et al.* [124] propose the use of documentation and two prompting strategies to generate property-based tests with LLMs. Their approach generates tests, including inputs as well as valid properties, as verified through mutation analysis. Evaluating their approach on methods from Python projects, they find that correct tests are generated for over 20% of the documented properties. CAT-LM [125] and ChatUnitTest [126] address the limited focal context available to LLMs for prompting. The former, trained on Java and Python projects, considers the relationship between test and source code files to generate new tests or complete existing ones. On the other hand, ChatUnitTest has capabilities for test generation, validation, and repair, and integrates into the development environment through plugins. Wang and colleagues [27] survey dozens of studies on test generation using pre-trained models. They highlight common themes, such as the need for fine-tuning and effective prompting, and identify the oracle problem as an open and important direction of further research.

Reflection: Conversational, learning-based agents can be valuable in the context of testing, especially when complementing human developers [109, 127–129]. However, researchers have highlighted several open problems with LLMs tasked with automated test generation [130, 131]. LLMs are also not proficient at determining the runtime behavior of programs [132], which implies that the tests they generate are not guaranteed to reflect realistic scenarios that may occur in the field.

Observation-based test generation: Most closely related to the contributions of this thesis are the relatively fewer works on test generation based on runtime observations. To motivate the need for observation-based testing, many of these studies cite the lack of overlap between behaviors that are tested in-house and behaviors that are exhibited by the system in the field. This is also the observation of Wang *et al.* [19], who demonstrate that field behaviors cover more code and kill more mutants than in-house tests. Wang and Orso [20] follow up on these findings in a short paper that proposes a technique called Replica for mimicking user behaviors within tests. For untested behaviors, Replica captures field data and supplements it with similar inputs produced through symbolic execution. Invariants for the system serve as the oracles, and the evaluation is based on the invariants covered by in-house tests versus those covered by the field-based inputs.

Capture (or record) and replay (C&R) is a related area of research. The premise of this approach is to monitor the sequence of events that occur as a system executes, and replay it offline so that the system is brought to the same

state within the test. jRapture [133] is a C&R tool that logs the interactions between the target application and its host system, and replays them within tests. GenUTest [134] observes the sequence of method calls to generate observation-based tests that use mocks. However, most C&R techniques typically emphasize on the captured test inputs, but not the oracle [89]. The related technique of *test carving* refers to the extraction of unit tests from larger tests. Elbaum and colleagues [135] carve unit tests from slow-running system tests, while Kampmann and Zeller [136] carve parameterized unit tests from system tests. MicroTest-Carver proposed by Deljouyi and Zaidman [114] produces unit tests from manual or automated end-to-end tests. A key limitation of the test carving approach is its reliance on system tests. Behaviors that are not represented within the system tests will also not be represented within the generated tests.

There are techniques that focus exclusively on deriving tests from real executions in the field. Bertolino *et al.* [21] survey 80 studies that propose strategies for field-based testing, grouping them into three categories. First, *ex-vivo* techniques use data from the field, for use within in-house tests inside the development environment [137]. Second, with *offline in-vivo* techniques, tests are run on a clone or a fork of the system, within the production environment [138]. Third, *online in-vivo* techniques can be likened to chaos engineering [139], and involve running tests directly on the target system in production, in parallel with normal execution [140–142]. A key finding from this survey is that the oracle problem is often overlooked among all three categories. The authors identify this as an essential direction for research on the topic.

Automated test generation from field observations has recently been adopted by Meta [143]. Closely aligned with our work, their approach, called TestGen, serializes the objects related to the invocation of methods, and uses them to generate regression tests. For non-void methods, the oracles within the generated tests verify the values observed in the field. For methods that do not return values, the oracles verify the absence of exceptions. Over a period of six months, the tests generated by TestGen have detected regressions in the codebase of Instagram.

Reflection: The fact that field behaviors differ from tested behaviors has come to be appreciated within the literature. Several studies propose field-based testing approaches, yet most focus on capturing inputs from the field. Automated oracle generation from production observations is under-represented among these studies. The generation of complete tests from field data – including inputs and oracles – has practical advantages [22].

3. Methodological Framework

"That rug really tied the room together."

The Big Lebowski

In this chapter, we present our approach for observing runtime behaviors with the goal of harnessing oracles that reflect the observed behaviors. First, Section 3.1 describes our key technical contribution, a framework that leverages runtime observations within test oracles. Next, in Section 3.2, we introduce the research questions we address in this thesis. Section 3.3 summarizes the protocol we use to answer our research questions, and Section 3.4 gives an overview of the projects we conduct our experiments on.

3.1 Framework for augmenting test oracles with runtime observations

In order to bridge the gap between tested behaviors and field behaviors, we propose a novel conceptual framework that observes runtime behaviors with the goal of utilizing them within tests. As illustrated in Figure 3.1, our framework receives three inputs related to a project: its source code, its test suite, and a runtime *workload* with which it can be exercised. The term workload refers to a set of representative operations that can be performed against a running system. If the project contains a test suite, the workload can refer to the execution of the tests in the test suite [103, 144]. In the context of this thesis, we utilize workloads that exercise the system in the field [19, 21], such as fetching directions from a routing application, or loading, editing, and saving a text file with a document processing program. We rely on field workloads motivated by the need to generate tests that are representative of field behaviors, as highlighted by Wang *et al.* [19]. The output from the framework is a set of focused tests for the project that contain an explicit oracle informed by runtime behaviors. Our proposed framework works as follows.

Test target selection: Phase (1) involves using the input source code to define a subset of target components that the generated tests will be associated with. These components may be selected based on developer-defined criteria. For example, the set of methods that have recently been updated make ideal

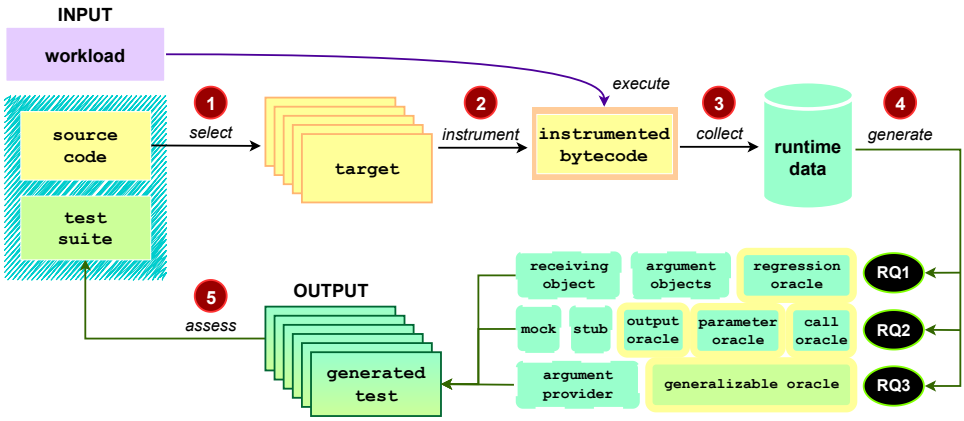


Figure 3.1: The general framework proposed in this thesis for augmenting test oracles with runtime observations. When the input project is exercised with a workload, our framework captures data at runtime and utilizes it to generate tests. We initialize this framework in distinct ways to answer our research questions (introduced in Section 3.2).

candidate targets [145]. Alternatively, we may deem as targets the methods in the source code that are inadequately tested, based on coverage [146, 147] or mutation analysis [148, 149] of the input test suite. It is essential to define a clear subset of methods to target for a focused monitoring campaign, that adds minimal overhead to the execution of the system at runtime.

Code instrumentation: Once the target methods are identified, phase (2) focuses on instrumenting them. The goal of the instrumentation is to add additional instructions that allow us to monitor the invocations of each target method, and configure what will be persisted at runtime, and how. The information that is available for us to harness at runtime is plentiful. For instance, we can monitor and store the sequence of method invocations [150] within a developer-defined class, and save snapshots of the objects associated with a target method [143]. We realize these requirements through dynamic instrumentation, without modifying the source code of the target methods, by preparing an agent. The agent includes instructions that are triggered when an event occurs at runtime. The agent is attached to the project when it is being deployed, such as through a `javaagent` for Java projects.

Runtime data collection: In the subsequent phase (3), the instrumented version of the program is executed with the input test and/or field workload. As we see from Figure 3.1, running the workload against the instrumented version of the program leads to the collection of runtime data on disk. Each invocation of a target method triggers the instructions within the instrumentation agent, which is configured with the exact protocol to be employed for data collection. For instance, the runtime data may be persisted on to a database, or within

files, serialized with the help of serialization libraries [151] in binary format, or as arguably more readable XML or JSON [152]. Regardless of the protocol employed for the serialization, it is useful to persist related objects such that they can be grouped together. For example, assigning identifiers to the receiving, argument, and returned objects can allow us to associate them with a single invocation of a target method. In this phase of data collection also, configurations made within the instrumentation agent can help minimize the overhead introduced from the serialization. One way to achieve this could be to set an upper limit on the number of instances of an event, such as a method invocation, to persist, or to save runtime data worth a certain amount of disk space [143].

Test generation: Eventually, after capturing the required kind and quantity of runtime data, the developer induces phase (4) offline (in-house) [21]. Based on the predetermined goal of the observation and data collection campaign, it is in this phase that the runtime data is utilized. Again, we can program the runtime data to be utilized in different ways. Figure 3.1 presents the three sets of artifacts, expressed as code elements, that we utilize for each of the research questions outlined in Section 3.2. The first set illustrates that the serialized receiving and argument objects can be utilized within a generated unit test to set up the test inputs, while the returned object can be encapsulated within an assertion statement, serving as a regression oracle. We explore the tests resulting from these artifacts in RQ1. Next, runtime data corresponding to the invocation of a target method, as well as nested method calls within this invocation, can be used to mock objects and stub them. Distinct aspects of their interactions can then be verified through dedicated *mock-based* oracles, which we discuss in RQ2. Alternatively, the set of arguments with which a target method is invoked can also be encapsulated within an argument provider method, to be used by a parameterized unit test [153]. RQ3 describes how developer-written oracles can be utilized within generated parameterized unit tests. In all three cases, the code artifacts produced are arranged into unit tests, which constitute the outputs from this crucial phase of our framework. These tests are compilable and executable, and can readily be appended to the existing test suite.

Test quality assessment: Finally, phase (5) involves incorporating the tests output from the previous phase into the test suite of the target project. Here, depending on the initial criteria used to select the target methods, an assessment can be performed to confirm that the new tests are useful additions. The generated tests contain inputs and oracles derived from the behaviors exhibited by the system at runtime, running against the input workload. Therefore, by design, they encapsulate representative behaviors that can complement developer-written unit tests.

Using the framework in the software development lifecycle: The framework of Figure 3.1 can be instantiated in different ways, depending on the specific goal of the observation-based test generation effort. For example, the component responsible for selecting targets can be tuned to cater to developer-defined crite-

ria, and the instrumentation agent can be configured to capture the runtime data as required. In order to minimize the performance costs incurred from capturing data at runtime, the framework can be employed periodically, when modifications are made to the project. In this thesis, we demonstrate the application of the framework for two use cases. First, developers can utilize the framework during manual, alpha-testing campaigns to ensure that mission-critical operations are represented in the generated tests. Second, for beta-testing campaigns in the field, the framework can be deployed to capture runtime behaviors triggered by real users, after ensuring that their privacy is not violated and sensitive data is not captured. In either case, the generated tests can be integrated into the test suite to complement the existing developer-written tests, or to bootstrap the creation of a test suite if one does not exist, as we do in Paper II.

3.2 Research questions

This thesis examines the implications of incorporating oracles that express observed runtime behaviors into the test suite of a software project. Our work addresses the following research questions.

RQ1 To what extent do oracles derived from runtime observations contribute to improved test quality?
This RQ assesses the extent to which observing a system as it executes, and harnessing these observations within automatically generated oracles, can complement the developer-written test suite of the system, per a specific adequacy criterion.

RQ2 To what extent do mock-based oracles derived from runtime observations contribute to regression testing?
The goal of this RQ is to determine whether different aspects of runtime behaviors can be faithfully recreated and verified within generated tests that make use of mocks. We also analyze the effectiveness of mock-based oracles in detecting regressions within the system.

RQ3 To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?
With this RQ, we analyze whether the oracles within developer-written unit tests are valid over a larger set of inputs captured while observing the system at runtime.

3.3 Experimental protocols

We initialize the general framework introduced in Section 3.1 to answer our three research questions listed in Section 3.2. This section describes our experimental protocol for each research question, applying the framework against the study subjects described in Section 3.4. For each question, we describe the criteria used to select the test generation targets, the runtime data captured for these targets,

the output from the framework, as well as the methodology employed for the assessment of the generated tests. These four aspects are also summarized in Table 3.1.

Protocol for RQ1

We answer RQ1 by initializing our general framework in a tool called PANKTI, evaluating it against three study subjects exercised with their field workloads as detailed in Section 3.4, PDFBox, BroadLeaf, and Jitsi. Note that this protocol can be adapted to target other source artefacts for the automated generation of tests. For example, we design a tool called AutoGraphQL that generates unit tests by capturing incoming requests that arrive at the GraphQL API endpoint, and oracles using the developer-defined GraphQL schema of web applications. Paper II describes AutoGraphQL and its evaluation in detail. In this thesis, we answer RQ1 using the evaluation conducted with PANKTI in Paper I.

Targets: The target methods for test generation are based on an adequacy criterion based on mutation analysis, defined on the original test suite of each project. Specifically, as indicated in Table 3.1, we target methods that are *pseudo-tested* [76, 77] for our evaluation. Such methods are covered by the test suite, meaning that they are invoked at least once during the execution of the existing tests. However, when the body of a pseudo-tested method is replaced by a statement returning a default value, none of the tests that cover the method detect this extreme mutation and fail. This indicates the lack of an explicit oracle for verifying the output of this method, rendering them inadequately tested. For example, the `PlanetExpress` class of Listing 3.1 defines two methods `calcBaseFare` and `makeReservation`. A developer-written test `testThatReservationIsSuccessful` indirectly covers `calcBaseFare`, through the invocation of `makeReservation` on line 19. However, removing the original body of `calcBaseFare` (lines 3-4) and simply returning the value `0.0` (line 5) from it does not cause the test to fail. This makes `calcBaseFare` pseudo-tested, and therefore a target for test generation with PANKTI.

Runtime data: Per the framework of Figure 3.1, PANKTI instruments each target method, adding instructions that facilitate the persistence of associated runtime data. Exercising the system with a workload in the field triggers the invocation of the target pseudo-tested methods. Corresponding to each of these invocations, PANKTI captures an *object profile*, which includes the object on which a target method is invoked, i.e., its *receiving object*, the objects that are passed to this invocation as the arguments, as well as the object returned from this invocation. PANKTI serializes these objects as XML, and saves them to disk.

Output: The captured object profiles are utilized to generate unit tests for each target method, that reflect the behaviors observed at runtime. For our example of `calcBaseFare` in, the test generated by PANKTI is presented on lines 22 to 33 of Listing 3.1, using the receiving, argument, and returned objects captured for one invocation of the pseudo-tested method. Each test generated by PANKTI follows the *Arrange-Act-Assert* pattern [154]. The arrange phase recreates the

```
1  public class PlanetExpress {
2    public double calcBaseFare(String source, String destination) {
3      ...
4      return fare;
5      return 0.0;
6    }

8    public ReservationStatus makeReservation(...) {
9      ...
10     double fare = calcBaseFare(source, destination);
11     return ...;
12   }
13 }
14 ..........................................................................................
15 // Developer-written test that covers calcBaseFare(String, String)
16 @Test
17 public void testThatReservationIsSuccessful() {
18   ...
19   assertEquals(..., planetExpress.makeReservation(...));
20 }
21 ..........................................................................................
22 // Test generated automatically by PANKTI  for calcBaseFare(String, String)
23 @Test
24 public void testForCalcBaseFare() {
25   // Arrange
26   PlanetExpress planetExpress = deserialize(
         "<pe.planet.express>...</pe.planet.express>");

28   // Act
29   double fare = planetExpress.calcBaseFare("Earth", "Moon");

31   // Assert
32   assertEquals(42.42, fare, 0.0);
33 }
```

Listing 3.1: **(RQ1)** The method `calcBaseFare` defined in the class `PlanetExpress` is covered by the test `testThatReservationIsSuccessful`, but is pseudo-tested. PANKTI generates the test `testForCalcBaseFare` for `calcBaseFare` using runtime observations, as a result of which it becomes well-tested.

receiving object (line 26) and the arguments, deserializing them from their serialized representation. In the act phase of the test (line 29), the target method is invoked on the recreated receiving object, passing it the recreated arguments. Finally, the oracle in the generated test verifies, through the assertion statement on line 32, that the output from this invocation is equal to the object returned at runtime. The unit test generated for each target method also serves as a regression test, expressing its behavior as observed at runtime, and capable of detecting changes to this behavior as the project evolves.

Assessment: We analyze the usefulness of the tests generated by PANKTI with respect to the original adequacy criterion. Each target method which is no longer pseudo-tested as a consequence of the generated tests represents a successful case. In our example, the extreme mutation of `calcBaseFare` will cause `testForCalcBaseFare` to fail, making the method well-tested. We present the results for this first RQ in Section 4.1.

Protocol for RQ2

A *mock* is a fixture used within a unit test to replace an actual object with a skeletal implementation [155]. Mocking allows for more focused testing, removing side effects from operations on external objects [156, 157]. The behaviour of mocks is configured using *stubs* [158]. Stubbing is a mechanism for returning outputs from a mock when it is triggered with a specific input, bypassing an actual operation. In order to answer RQ2, we initialize our framework of Figure 3.1 into a technique called RICK, which generates unit tests with mocks, stubs, and *mock-based* oracles. Paper IV presents the evaluation of RICK against GRAPHHOPPER, GEPHI, and PDFBOX, exercised with the field workloads described in Section 3.4.

Targets: Note from Table 3.1 that the targets for test generation with RICK are methods that invoke *mockable* methods. A mockable method refers to a method that is called within the body of a target method, on a parameter or field object of a type that is external to the declaring type of the target method. We illustrate this through the example of the target method `createIssue` defined in the class `Catalog` in Listing 3.2. This target method calls the method `register` on an object of type `AdService` (line 5), which is declared as a field within `Catalog` (line 2). Additionally, `createIssue` calls two methods, `getAvailability` (line 7) and `dispatch` (line 8), on the object of type `PrintService`, which is one of the parameters of `createIssue` (line 4). This makes these three nested method calls to `register`, `getAvailability`, and `dispatch` mockable within the target method `createIssue`.

Runtime data: Per Table 3.1, RICK captures data about each target method, and its corresponding mockable methods, by instrumenting them to monitor their invocations at runtime. As with PANKTI, for one invocation of a target method, RICK captures its receiving object, its arguments, as well as the object it returns. Additionally, for each mockable method invocation that occurs within the invocation of the target method, RICK serializes the arguments, the returned object, as well as the sequence and frequency with which these nested calls occur.

```
1  public class Catalog {
2    AdService adService;
3    ...
4    public boolean createIssue(String theme, PrintService service) {
5      adService.register(theme); // mockable method #1
6      ...
7      if (service.getAvailability() >= 1) { // mockable method #2
8        service.dispatch(numIssues); // mockable method #3
9      }
10     ...
11     return issueStatus;
12   }
13 }
```

Listing 3.2: **(RQ2)** The target method `createIssue` contains three mockable method calls on objects of external types.

Output: Using the data captured at runtime, RICK generates unit tests that mock the external types, stub their behavior, and contain three kinds of mock-based oracles. First, the *output oracle*, OO, verifies the equality of the output returned from the target method within the test with the returned object captured in the field. Next, the *parameter oracle* or PO verifies the arguments with which the mock method calls occur within the target method. The third oracle is the *call oracle* (CO), which verifies the order and frequency of these mock method calls within the target method. Each mock-based oracle thus verifies a unique aspect of the captured runtime behavior. Listing 3.3 presents the tests generated by RICK for the target method `createIssue` of Listing 3.2, using the observations made from one of its invocations in the field. Note that the arrange and act phases (lines 3 to 10) of Listing 3.3 are common to the three generated tests, `testForCreateIssue_OO` (lines 16-21), `testForCreateIssue_PO` (lines 23-30), and `testForCreateIssue_CO` (lines 32-40). Each generated test first reconstructs the receiving object by deserializing it (line 4). This is followed by statements that mock the external field (line 5) and parameter (line 6) types. Next, the mocks are configured by stubbing them using observed inputs and outputs. The statement on line 7 stubs `mockPrintService` to directly return `3` when `getAvailability` is invoked on it, without the actual invocation of `getAvailability`. The target method is then invoked on the receiving object in the act phase of the generated tests, as on line 10. Finally, each mock-based oracle, OO, PO, and CO, verify the target method output, the arguments of the mock method calls, and the sequence and frequency of these calls, respectively.

Assessment: In order to answer RQ2, and determine if production observations can be faithfully recreated through mocks, we execute the tests generated by RICK. Each passing test represents a successful case, where RICK has correctly captured field behaviors, and expressed them within the generated test, including the reconstruction of the objects, the configuration of the mocks through stubs, as

```
@Test
public void testForCreateIssue_XO() {
  // Arrange
  Catalog catalog = deserialize("<jpeter.man.catalog>...</jpeter.man.catalog>");
  AdService mockAdService = mockField_adService_InCatalog(catalog);
  PrintService mockPrintService = mock(PrintService.class);
  when(mockPrintService.getAvailability()).thenReturn(3);

  // Act
  boolean status = catalog.createIssue("HATS", mockPrintService);

  // Assert
  ...
}
.....................................................................................
@Test
public void testForCreateIssue_OO() {
  ...
  // Assert
  assertTrue(status);
}
.....................................................................................
@Test
public void testForCreateIssue_PO() {
  ...
  // Assert
  verify(mockAdService, atLeastOnce()).register("HATS");
  verify(mockPrintService, atLeastOnce()).getAvailability();
  verify(mockPrintService, atLeastOnce()).dispatch(42);
}
.....................................................................................
@Test
public void testForCreateIssue_CO() {
  ...
  // Assert
  InOrder orderVerifier = inOrder(mockAdService, mockPrintService);
  orderVerifier.verify(mockAdService, times(1)).register(anyString());
  orderVerifier.verify(mockPrintService, times(1)).getAvailability();
  orderVerifier.verify(mockPrintService, times(1)).dispatch(anyInt());
}
```

Listing 3.3: **(RQ2)** Using runtime data captured from the invocations of the target method `createIssue` and its mockable method calls (Listing 3.2), RICK generates unit tests with mocks, stubs, and mock-based oracles: OO, PO, or CO.

well as the assertion and verification statements that constitute the mock-based oracles. Furthermore, we conduct mutation analysis [159] to assess the effectiveness of the three kinds of mock-based oracles produced by RICK at detecting regressions in the system. Section 4.2 presents the results from this evaluation.

Protocol for RQ3

To answer RQ3, we initialize the framework of Figure 3.1 into a mechanism called PROZE, evaluate it using the test and field workloads of PDFBox and SAML (introduced Section 3.4), and present the evaluation in Paper III. PROZE transforms existing developer-written unit tests into *parameterized unit tests* [153]. A parameterized unit test (PUT) is a kind of unit test that accepts at least one parameter. The oracle within a PUT is comparatively more general, as it holds for all values supplied to its parameters [160]. When a PUT is run, its parameters are sequentially assigned values by an associated *argument provider*, which is typically implemented as a method.

Targets: As noted in Table 3.1, PROZE selects as targets the methods that are called directly by a developer-written unit test. Consider the developer-written unit test called `testOneGalaxy` presented in Listing 3.4. The test does not have any parameters, and contains three assertion statements. The input within this test includes the invocation of the method `locateObject` with the `String` argument `M51a` (line 4). PROZE identifies the `locateObject` method as a target as it is directly invoked within the test.

Runtime data: PROZE instruments each target method, with the goal of capturing the arguments that are passed to it whenever it is invoked during the execution of the test suite, or in the field as a consequence of the field workload. For example, during the execution of all the developer-written tests, PROZE finds that the target method `locateObject` is invoked with two unique `String` arguments, `M51a` (the original argument passed to it within `testOneGalaxy`), and `M63`. Additionally, PROZE observes that `locateObject` is invoked with the arguments `M100`, `M101`, and `M106` in the field. PROZE captures all five of these arguments.

Output: PROZE generates the argument provider method, using the set of arguments captured for a target method at runtime, across test and field executions. Each generated PUT is parameterized over a single target method. The arguments passed to the PUT by the argument provider are assigned as the arguments to the target method. Moreover, as is true for `testOneGalaxy` in Listing 3.4, a developer-written unit test may have multiple oracles, expressed through multiple assertion statements. In order to discover which of these oracles is generalizable to a wider input range, PROZE isolates each oracle into its own PUT. Lines 10 to 16 of Listing 3.4 presents one of the PUTs `testMultipleGalaxies` generated by PROZE from `testOneGalaxy`, as well as its argument provider method `provideGalaxyID` (lines 18 to 27). The PUT has a single parameter `identifier` of type `String` (line 12), which is assigned as the argument of the target method `locateObject` (line 14). Note that the PUT contains only one of the three assertion statements of the original test (line 15). The argument provider method includes all the

```java
@Test
public void testOneGalaxy() {
  MessierCatalog catalog = MessierCatalog.initialize();
  MessierObject object = catalog.locateObject("M51a");
  assertEquals("Whirlpool", object.getName());
  assertEquals(CANES_VENATICI, object.getConstellation());
  assertTrue(catalog.getSpiralGalaxies().contains(object));
}
.....................................................................................
@ParameterizedTest
@MethodSource("provideGalaxyID")
public void testManyGalaxies(String identifier) {
  MessierCatalog catalog = MessierCatalog.initialize();
  MessierObject object = catalog.locateObject(identifier);
  assertEquals(CANES_VENATICI, object.getConstellation());
}

// Argument provider for the PUT
private static Stream<Arguments> provideGalaxyID() {
  return Stream.of(
    Arguments.of("M51a"), // original argument
    Arguments.of("M63"),
    Arguments.of("M100"),
    Arguments.of("M101"),
    Arguments.of("M106")
  );
}
```

Listing 3.4: **(RQ3)** A developer-written unit test, `testOneGalaxy`, calls the target method `locateObject` and contains three assertions to verify the details of a single test input. PROZE reuses the second assertion within the generated parameterized unit test `testManyGalaxies`, which is parameterized over `locateObject`. The `String` parameter of this test is supplied values captured at runtime, by the argument provider `provideGalaxyID`.

five `String` values captured during test and field invocations of `locateObject`, including the argument in the original test, `M51a`. When `testMultipleGalaxies` is run, it is supplied with each of these five values, and the assertion statement is evaluated against each of them.

Assessment: By design, each test generated by PROZE contains a developer-written oracle. In order to answer RQ3 and determine if runtime observations can facilitate the generalization of existing developer-written oracles, we run these generated tests. As mentioned in Table 3.1, we classify the oracle of a PUT as being *strongly-coupled*, *decoupled*, or *falsifiably-coupled*. The oracle of a PUT is strongly-coupled if it holds only for the original test input, and no other input supplied by the argument provider. On the other hand, if the PUT passes for all inputs supplied to it by the argument provider, we conclude that the oracle is decoupled with the target method. An oracle is falsifiably-coupled if it is valid for a subset of the arguments supplied by the argument provider, while failing

Table 3.1: Summary of our experimental protocol for RQ1, RQ2, and RQ3. We initialize the infrastructure of Section 3.1 to select different kinds of TARGETS, and capture diverse RUNTIME DATA, with the goal of producing the OUTPUT that addresses the three RQs, using relevant ASSESSMENT metrics.

	TARGETS	RUNTIME DATA	OUTPUT	ASSESSMENT
RQ 1	pseudo-tested methods	receiving object, argument objects, returned object	unit tests with **regression oracle** verifying returned object	method status: pseudo-tested vs. well-tested
RQ 2	methods and mockable methods called by them	for methods: receiving object, argument objects, returned object, and for mockable methods: argument objects, returned object, method call sequence	unit tests with mocks, stubs, and mock-based oracle **(output oracle, parameter oracle, call oracle)**	faithful reproduction of production behavior, and mutation analysis
RQ 3	methods called directly by developer-written unit tests	argument objects	argument providers and parameterized unit tests with **generalizable, developer-written oracle**	strongly-coupled, decoupled, and falsifiably-coupled oracles

for the others. This indicates an oracle that is demonstrably able to distinguish between valid and invalid inputs. For example, the oracle, and consequently the PUT, of Listing 3.4 passes for three of the five inputs, since `M51a`, `M63`, and `M106` all belong to the same constellation of `CANES_VENTICI`, whereas the two others do not. This means that we have succeeded in discovering an existing oracle that is generalizable, as well as falsifiably-coupled with the input supplied to the target method `locateObject`. We present the details of our evaluation of PROZE, and the answer to RQ3 in Section 4.3.

Summary of experimental protocol

Table 3.1 highlights the four key aspects of the protocol we use to answer our three research questions.

3.4 Study subjects and workloads

In order to answer our three research questions, we conduct experiments on real-world software projects, exercising them with representative field workloads. Designing these workloads is an essential aspect of our evaluation, since the runtime observations we utilize for test generation are sourced from them. In this section, we list these projects, in alphabetical order, presenting their summary and describing their workloads.

1. BroadLeaf: An open-source Java project, BroadLeaf is an e-commerce application implemented using the Spring Boot framework. It includes features for product, user, and order management. As of this writing, BroadLeaf has over 80 contributors on GitHub. For our experiments with BroadLeaf in the field for Paper I, we design a representative e-commerce workload, running its server and interacting with its front-end.

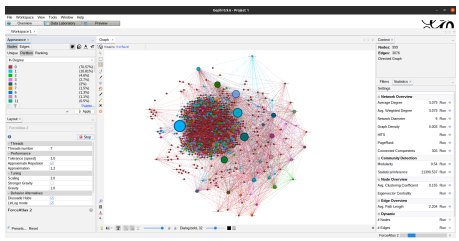
2. Frontapp: We use Frontapp as an industrial study subject for our experiments in Paper II. Frontapp is the primary, user-facing website developed and maintained by RedEye AB, an investment banking company based in Stockholm. Its backend server, developed in PHP, handles requests through both REST and GraphQL API endpoints. The runtime data we captured for our experiments were GraphQL query requests, triggered by actual user traffic over a period of 33 days.

3. Gephi: A GUI application for working with graph data, the Gephi project has nearly 90 contributors on GitHub. We use Gephi for our evaluation in Paper IV. For our experiments, we import a graph dataset of the artifacts published on Maven Central [161] into a running instance of Gephi, compute statistics on it, such as its average degree and density, manipulate its layout (pictured in Figure 3.2a), and export the resulting graph as PDF and SVG.

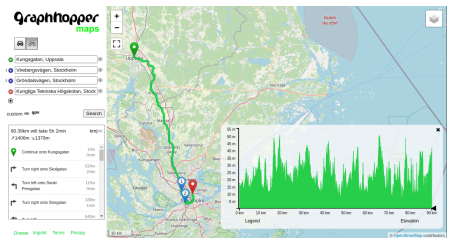
4. GraphHopper: Well-known and open-source, GraphHopper is a Java application for fetching the route between different coordinates on a map, using diverse modes of transportation. The repository of the project on GitHub has over 100 contributors. We use GraphHopper for our experiments in Paper IV, requesting for the car and bike routes between several locations in Sweden (as depicted in Figure 3.2b).

5. Jitsi: Jitsi is an open-source video-conferencing platform implemented in Java. The project repository on GitHub has nearly 500 contributors. We use Jitsi, monitoring targets within its `jicofo` component, as a study subject in Paper I, conducting an hour-long video call among the authors, also sending messages in the chat box (Figure 3.2c).

6. PDFBox: A project from the Apache Software Foundation, PDFBox is a library and command-line tool for PDF manipulation. It supports most PDF



(a) Gephi



(b) GraphHopper

(c) Jitsi

Figure 3.2: Snapshots from the field — we exercise the study subjects with realistic workloads in order to capture data that reflect their runtime behaviors.

generation and manipulation operations, including conversion from and to text and images, encryption, or merging and splitting, among others. We use PDFBox as a study subject in Papers I, III, and IV, using it to perform 10 representative operations on 5 real-world documents from [162].

7. SAML: An extension of the WSO2 identity server, SAML provides functionalities for performing user authentication using the Security Assertion Markup Language (SAML) protocol. The project has nearly 100 contributors on GitHub. For our evaluation in (Paper III), we work with two modules of SAML, called `query` and `sso`, signing in and out across different applications.

8. Saleor: Saleor is an e-commerce application with a backend server implemented in Django, a frontend based on TypeScript, and a GraphQL API to handle HTTP requests. More than 250 developers have contributed to the GitHub repository of Saleor. For our experiments with it in Paper II, we interact with its frontend, performing actions that are typical of e-commerce platforms for both users and administrators.

4. Experimental Results

"Roads go ever ever on,
over rock and under tree,
by caves where never sun has
shone, by streams that never find
the sea."

J.R.R. Tolkien, The Hobbit

This chapter presents the answers to our three research questions, per their protocol detailed in Chapter 3. We conclude this chapter with a summary of our findings in Section 4.4.

4.1 RQ1: To what extent do oracles derived from runtime observations contribute to improved test quality?

Per the protocol described in Section 3.3, we initialize our general framework explained in Section 3.1 into a test generation pipeline called PANKTI. We evaluate PANKTI against BROADLEAF, JITSI, and PDFBOX, exercising them with the representative field workloads outlined in Section 3.4. We rely on extreme

Table 4.1: Summary of results for RQ1: PANKTI generates TESTS for pseudo-tested TARGETS using observations made at runtime, as the three PROJECTS run in the field. Of these 86 target methods, 61.6% are no longer pseudo-tested, as a consequence of the oracles generated by PANKTI.

PROJECT	TARGETS	TESTS	PASSING	FAILING	TARGET STATUS	
					PSEUDO-TESTED	WELL-TESTED
BROADLEAF	11	351	244	107	5 / 11	**6 / 11**
JITSI	29	20	20	0	10 / 29	**19 / 29**
PDFBOX	46	13,851	13,614	237	18 / 46	**28 / 46**
TOTAL	86	14,222	13,878 (97.6%)	344 (2.4%)	33 / 86 (38.4%)	**53 / 86 (61.6%)**

mutation analysis with Descartes [77] for selecting the targets for test generation with PANKTI. Descartes reports the methods that are pseudo-tested by the developer-written test suite of each project.

Table 4.1 summarizes the results from our evaluation. In total, we target 86 pseudo-tested methods across the three projects, which include 11 methods in BROADLEAF, 29 in JITSI, and 46 in PDFBOX. Recall from the example of Listing 3.1 that these methods are covered by at least one test, but their behavior is insufficiently specified by all existing developer-written oracles. PANKTI instruments each of these 86 target methods in order to monitor their invocations at runtime, triggered by the field workload of the system. Corresponding to each invocation, PANKTI captures the receiving object, the argument objects, as well as the object returned from the invocation. These objects are serialized to disk as XML using the XStream serialization library. Per the framework outlined in Figure 3.1, these serialized runtime objects are then utilized by PANKTI to generate unit tests for the target methods. In total, PANKTI generates 14,222 unit tests from observing the invocations of the 86 methods at runtime, as highlighted in the last row of Table 4.1. Adding these generated tests to the test suite of the three projects, and conducting extreme mutation analysis with Descartes on the updated test suite, we find that 53 of the 86 target methods (61.6%) are no longer pseudo-tested. This means that extreme mutation of these 53 methods is detected by the explicit oracles generated for them by PANKTI.

We illustrate with the example of the target method `getLeftSideBearing`, defined in the class `HorizontalMetricsTable` of PDFBOX, presented on lines 1 to 12 of Listing 4.1. This method is covered by five developer-written tests, but is pseudo-tested by the test suite of PDFBOX. Consequently, PANKTI identifies `getLeftSideBearing` as a target, and instruments it in order to capture objects associated with it, when it gets invoked at runtime. Using each captured *object profile*, i.e., a unique combination of receiving, argument, and returned objects observed in the field, PANKTI generates 360 unit tests for `getLeftSideBearing`. Figure 4.1 showcases one of these object profiles captured by PANKTI at runtime, which includes a snapshot of the receiving object of type `HorizontalMetricsTable`, the `int` argument `49`, as well as the returned `int` value, `152`. Lines 14 to 25 of Listing 4.1 present the test generated from this object profile of `getLeftSideBearing`. On line 17, the `HorizontalMetricsTable` receiving object is reconstructed from its captured state, by reading a resource file and deserializing its contents. Next, line 18 initializes the `int` argument to `49`. The receiving object and the argument constitute the test input. On line 21, `getLeftSideBearing` is invoked on the receiving object `hMTable`, with the same `gid` as it was called with in the field. Finally, using the explicit oracle on line 24, the equality of the `int` returned from this invocation of the target method, and the one captured at runtime, i.e., 152, is verified. We add the 360 tests generated by PANKTI for `getLeftSideBearing` to the test suite of PDFBOX. All of these tests pass, and Descartes no longer reports `getLeftSideBearing` as being pseudo-tested, owing to the assertion statement within each generated test that clearly specifies its expected behavior.

```
1  public class HorizontalMetricsTable {
2    short[] leftSideBearing;
3    int numHMetrics;
4    ...
5    public int getLeftSideBearing(int gid) {
6      if (gid < numHMetrics)
7        return leftSideBearing[gid];
8      else
9        return nonHorizontalLeftSideBearing[gid - numHMetrics];
10   }
11   ...
12 }
13 ..........................................................................................
14 @Test
15 public void testGetLeftSideBearing {
16   // Arrange
17   HorizontalMetricsTable hMTable = deserialize("getLeftSideBearing-receiving.xml");
18   int gid = 49;

20   // Act
21   int leftSideBearing = hMTable.getLeftSideBearing(gid);

23   // Assert
24   assertEquals(152, leftSideBearing);
25 }
```

Listing 4.1: **(RQ1)** Target method `getLeftSideBearing` defined in the class `HorizontalMetricsTable` of PDFBox is pseudo-tested. Observing one of its invocations in the field, PANKTI generates a unit test with an explicit oracle, which makes `getLeftSideBearing` well-tested.

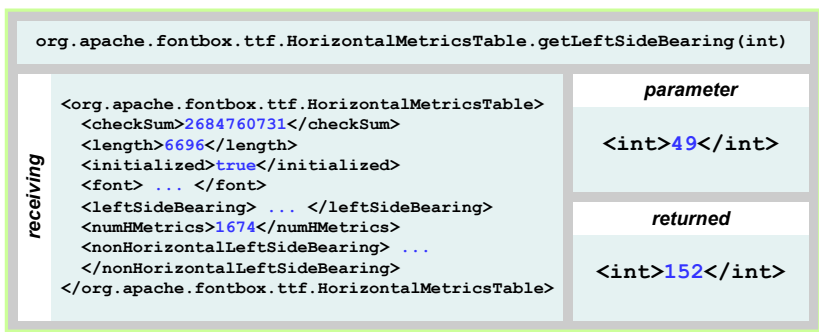


Figure 4.1: An object profile captured by PANKTI for one invocation of the target method `getLeftSideBearing` (lines 5 to 10 of Listing 4.1) in the field.

Per Table 4.1, of the 14,222 tests generated by PANKTI for the three projects, 97.6% pass. The 344 generated tests that fail do so for two key reasons. First, PANKTI does not override the `equals` method for arbitrary objects, which causes object comparison within the generated assertion to fail. Second, deserialization from captured object states is not guaranteed to reconstruct the exact object observed in the field, especially with regards to external resources and transient fields. Furthermore, 33 of the 86 target methods remain pseudo-tested, despite passing tests generated by PANKTI. We attribute this non-improvement in the test quality of these target methods to mutant equivalence. For example, extreme mutation of a target method that returns a non-primitive object implies replacing the body of the method with a `return null` statement. If the method always returns `null` at runtime, this mutation would go undetected by the tests generated by PANKTI.

To what extent do oracles derived from runtime observations contribute to improved test quality?

We devise a mechanism called PANKTI, which observes the invocations of target methods, recording their receiving and argument objects, as well as the object returned from them. PANKTI uses this captured data to generate unit tests for target methods that reflect their behaviors observed in the field. Our results demonstrate that the explicit oracles contained in the generated tests contribute to the improvement in test quality for a majority of the target methods. Extreme mutation of these methods, by removing their body, is no longer undetected, thanks to the oracles generated by PANKTI. We conclude that runtime observations can be used to generate oracles that augment the test quality of the system under test.

4.2 RQ2: To what extent do mock-based oracles derived from runtime observations contribute to regression testing?

We initialize our framework from Section 3.1 into a mechanism called RICK, which observes invocations of target methods as well as the mockable methods within them. The unit tests generated by RICK contain mocks, stubs, and mock-based oracles, that correspond to the data captured in the field.

Following the protocol described in Section 3.3, we evaluate RICK against GEPHI, GRAPHHOPPER, and PDFBOX to answer our second research question. Table 4.2 summarizes the results of this evaluation. In total, RICK targets 128 methods across the three projects, observing one invocation of each method, as well as the invocations of its associated mockable method calls, to generate 294 unit tests. Let us consider the case of GRAPHHOPPER, which has 23 target methods that are triggered in the field. RICK observes one invocation of each

Table 4.2: Summary of results for RQ2: RICK selects methods from three PROJECTS as TARGETS for the generation of unit tests. The number of statements corresponding to the three kinds of mock-based oracles in the generated tests are represented by the columns OO, PO, and CO. A majority of the generated tests (52.4%) SUCCESSFULLY MIMIC, end-to-end, the observed runtime behaviors.

PROJECT	TARGETS	MOCK OBJECTS	MOCK METHODS	STUBS	STATEMENTS			TEST RESULTS		
					OO	PO	CO	UNHANDLED BEHAVIOR	INCOMPLETELY MIMIC	SUCCESSFULLY MIMIC
GEPHI	57	67	93	103	12	94	135	56 / 126	26 / 126	**44 / 126**
GRAPHHOPPER	23	31	45	81	16	88	78	5 / 62	20 / 62	**37 / 62**
PDFBOX	48	53	66	38	10	75	80	22 / 106	11 / 106	**73 / 106**
TOTAL	128	151	204	222	38	257	293	83 / 294 (28.2%)	57 / 294 (19.4%)	**154 / 294 (52.4%)**

of these methods to generate 62 unit tests. Within these tests, 31 objects of external types are mocked, on which 45 methods are called. Moreover, based on observations made at runtime, the mock objects are stubbed using their captured arguments and returned values. There are 81 stubs across the 62 generated tests. Recall from Section 3.3 that RICK generates three kinds of mock-based oracles. The output oracle (OO) verifies the expected output from the target method, the parameter oracle (PO) verifies the arguments passed to the mock methods, while the call oracle (CO) verifies the sequence and frequency of these mock method calls. The OO is expressed as a JUnit5 assertion statement, while the PO and CO are specified through the verification API of Mockito [163, 164]. As highlighted in Table 4.2, the 62 tests generated by RICK for GRAPHHOPPER contain 16 OO statements, 88 PO statements, and 78 CO statements.

We illustrate with the example of Listing 4.2, which presents an excerpt of the class `LineIntIndex` from GRAPHHOPPER. This class declares the field `dataAccess` of type `DataAccess` on line 2, and includes, from lines 4 to 14, the target method `loadExisting`. Within `loadExisting`, there are four calls to two distinct mockable methods on `dataAcess`. The mockable method `loadExisting` is called on line 6, which is followed by three calls to the mockable method `getHeader` with different `int` arguments (lines 9 to 11). RICK observes one invocation of the target method `loadExisting` in the field, and captures its `LineIntIndex` receiving object, as well as the `boolean` value returned value from it. Additionally, within this invocation, RICK also captures the data corresponding to the mockable call to `loadExisting` (i.e., the returned `boolean` value) and the three calls to `getHeader` (i.e., their argument and returned `int`s).

Offline, RICK processes the data captured in the field in order to generate

```
1  public class LineIntIndex {
2    DataAccess dataAccess;
3    ...
4    public boolean loadExisting() {
5      ...
6      if (!dataAccess.loadExisting())
7        return false;
8      ...
9      GHUtility.checkDAVersion(..., dataAccess.getHeader(0));
10     checksum = dataAccess.getHeader(1 * 4);
11     minResolutionInMeter = dataAccess.getHeader(2 * 4);
12     ...
13     return true;
14   }
15 }
```

Listing 4.2: **(RQ2)** Target method `loadExisting` defined in the class `LineIntIndex` of GraphHopper has mockable method calls on the `dataAccess` field, `loadExisting` on line 6 and `getHeader` on lines 9 to 11.

tests. Specifically, the data corresponding to one invocation of a target method is transformed into three unit tests, one for each kind of mock-based oracle. The generated tests include the inputs in the form of the deserialized and receiving argument objects, as well as the mocks and stubs for the external objects. For the target method `loadExisting` of Listing 4.2, RICK generates three tests using the data captured from its invocation. Each test contains one of the mock-based oracles OO, PO, and CO. The common *Arrange* and *Act* phases of these tests are presented in Listing 4.3. As part of the Arrange phase, the receiving `LineIntIndex` object is first deserialized and reconstructed to its observed production state (line 4). On line 5, `DataAccess` is mocked into the `mockDataAccess` object, and inserted as a mocked field within `lineIntIndex`. Next, based on their observed invocations, `loadExisting` and `getHeader` are stubbed, i.e., they are configured to return the observed values given the captured arguments, if any (lines 6 to 9). In the Act phase, the target `loadExisting` method is invoked on `lineIntIndex`. The Assert phase in the test with the output oracle (line 22) verifies that the returned `boolean` value from `loadExisting` is `true`, through the JUnit5 `assertTrue` method. Next, the test with the parameter oracle uses the `verify` API of Mockito (lines 29 to 32) to confirm the arguments with which the mock method calls to `loadExisting` and `getHeader` occur, when invoked on `mockDataAccess`. Finally, the test with the call oracle uses Mockito's `inOrder` to verify that `loadExisting` is first called on `mockDataAcess` once (line 40), followed by three invocations of `getHeader` (line 41). Running the three tests, we find that the OO, PO, and CO all hold. This implies that RICK has successfully transformed runtime observations from the invocation of `loadExisting` into mocks, stubs, and assertion and verification statements.

Overall, as summarized in Table 4.2, 52.4% of the tests generated by RICK

pass, demonstrating its capability to generate tests that SUCCESSFULLY MIMIC observed runtime behaviors through mocks. With respect to the INCOMPLETELY MIMIC and UNHANDLED BEHAVIOR cases of Table 4.2, we find that the generated tests that are not successful at recreating the observed runtime behaviors. We attribute these test failures to incomplete (de)serialization, and side-effects of mocking. We elaborate on these cases in Paper IV.

```
// (Extreme) mutant #1: Lines 5 to 13
public boolean loadExisting() {
  ...
  return true;
  return true;
}

// (Extreme) mutant #2: Lines 5 to 13
public boolean loadExisting() {
  ...
  return true;
  return false;
}

// Mutant #3: Line 6
if (!dataAccess.loadExisting())
if (dataAccess.loadExisting())

// Mutant #4: Line 10
checksum = dataAccess.getHeader(1 * 4);
checksum = dataAccess.getHeader(1 / 4);

// Mutant #5: Line 11
minResolutionInMeter = dataAccess.getHeader(2 * 4);
minResolutionInMeter = dataAccess.getHeader(2 / 4);
```

Listing 4.4: **(RQ2)** Five first-order mutants are introduced in the target method `loadExisting`. The line numbers correspond to the line numbers in Listing 4.2.

We conduct mutation analysis in order to assess the effectiveness of the successful mock-based oracles generated by RICK at detecting regressions. We rely on LittleDarwin [165] to introduce first-order mutants in each target method, before re-running the generated tests to determine if the OO, PO, and CO detect the regression. The five mutants produced for the target method `loadExisting` of Listing 4.2 are presented in Listing 4.4. The OO test generated by RICK detects 2 of these mutants (#2 and #3), as the assertion on line 22 of Listing 4.3 fails on an output that differs from the expected `boolean` value. On the other hand, the CO of lines 39 to 41 kills 3 mutants (#1, #2, and #3), as the invocations to the methods `loadExisting` and `getHeader` are expected on the mocked `DataAccess` object, but do not occur due to the mutation. The verification statements in the PO (lines 29-32) kill all 5 mutants. This is because the expected mock method invocations within the MUT either do not occur entirely, or occur with unexpected arguments, causing the PO test to fail. Overall, we introduce 449 mutants in the target methods in GEPHI, GRAPHHOPPER, and PDFBOX. The mock-based oracles generated by RICK kill 210 of these mutants, and as we see from the example of `loadExisting`, the OO, PO, and CO complement in each other in their ability to detect regressions.

```
@Test
public void testForloadExisting_XO {
  // Arrange
  LineIntIndex lineIntIndex = deserialize("loadExisting-receiving.xml");
  DataAccess mockDataAccess = mockField_dataAccess_InLineIntIndex(lineIntIndex);
  when(mockDataAccess.loadExisting()).thenReturn(true);
  when(mockDataAccess.getHeader(0)).thenReturn(5);
  when(mockDataAccess.getHeader(4)).thenReturn(1813699);
  when(mockDataAccess.getHeader(8)).thenReturn(300);

  // Act
  boolean actual = lineIntIndex.loadExisting();

  // Assert
  ...
}
.....................................................................................
@Test
public void testForloadExisting_OO {
  ...
  // Assert
  assertTrue(actual);
}
.....................................................................................
@Test
public void testForloadExisting_PO {
  ...
  // Assert
  verify(mockDataAccess, atLeastOnce()).loadExisting();
  verify(mockDataAccess, atLeastOnce()).getHeader(0);
  verify(mockDataAccess, atLeastOnce()).getHeader(4);
  verify(mockDataAccess, atLeastOnce()).getHeader(8);
}
.....................................................................................
@Test
public void testForloadExisting_CO {
  ...
  // Assert
  InOrder orderVerifier = inOrder(mockDataAccess);
  orderVerifier.verify(mockDataAccess, times(1)).loadExisting();
  orderVerifier.verify(mockDataAccess, times(3)).getHeader(anyInt());
}
```

Listing 4.3: **(RQ2)** RICK generates three tests for the target method `loadExisting` of GRAPHHOPPER, each verifying its observed behaviors through one mock-based oracle.

To what extent do mock-based oracles derived from runtime observations contribute to regression testing?

We propose RICK, which observes the invocations of target methods and their mockable method calls. Using the data captured from these observations in the field, RICK generates unit tests for each target method, which mock external objects, and configure their interactions through stubs. The generated tests express the observed behaviors through mock-based oracles, verifying the output from the target method (OO), the arguments that the mock methods should be invoked with (PO), as well as the order in which, and the number of times that, these mock method calls should occur (CO). The ability to generate unit tests with mocks, stubs, and mock-based oracles from runtime observations is novel and unique to RICK. Furthermore, through mutation analysis, we demonstrate that the generated mock-based oracles are good at detecting regressions, and complement each other in their ability to detect bugs.

4.3 RQ3: To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?

Following the protocol of Section 3.3, we use PROZE to capture the arguments of target methods that are directly invoked by developer-written tests. PROZE transforms these tests into parameterized unit tests (PUTs), using the captured arguments to generate argument providers for these PUTs.

Table 4.3 summarizes the results of our evaluation of PROZE on three modules of PDFBox (`fontbox`, `xmpbox`, and `pdfbox`), and two modules of SAML (`query` and `sso`). In total, PROZE identifies 128 target methods in these five modules, which are methods that are directly invoked by the 248 *conventional* unit tests (CUTs) in their developer-written test suites. As detailed in Section 3.3, PROZE isolates each assertion statement contained in these CUTs into its own PUT, which is parameterized over a single target method. Furthermore, each PUT is supplied arguments through a corresponding argument provider, which is generated using the union of arguments captured for the target method from its test and field invocations. In total, PROZE derives 2,287 PUTs from the 248 CUTs. However, we disregard 373 invalid PUTs that fail when supplied with the original arguments of the target method they are parameterized over. As we see from Table 4.3, this results in 1,914 PUTs that are eligible for classification as STRONGLY-COUPLED, FALSIFIABLY-COUPLED, or DECOUPLED. Running these 1,914 PUTs, we find that 686 are strongly-coupled, i.e., their oracle holds only when the target method is invoked with the original argument within the PUT. Next, for 217 PUTs, we discover a valid and invalid set of inputs that make the oracle pass and fail, respectively. These are the falsifiably-coupled cases, where PROZE has successfully

Table 4.3: Summary of results for RQ3: PROZE transforms developer-written conventional unit tests (CUTs) into parameterized unit tests (PUTs). The generated PUTs are classified as STRONGLY-COUPLED, DECOUPLED, or FALSIFIABLY-COUPLED, based on the generalizability of their oracle. PROZE finds 217 falsifiably-coupled oracles, for which it has captured a valid and invalid set of inputs at runtime.

MODULE	TARGET METHODS	CUTS	PUTS	STRONGLY -COUPLED	DECOUPLED	FALSIFIABLY -COUPLED
`fontbox`	12	15	135	31	92	12
`xmpbox`	27	36	543	94	409	40
`pdfbox`	55	155	1,150	510	479	161
`query`	2	3	7	0	7	0
`sso`	32	39	79	51	24	4
TOTAL	128	248	**1,914**	686	1,011	**217**

found generalizable developer-written assertions. Finally, the oracle within 1,011 generated PUTs are valid for all the inputs that are supplied by the argument provider. We refer to this class of PUTs as decoupled.

We illustrate with an end-to-end example from the `xmpbox` module of PDFBOX, whose test suite contains the developer-written CUT `testBagManagement` presented from lines 1 to 17 of Listing 4.5. This CUT calls the target method `createText` on line 8, with a set of four `String` arguments, `null, "rdf", "li", "valueOne"`. The CUT contains four assertions on lines 11, 12, 15 and 16. PROZE derives four PUTs from this CUT, each containing exactly one assertion statement from the CUT (line 27). Furthermore, each PUT is parameterized over `createText`, i.e., it accepts four `String` arguments (line 23) which are then used to invoke `createText` (line 25). The argument provider for these PUTs (lines 31-39) is generated using the 738 unique sets of arguments captured by PROZE for `createText`, while we run the test suite of `xmpbox`, as well as PDFBOX with a field workload. This union contains the original set of arguments (line 34) with which `createText` is invoked with, within `testBagManagement`. Note that each PUT is linked to the generated argument provider through the JUnit5 `@MethodSource` annotation (line 22). Running the generated PUTs, we find that two of them that contain the assertions from lines 11 and 15 pass only when supplied the original argument of `createText`. This makes these two assertions strongly-coupled to the invocation of the target method. Next, the PUT which contains the assertion on line 12 passes for all 738 arguments that are supplied by the provider. We therefore conclude that this oracle is decoupled from the invocation of `createText`. Finally, the PUT with the assertion on line 16 holds for the original set of arguments (line 34), as well as exactly one additional set,

```java
@Test
public void testBagManagement() {
  XMPMetadata parent = XMPMetadata.createXMPMetadata();
  XMPSchema schem = new XMPSchema(parent, "nsURI", "nsSchem");
  String bagName = "BAGTEST";
  String value1 = "valueOne";
  String value2 = "valueTwo";
  schem.addBagValue(bagName, schem.getMetadata().getTypeMapping().createText(null,
      "rdf", "li", value1));
  schem.addQualifiedBagValue(bagName, value2);
  List<String> values = schem.getUnqualifiedBagValueList(bagName);
  assertEquals(value1, values.get(0));
  assertEquals(value2, values.get(1));
  schem.removeUnqualifiedBagValue(bagName, value1);
  List<String> values2 = schem.getUnqualifiedBagValueList(bagName);
  assertEquals(1, values2.size());
  assertEquals(value2, values2.get(0));
}
.....................................................................................

// Generated PUT parameterized over createText(String, String, String, String)
@ParameterizedTest
@MethodSource("provideCreateTextArgs")
public void testBagManagement_PUT_N(String ns, String prefix, String propName,
    String value) {
  ...
  schem.addBagValue(bagName, schem.getMetadata().getTypeMapping().createText(ns,
      prefix, propName, value));
  ...
  // one of lines 11, 12, 15, or 16 per PUT
  ...
}

// Argument provider with 738 captured inputs
static Stream<Arguments> provideCreateTextArgs() {
  return java.util.stream.Stream.of(
    Arguments.of(null, "rdf", "li", "valueOne"), // original argument
    ...
    Arguments.of(null, "pdf", "PDFVersion", "1.4"),
    Arguments.of("nsURI", "nsSchem", "li", "valueTwo")
  );
}
```

Listing 4.5: **(RQ3)** The CUT `testBagManagement` in `xmpbox` has 4 assertions, and directly calls target method `createText`. PROZE captures 738 arguments for `createText` at runtime, and generates the argument provider method `provideCreateTextArgs`. This provider supplies arguments to 4 generated PUTs parameterized over `createText`. The parts common to the 4 generated PUTs are included in `testBagManagement_PUT_N`.

i.e., `"nsURI", "nsSchem", "li", "valueTwo"` (line 37). We classify this as a falsifiably-coupled case.

As we see from Table 4.3, PROZE successfully identifies 217 falsifiably-coupled developer-written oracles that are provably sensitive to valid and invalid test inputs captured at runtime, and that are generalizable to a larger input range. We envision that these PUTs can be integrated into the test suite after (i) limiting the argument provider to deliver only valid arguments, by removing the falsifying arguments from the initial provider; and (ii) if multiple PUTs use the same argument providers, merging the assertions of these different PUTs into one PUT.

To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?

PROZE is a novel approach for the transformation of developer-written unit tests into parameterized unit tests (PUTs). It captures the union of arguments passed to a target method at runtime to generate argument providers that supply values to each generated PUT, parameterized over a single target method and containing a single developer-written oracle. With PROZE, we demonstrate that developers indeed write oracles that generalize over a larger input range than the one they envision. PROZE successfully harnesses runtime observations to discover such oracles in existing test suites.

4.4 Conclusion about observation-based oracle generation

Each of our research questions highlights a distinct way of expressing observed runtime behaviors as test oracles. First, the explicit assertions generated by PANKTI (RQ1) guarantee that regressions, including those introduced by extreme mutations, do not go undetected by the test suite. The generated tests therefore improve the test quality of the target project based on the defined test adequacy criterion of mutation analysis. Second, the mock-based oracles generated by RICK (RQ2) demonstrate that different aspects of field behavior can be expressed using the configured machinery of mocks and stubs. These mock-based oracles are capable of detecting regressions that are introduced in the system as it evolves. Third, with PROZE (RQ3) we discover existing developer-written oracles that are valid over a larger range of inputs captured at runtime, allowing the test suite to be strengthened through parameterized unit tests. Runtime observations can indeed contribute to the generalization of developer-written oracles. These findings are novel to the best of our knowledge. They address the diverse ways in which runtime observations can be incorporated into the test suite to augment it, such that the nominal operative behaviors of the target system are better represented. Our approach can be applied to software projects regardless of the maturity of their current testing practices. The oracles produced using runtime observations act not only as feedback from real behaviors into the test suite, but

also strengthen the test suite by facilitating the detection of regressions as the system inevitably evolves with time.

Per the survey by Bertolino *et al.* [21] on field-based testing strategies, we position our contributions as ex-vivo techniques that use data from the field to generate tests in the development environment. The generated tests are intended to complement the developer-written test suite, and be run in-house. Furthermore, with respect to the literature on automated generation of oracles summarized in Section 2.1, our work is classified under behavior-based oracle generation strategies. The novelty of our work lies in the utilization of real-world behaviors for the generation of regression oracles (RQ1, RQ2) and mock-based oracles (RQ2), as well as for the generalization of existing developer-written oracles (RQ3). Through our contributions, we propose a solution to the oracle problem [15] by observing systems in the field to determine their expected behaviors. Moreover, we generate complete tests by instantiating our framework of Section 3.1. Relative to the observation-based test generation strategies discussed in Section 2.2, each test we generate includes both of the components essential to a software test, its inputs and its oracle, derived from runtime behaviors.

We note here that, as with related work on observation-based test generation [21,143], our approach relies on monitoring and serialization at runtime. There are performance implications of this test generation strategy that must be considered. With PANKTI, RICK, and PROZE, we instrument target methods to serialize data related to their invocations. For PROZE, our evaluation focuses on target methods with primitive or `String` arguments, for which the serialization cost is minimal. However, with PANKTI and RICK, that also serialize non-primitive objects, we employ strategies to limit the cost incurred from monitoring the invocations of target methods. First, we define criteria to select target methods that are useful for test generation, such as targeting pseudo-tested methods with PANKTI, or methods with mockable method calls with RICK. Second, during the execution of the test suite, or the project in the field, a target method may be invoked thousands of times. We limit the number of invocations, and the size of the captured data on disk, to sample a subset of these invocations. We discuss the performance implications of PANKTI and RICK in Papers I and IV, respectively.

5. Conclusion

"All these sights and sounds and smells will be yours to enjoy, Wilbur — this lovely world, these precious days..."

E.B. White, Charlotte's Web

The process of software testing is key to ensuring the quality of all the diverse software systems that we have come to depend on. In this thesis, we have provided an overview of the vibrant ecosystem of software testing techniques, especially focusing on strategies for the automated generation of software tests. Our own contributions in this domain are motivated by the observation that, despite significant developer effort, it is seldom possible to account for, and test, all the scenarios that a software system might encounter in the field [21]. In fact, the oracles contained within developer-written tests might not reflect behaviors that are exhibited by a system in production, when it is exercised with real workloads by end-users [19]. This gap between tested behaviors and field behaviors is not addressed explicitly by the techniques proposed in the literature on automated software test generation.

5.1 Summary

We provide evidence that runtime observations can be harnessed for the generation of test inputs and test oracles. In order to do so, we propose a novel conceptual framework for observing a system as it runs in the field, capturing runtime behaviors in the form of data, and using this data to generate tests. The generated tests, including test inputs and oracles, are representative of the observed field behaviors, and are intended to complement developers in their testing activities. Furthermore, we provide three distinct instances of our conceptual framework, targeting diverse ways of utilizing runtime observations for the augmentation of test oracles.

First, with PANKTI, we target methods that are inadequately tested by the developer-written test suite of the system under test. PANKTI observes the invocations of these methods in the field, and captures the objects associated with the invocations. It then uses these captured objects to generate unit tests that

recreate the observed production state. Through our evaluation with pseudo-tested methods within three notable Java projects, we demonstrate that over 60% of them are no longer pseudo-tested as a consequence of the explicit oracles contained within the tests generated by PANKTI.

Second, we propose a mechanism called RICK that captures objects related to the invocations of target methods, as well as the methods within them. These objects are used by RICK to generate unit tests with mocks, stubs, and three kinds of mock-based oracles. The output oracle verifies that the output from the target method within the test is the same as the output observed in the field. The parameter oracle verifies the arguments with which nested method calls occur within the invocation of the target method, and the call oracle verifies the expected sequence and frequency of these nested method calls. These three mock-based oracles also detect regressions introduced within the system. A majority of the tests generated by RICK for three large Java projects successfully reproduce the entirety of the observed field behaviors.

Third, PROZE targets methods that are directly invoked by developer-written unit tests. It captures the arguments with which these target methods are invoked in the field and during the execution of the test suite, and generates data providers with these runtime arguments. PROZE derives parameterized unit tests from the existing unit tests, supplying them arguments through the generated data providers. From our evaluation of PROZE against five Java modules from two open-source projects, we discover over 200 existing developer-written oracles that are provably generalizable. This means that, for these oracles, we find both valid and invalid inputs, and demonstrate that they indeed hold over a larger set of input values than expressed in the original unit tests authored by developers.

Overall, these results provide strong and sound support to our core thesis:

> **Runtime observations, made as a system executes in the field, can be utilized to generate complete tests, with inputs and oracles. The generated tests augment the developer-written test suite, serving as regression tests that are representative of real-world usage scenarios.**

5.2 Future work

We are hopeful that our findings will encourage software developers to incorporate observation-based testing in their practice. Additionally, we are optimistic about the implications of our contributions within software engineering research. In particular, we propose three directions that further our work on harnessing runtime observations for augmenting software quality.

Deriving general oracles from runtime observations: Per our methodology, the oracles we generate are fine-tuned to the exact production state recreated within the generated tests. However, as we demonstrate with PROZE in Paper

III, it is desirable to discover oracles that are less tightly coupled with specific input scenarios. Such oracles can express more general behaviors of the system, and can be likened to invariants [57] or properties [46] that generalize to a large input space. We propose that it is possible to use runtime observations to derive such general oracles that are valid for a large set of representative field inputs.

General oracles from production behaviors would require the curation of large dataset of captured field behaviors for target units, on which statistical or learning-based approaches can be applied. For example, invariants may be deduced on the basis of a range of inputs and outputs that are valid for a target method. Additionally, the range of inputs that result in equivalent outputs, or the states of internal variables [42] can serve as invariants. Metamorphic relations, which refer to the change in the state of the output with respect to the change in the input, can also be derived by observing changes in the states of production objects.

The oracles discovered from runtime monitoring can be utilized within generated tests. They may also be embedded in production code, to dynamically ensure correctness and safeguard against deviations from expected behaviors [166]. This would help cultivate a sense-of-self [167] within the system with respect to its nominal operational behaviors.

Deriving oracles by leveraging usage from client projects: In this thesis, we evaluate our approach against executable projects for which we design workloads that resemble usage scenarios by human users. However, most software projects are actually intended to be used as *libraries* or dependencies by a large number of diverse *client* projects [168].

Well-known library projects often have strong developer-written test suites. However, their tests suites might not reflect all the ways in which the libraries are exercised by client projects in the wild. For such libraries, the field workload can be considered as their usage by client projects, either within their developer-written test suite, or in the field. There are studies that investigate the advantages of adopting client usage information to strengthen the test quality of libraries [169, 170].

We envision that our proposed framework can be adapted to generate tests for library projects based on its usage by diverse clients. Moreover, an interesting application of this technique would be in the context of domain-specific libraries. For example, multiple JSON serialization libraries exist for all programming languages, each following the JSON specification [152]. Oracles can be automatically produced by evaluating the usage of JSON libraries across client projects and programming languages. This empirical analysis will shed light on the behavioral diversity among libraries that implement the same specification. Additionally, we will gain insight into the distinct ways in which their APIs are used within client projects. The oracles generated from this technique may also be transplanted [171] across the libraries.

Runtime data, faking, and generative AI for test inputs: The inputs and oracles contained in the tests generated in this thesis are sourced purely from

runtime observations. Meanwhile, through our investigation of large open-source test suites, we find that developers rely on data faking as a useful way of supplying realistic inputs within automated tests. Additionally, generative AI models have recently found application in almost all software development tasks. An initial investigation into the use of data fakers with generative AI looks promising [172].

We hypothesize that using runtime data in conjunction with data faking and generative AI can be fruitful for producing test inputs that are representative of the application domain. One way to achieve this would be to fine-tune a generative model on a sample of captured runtime data. The model can then be used to produce an in-house data faker that is capable of supplying interesting and realistic test inputs to tests that are manually created or automatically generated. Such in-house data fakers may be developed from scratch, or be fine-tuned forks of open-source faking libraries. In both cases, we believe that our strategy would help developers use representative and domain-specific test input data within their test suites. This technique would also be valuable across diverse natural and programming languages.

References

[1] B. Baudry and M. Monperrus, "Exhaustive Survey of Rickrolling in Academic Literature," in *Proceedings of SIGBOVIK'22*, 2022.

[2] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and J. C. Carver, "Software Engineering Practices for Scientific Software Development: A Systematic Mapping Study," *Journal of Systems and Software*, vol. 172, p. 110848, 2021.

[3] C. Soto-Valero, M. Monperrus, and B. Baudry, "The Multibillion Dollar Software Supply Chain of Ethereum," *Computer*, vol. 55, no. 10, pp. 26–34, 2022.

[4] B. Baudry and M. Monperrus, "Programming Art With Drawing Machines," *Computer*, vol. 57, no. 07, pp. 104–108, 2024.

[5] W. E. Wong, X. Li, and P. A. Laplante, "Be More Familiar with Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures," *Journal of Systems and Software*, vol. 133, pp. 68–94, 2017.

[6] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, Tests, and Oracles: The Foundations of Testing Revisited," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 391–400, 2011.

[7] M. Aniche, C. Treude, and A. Zaidman, "How Developers Engineer Test Cases: An Observational Study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2021.

[8] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[9] L. Gazzola, L. Mariani, F. Pastore, and M. Pezze, "An Exploratory Study of Field Failures," in *2017 IEEE 28th International Symposium on Software Reliability Engineering*, pp. 67–77, IEEE, 2017.

[10] M. Pezze and C. Zhang, "Automated Test Oracles: A Survey," in *Advances in Computers*, vol. 95, pp. 1–48, Elsevier, 2014.

[11] G. Jahangirova, *Oracle Assessment, Improvement and Placement*. PhD thesis, University College London, 2019.

[12] J. Leveau, X. Blanc, L. Réveillère, J.-R. Falleri, and R. Rouvoy, "Fostering the Diversity of Exploratory Testing in Web Applications," *Software Testing, Verification and Reliability*, vol. 32, no. 5, p. e1827, 2022.

[13] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, *et al.*, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[14] U. Kanewala and J. M. Bieman, "Testing Scientific Software: A Systematic Literature Review," *Information and Software Technology*, vol. 56, no. 10, pp. 1219–1232, 2014.

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[16] W. Sun, Z. Guo, M. Yan, Z. Liu, Y. Lei, and H. Zhang, "Method-Level Test-to-Code Traceability Link Construction by Semantic Correlation Learning," *IEEE Transactions on Software Engineering*, 2024.

[17] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Assert Statements for Unit Test Cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1398–1409, 2020.

[18] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 201–211, IEEE, 2015.

[19] Q. Wang, Y. Brun, and A. Orso, "Behavioral Execution Comparison: Are Tests Representative of Field Behavior?," in *2017 IEEE International Conference on Software Testing, Verification and Validation*, pp. 321–332, IEEE, 2017.

[20] Q. Wang and A. Orso, "Mimicking User Behavior to Improve In-House Test Suites," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 318–319, IEEE, 2019.

[21] A. Bertolino, P. Braione, G. D. Angelis, L. Gazzola, F. Kifetew, L. Mariani, M. Orrù, M. Pezze, R. Pietrantuono, S. Russo, *et al.*, "A Survey of Field-Based Testing Techniques," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–39, 2021.

[22] N. Alshahwan, M. Harman, and A. Marginean, "Software Testing Research Challenges: An Industrial Perspective," in *2023 IEEE Conference on Software Testing, Verification and Validation*, pp. 1–10, IEEE, 2023.

[23] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 4, pp. 1–49, 2015.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *29th International Conference on Software Engineering*, pp. 75–84, IEEE, 2007.

[25] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM Sigsoft Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[26] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[27] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, 2024.

[28] N. Li and J. Offutt, "Test Oracle Strategies for Model-based Testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2016.

[29] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-Checking Oracles from Intrinsic Software Redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 931–942, 2014.

[30] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic Generation of Oracles for Exceptional Behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 213–224, 2016.

[31] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating Code Comments to Procedure Specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 242–253, 2018.

[32] M. Motwani and Y. Brun, "Automatically Generating Precise Oracles from Structured Natural Language Specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering*, pp. 188–199, IEEE, 2019.

[33] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[34] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2008.

[35] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated Oracle Data Set Selection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 193–203, 2014.

[36] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "Automated Oracle Data Selection Support," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1119–1137, 2015.

[37] C. Xu, V. Terragni, H. Zhu, J. Wu, and S.-C. Cheung, "MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases," *ACM Transactions on Software Engineering and Methodology*, 2024.

[38] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the Crowd Solve the Oracle Problem?," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 342–351, IEEE, 2013.

[39] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, IEEE, 2017.

[40] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A Comprehensive Survey of Trends in Oracles for Software Testing," *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[41] N. Li and J. Offutt, "An Empirical Analysis of Test Oracle Strategies for Model-Based Testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 363–372, IEEE, 2014.

[42] Y. Xiong, D. Hao, L. Zhang, T. Zhu, M. Zhu, and T. Lan, "Inner Oracles: Input-specific Assertions on Internal States," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 902–905, 2015.

[43] M. Nassif, A. Hernandez, A. Sridharan, and M. P. Robillard, "Generating Unit Tests for Documentation," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3268–3279, 2021.

[44] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 815–816, 2007.

[45] D. Saff, M. Boshernitsan, and M. D. Ernst, "Theories in Practice: Easy-To-Write Specifications That Catch Bugs," tech. rep., MIT Computer Science and Artificial Intelligence Laboratory, 2008.

[46] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 268–279, 2000.

[47] D. R. MacIver, Z. Hatfield-Dodds, *et al.*, "Hypothesis: A New Approach to Property-Based Testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[48] C. Wauters and C. De Roover, "Property-Based Testing Within ML Projects: An Empirical Study," in *International Conference on Software Maintenance and Evolution*, IEEE, 2024.

[49] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-Based Testing in Practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[50] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable Modified Condition/Decision Coverage," in *2013 35th International Conference on Software Engineering*, pp. 102–111, IEEE, 2013.

[51] T. Xie, "Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking," in *European Conference on Object-Oriented Programming*, pp. 380–403, Springer, 2006.

[52] M. Staats, G. Gay, and M. P. Heimdahl, "Automated Oracle Creation Support, Or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing," in *2012 34th International Conference on Software Engineering*, pp. 870–880, IEEE, 2012.

[53] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2013.

[54] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A Survey on Metamorphic Testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[55] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic Testing: A Review of Challenges and Opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[56] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "MeMo: Automatically Identifying Metamorphic Relations in Javadoc Comments for Test Automation," *Journal of Systems and Software*, vol. 181, p. 111041, 2021.

[57] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[58] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding User Understanding: Determining Correctness of Generated Program Invariants," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 188–198, 2012.

[59] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic Symbolic Execution for Invariant Inference," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 281–290, 2008.

[60] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 163–174, 2022.

[61] F. Molina, "Applying Learning Techniques to Oracle Synthesis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1153–1157, 2020.

[62] F. Molina, P. Ponzio, N. Aguirre, and M. Frias, "EvoSpex: An Evolutionary Algorithm for Learning Postconditions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 1223–1235, IEEE, 2021.

[63] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A Neural Method for Test Oracle Generation," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2130–2141, 2022.

[64] G. Fraser and A. Arcuri, "Evolutionary Generation of Whole Test Suites," in *2011 11th International Conference on Quality Software*, pp. 31–40, IEEE, 2011.

[65] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, "Neural-based Test Oracle Generation: A Large-scale Evaluation and Lessons Learned," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 120–132, 2023.

[66] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards More Realistic Evaluation for Neural Test Oracle Generation," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 589–600, 2023.

[67] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing Evaluation Metrics for Neural Test Oracle Generation," *IEEE Transactions on Software Engineering*, 2024.

[68] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the Enemy of Test Oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 70–81, 2022.

[69] F. Molina and A. Gorla, "Test Oracle Automation in the Era of LLMs," *arXiv preprint arXiv:2405.12766*, 2024.

[70] S. B. Hossain and M. Dwyer, "TOGLL: Correct and Strong Test Oracle Generation with LLMs," *arXiv preprint arXiv:2405.03786*, 2024.

[71] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A Snowballing Literature Study on Test Amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[72] M. Staats, P. Loyola, and G. Rothermel, "Oracle-Centric Test Case Prioritization," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 311–320, IEEE, 2012.

[73] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 90–99, IEEE, 2011.

[74] D. Schuler and A. Zeller, "Checked Coverage: An Indicator for Oracle Quality," *Software Testing, Verification and Reliability*, vol. 23, no. 7, pp. 531–551, 2013.

[75] G. Fraser and A. Zeller, "Mutation-driven Generation of Unit Tests and Oracles," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pp. 147–158, 2010.

[76] R. Niedermayr, E. Juergens, and S. Wagner, "Will My Tests Tell Me if I Break This Code?," in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pp. 23–29, 2016.

[77] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A Comprehensive Study of Pseudo-Tested Methods," *Empirical Software Engineering*, vol. 24, pp. 1195–1225, 2019.

[78] C. Huo and J. Clause, "Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 621–631, 2014.

[79] X. Guo, M. Zhou, X. Song, M. Gu, and J. Sun, "First, Debug the Test Oracle," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 986–1000, 2015.

[80] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test Oracle Assessment and Improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 247–258, 2016.

[81] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "OASIs: Oracle Assessment and Improvement Tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 368–371, 2018.

[82] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An Empirical Validation of Oracle Improvement," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1708–1728, 2019.

[83] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary Improvement of Assertion Oracles," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1178–1189, 2020.

[84] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "GAssert: A Fully Automated Tool to Improve Assertion Oracles," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 85–88, IEEE, 2021.

[85] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, "AGORA: Automated Generation of Test Oracles for REST APIs," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1018–1030, 2023.

[86] S. Tuli, K. Bojarczuk, N. Gucevska, M. Harman, X.-Y. Wang, and G. Wright, "Simulation-Driven Automated End-to-End Test and Oracle Inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 122–133, IEEE, 2023.

[87] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 280–290, 2018.

[88] A. E. Genç, H. Sözer, M. F. Kıraç, and B. Aktemur, "ADVISOR: An Adjustable Framework for Test Oracle Automation of Visual Output Systems," *IEEE Transactions on Reliability*, vol. 69, no. 3, pp. 1050–1063, 2019.

[89] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, "On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, 2014.

[90] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 183–192, IEEE, 2014.

[91] K. Baral, J. Johnson, J. Mahmud, S. Salma, M. Fazzini, J. Rubin, J. Offutt, and K. Moran, "Automating GUI-based Test Oracles for Mobile Apps," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 309–321, 2024.

[92] R. Jabbarvand, F. Mehralian, and S. Malek, "Automated Construction of Energy Test Oracles for Android," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 927–938, 2020.

[93] G. Gay, S. Rayadurgam, and M. P. Heimdahl, "Automated Steering of Model-based Test Oracles to Admit Real Program Behaviors," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 531–555, 2016.

[94] A. Afzal, C. Le Goues, and C. S. Timperley, "Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4535–4552, 2021.

[95] A. Arrieta, M. Otaegi, L. Han, G. Sagardui, S. Ali, and M. Arratibel, "Automating Test Oracle Generation in DevOps for Industrial Elevators," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 284–288, IEEE, 2022.

[96] A. Gartziandia, A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Machine Learning-based Test oracles for Performance Testing of Cyber-physical Systems: An Industrial Case Study on Elevators Dispatching Algorithms," *Journal of Software: Evolution and Process*, vol. 34, no. 11, p. e2465, 2022.

[97] G. Jahangirova, A. Stocco, and P. Tonella, "Quality Metrics and Oracles for Autonomous Vehicles Testing," in *2021 14th IEEE International Conference on Software Testing, Verification and Validation*, pp. 194–204, IEEE, 2021.

[98] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[99] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pp. 419–423, Springer, 2006.

[100] N. Tillmann and J. De Halleux, "Pex–White Box Test Generation for .NET," in *International Conference on Tests and Proofs*, pp. 134–153, Springer, 2008.

[101] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering*, pp. 193–202, 2009.

[102] G. Fraser and A. Arcuri, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, 2014.

[103] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic Generation of System Test Cases from Use Case Specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 385–396, 2015.

[104] S. Liu and S. Nakajima, "Automatic Test Case and Test Oracle Generation Based on Functional Scenarios in Formal Specifications for Conformance Testing," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 691–712, 2020.

[105] M. Pezze, P. Rondena, and D. Zuddas, "Automatic GUI Testing of Desktop Applications: An Empirical Assessment of the State of the Art," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pp. 54–62, 2018.

[106] T. Su, J. Wang, and Z. Su, "Benchmarking Automated GUI Testing for Android Against Real-World Bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 119–130, 2021.

[107] S. Balsam and D. Mishra, "Web Application Testing—Challenges and Opportunities," *Journal of Systems and Software*, p. 112186, 2024.

[108] R. K. Yandrapally and A. Mesbah, "Fragment-based Test Generation for Web Apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1086–1101, 2022.

[109] X. Yu, L. Liu, X. Hu, J. Keung, X. Xia, and D. Lo, "Practitioners' Expectations on Automated Test Generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1618–1630, 2024.

[110] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Precise Identification of Problems for Structural Test Generation," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 611–620, 2011.

[111] N. Alshahwan, A. Blasi, K. Bojarczuk, A. Ciancone, N. Gucevska, M. Harman, M. Krolikowski, R. Rojas, D. Martac, S. Schellaert, *et al.*, "Enhancing Testing at Meta with Rich-State Simulated Populations," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 1–12, 2024.

[112] G. Fraser and A. Zeller, "Exploiting Common Object Usage in Test Case Generation," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 80–89, IEEE, 2011.

[113] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities," in *2020 IEEE International Conference on Software Maintenance and Evolution*, pp. 523–533, IEEE, 2020.

[114] A. Deljouyi and A. Zaidman, "Generating Understandable Unit Tests Through End-To-End Test Scenario Carving," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation*, pp. 107–118, IEEE, 2023.

[115] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/Deep Learning for Software Engineering: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1188–1231, 2022.

[116] L. Chemnitz, D. Reichenbach, H. Aldebes, M. Naveed, K. Narasimhan, and M. Mezini, "Towards Code Generation from BDD Test Case Specifications: A Vision," in *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI*, pp. 139–144, IEEE, 2023.

[117] E. Jabbar, H. Hemmati, and R. Feldt, "Investigating Execution Trace Embedding for Test Case Prioritization," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security*, pp. 279–290, IEEE, 2023.

[118] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimarães, "Machine Learning Applied to Software Testing: A Systematic Mapping Study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019.

[119] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit Test Case Generation with Transformers and Focal Context," *arXiv preprint arXiv:2009.05617*, 2020.

[120] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[121] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3Test: Assertion-Augmented Automated Test Case Generation," *Information and Software Technology*, p. 107565, 2024.

[122] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," *IEEE Transactions on Software Engineering*, 2023.

[123] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot's Code Suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 1–5, 2022.

[124] V. Vikram, C. Lemieux, and R. Padhye, "Can Large Language Models Write Good Property-based Tests?," *arXiv preprint arXiv:2307.04346*, 2023.

[125] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "CAT-LM Training Language Models on Aligned Code and Tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, pp. 409–420, IEEE, 2023.

[126] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 572–576, 2024.

[127] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards Autonomous Testing Agents Via Conversational Large Language Models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1688–1693, IEEE, 2023.

[128] F. G. de Oliveira Neto, "Unveiling Assumptions: Exploring the Decisions of AI Chatbots and Human Testers," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 45–49, 2024.

[129] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated Unit Test Improvement Using Large Language Models at Meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 185–196, 2024.

[130] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large Language Models for Software Engineering: Survey and Open Problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp. 31–53, IEEE, 2023.

[131] R. Santos, I. Santos, C. Magalhaes, and R. de Souza Santos, "Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing," in *2024 IEEE Conference on Software Testing, Verification and Validation*, pp. 353–360, IEEE, 2024.

[132] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning Runtime Behavior of a Program with LLM: How Far Are We?," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 140–152, IEEE Computer Society, 2024.

[133] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay Tool for Observation-Based Testing," in *Proceedings of the 2000 ACM Sigsoft International Symposium on Software Testing and Analysis*, pp. 158–167, 2000.

[134] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: A Unit Test and Mock Aspect Generation Tool," *International Journal on Software Tools for Technology Transfer*, vol. 11, pp. 273–290, 2009.

[135] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving Differential Unit Test Cases From System Test Cases," in *Proceedings of the 14th ACM Sigsoft International Symposium on Foundations of Software Engineering*, pp. 253–264, 2006.

[136] A. Kampmann and A. Zeller, "Carving Parameterized Unit Tests," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 248–249, IEEE, 2019.

[137] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object Capture-Based Automated Testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pp. 159–170, 2010.

[138] D. Qu, C. Zhao, Y. Jiang, and C. Xu, "Towards Life-Long Software Self-Validation in Production," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pp. 357–366, 2024.

[139] L. Zhang, B. Morin, P. Haller, B. Baudry, and M. Monperrus, "A Chaos Engineering System for Live Analysis and Falsification of Exception-Handling in the JVM," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2534–2548, 2019.

[140] M. Ceccato, D. Corradini, L. Gazzola, F. M. Kifetew, L. Mariani, M. Orru, and P. Tonella, "A Framework for In-Vivo Testing of Mobile Applications," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, pp. 286–296, IEEE, 2020.

[141] L. Gazzola, L. Mariani, M. Orrú, M. Pezze, and M. Tappler, "Testing Software in Production Environments with Data From the Field," in *2022 IEEE Conference on Software Testing, Verification and Validation*, pp. 58–69, IEEE, 2022.

[142] A. Bertolino, G. De Angelis, B. Miranda, and P. Tonella, "In Vivo Test and Rollback of Java Applications as They Are," *Software Testing, Verification and Reliability*, vol. 33, no. 7, p. e1857, 2023.

[143] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based Unit Test Generation at Meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 173–184, 2024.

[144] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-Based Slicing for System-Level Test Cases," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 517–528, 2021.

[145] B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry, "An approach and benchmark to detect behavioral changes of commits in continuous integration," *Empirical Software Engineering*, vol. 25, pp. 2379–2415, 2020.

[146] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing Non-Adequate Test Suites Using Coverage Criteria," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 302–313, 2013.

[147] M. Hilton, J. Bell, and D. Marinov, "A Large-Scale Study of Test Coverage Evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 53–63, 2018.

[148] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical Mutation Testing at Scale: A View from Google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2021.

[149] K. Jain, G. T. Kalburgi, C. Le Goues, and A. Groce, "Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts," in *2023 IEEE 34th International Symposium on Software Reliability Engineering*, pp. 102–113, 2023.

[150] Q. Wang and A. Orso, "Improving Testing by Mimicking User Behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution*, pp. 488–498, IEEE, 2020.

[151] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *2012 Second International Conference on Digital Information and Communication Technology and its Applications*, pp. 177–182, IEEE, 2012.

[152] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "The Behavioral Diversity of Java JSON Libraries," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering*, pp. 412–422, IEEE, 2021.

[153] N. Tillmann and W. Schulte, "Parameterized Unit Tests," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 253–262, 2005.

[154] C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically Tagging the "AAA" Pattern in Unit Test Cases Using Machine Learning Models," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–3, 2022.

[155] T. Mackinnon, S. Freeman, and P. Craig, "Endo-Testing: Unit Testing with Mock Objects," *Extreme Programming Examined*, pp. 287–301, 2000.

[156] D. Thomas and A. Hunt, "Mock Objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[157] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To Mock or Not to Mock? An Empirical Study on Mocking Practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pp. 402–412, IEEE, 2017.

[158] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "StubCoder: Automated Generation and Repair of Stub Code for Mock Objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

[159] H. Du, V. K. Palepu, and J. A. Jones, "Ripples of a Mutation—An Empirical Study of Propagation Effects in Mutation Testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[160] G. Fraser and A. Zeller, "Generating Parameterized Unit Tests," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 364–374, 2011.

[161] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, pp. 344–348, IEEE, 2019.

[162] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing Science to Digital Forensics with Standardized Forensic Corpora," *Digital Investigation*, vol. 6, pp. S2–S11, 2009.

[163] S. Mostafa and X. Wang, "An Empirical Study on the Usage of Mocking Frameworks in Software Testing," in *2014 14th International Conference on Quality Software*, pp. 127–132, IEEE, 2014.

[164] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock Objects for Testing Java Systems: Why and How Developers Use Them, and How They Evolve," *Empirical Software Engineering*, vol. 24, pp. 1461–1498, 2019.

[165] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*, pp. 148–163, Springer, 2017.

[166] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus, "SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java," *arXiv preprint arXiv:2407.00246*, 2024.

[167] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp. 120–128, IEEE, 1996.

[168] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies," *IEEE Transactions on Software Engineering*, 2023.

[169] I. Schittekat, M. Abdi, and S. Demeyer, "Can We Increase the Test-Coverage in Libraries using Dependent Projects' Test-Suites?," in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pp. 294–298, 2022.

[170] M. Abdi and S. Demeyer, "Test Transplantation Through Dynamic Test Slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation*, pp. 35–39, IEEE, 2022.

[171] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated Software Transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 257–269, 2015.

[172] B. Baudry, K. Etemadi, S. Fang, Y. Gamage, Y. Liu, Y. Liu, M. Monperrus, J. Ron, A. Silva, and D. Tiwari, "Generative AI to Generate Test Data Generators," *IEEE Software*, 2024.

Part II

Included Papers

Production Monitoring to Improve Test Suites

Deepika Tiwari, Long Zhang, Martin Monperrus, and Benoit Baudry

KTH Royal Institute of Technology, Sweden

***Abstract*—In this paper, we propose to use production executions to improve the quality of testing for certain methods of interest for developers. These methods can be methods that are not covered by the existing test suite, or methods that are poorly tested. We devise an approach called PANKTI which monitors applications as they execute in production, and then automatically generates differential unit tests, as well as derived oracles, from the collected data. PANKTI's monitoring and generation focuses on one single programming language, Java. We evaluate it on three real-world, open-source projects: a videoconferencing system, a PDF manipulation library, and an e-commerce application. We show that PANKTI is able to generate differential unit tests by monitoring target methods in production, and that the generated tests improve the quality of the test suite of the application under consideration.**

***Index Terms*—Production monitoring, test improvement, test quality, test generation, test oracle**

I. Introduction

SOFTWARE developers write unit tests to assess programs with respect to an expected behavior captured in the form of a test oracle. Yet, the development of strong test suites is challenging: the selection of test data from complex input spaces is hard [1], the specification of good test oracles is costly [2], [3], and the assessment of test quality is time-consuming [4], [5]. Test improvement has recently emerged as one way to assist developers in these tasks [6]. The key idea is to combine developer-written tests with automatic analysis and generation techniques to complement these developer-written tests. For example, one can consolidate test oracles with additional method calls [7], or one can augment test inputs through operational abstraction [8]. Our intuition is that we can also use production data to cope with the weaknesses of developer-written tests.

In this paper, we introduce a novel approach called PANKTI for improving test suites with observations collected in production. PANKTI monitors an application in production, to collect execution data before and after certain methods of interest are executed, called target methods in this paper. PANKTI subsequently extracts test input data and derived oracles [2] from the collected data, and finally generates new test cases. The motivation of using production data is that production workloads may exercise behaviors not observed during the test execution [9]. PANKTI targets new test cases towards specific methods that are selected by the developers, according to what they need. For example, developers can decide to target methods that are weakly-tested, as determined by code coverage [10], [11] or mutation testing [12], [13].

PANKTI works as follows, it first instruments the application to monitor the execution of target methods. The application is then deployed in production and executed according to some workload. For each invocation of a target method, the receiving object, the objects passed as arguments, as well as the returned object are serialized and stored. Once production monitoring data has been collected and persisted, PANKTI deserializes the collected production objects to extract test inputs and synthesize derived oracles [2] that capture the behavior observed in production. The last step consists in assembling the input and the oracle in a new differential unit test [14] for the target method, that is, a test that is able to capture behavioral change with respect to a reference version. The essence of PANKTI is to recreate, in the generated differential unit test, the behavior observed in production. The final step in the PANKTI test generation process consists in running the new tests to determine if indeed the overall test quality improves.

We implement PANKTI for Java and evaluate it on three real-world, open-source, complex software systems: a videoconferencing system called Jitsi, a PDF manipulation library called PDFBox, and an e-commerce application called Broadleaf. These projects have between 28K and 729K lines of code and at least one thousand stars on GitHub. For evaluation purposes, we select as targets the methods that are found weakly-tested with respect to extreme mutation analysis [15]. For these experiments, PANKTI targets a total of 86 weakly-tested methods, and observes 122,194 invocations of these methods in production. With the objects collected from these invocations, PANKTI generates 14,222 differential unit tests, of which 13,878 (97.6%) tests pass. Thanks to these PANKTI-generated tests, the test quality of 53 of 86 (61.6%) target methods is found to have improved per the considered test adequacy criterion. These results show that PANKTI is able to automatically transform data observed in production into differential unit tests that improve the quality of a test suite. To sum up, our contributions are:

- PANKTI, a tool that monitors programs in production and uses the observed data to automatically generate differential unit tests that target weakly-tested methods;
- An evaluation of the tool on three notable, real-world, open-source Java projects run in production: a videoconferencing system, a PDF manipulation library, and an e-commerce system;
- A publicly available open-source implementation for Java[1], and open science experimental data[2] for reproducibility and extension by further research.

[1] https://github.com/castor-software/pankti
[2] https://doi.org/10.5281/zenodo.4298604

In section II, we describe our approach for test generation by monitoring programs in production. section III describes our evaluation methodology, section IV describes experimental results, section V discusses key insights and challenges from these results, section VI discusses related work, and section VII concludes the paper.

II. The Pankti Approach to Test Generation

This section describes the details of our technical contribution. Our tool, called PANKTI, exploits observations in production in order to generate new tests that complement the test suite of a Java application.

A. Overview

The key concepts to describe PANKTI are defined below.

Test improvement: Test improvement is the process of strengthening an existing manually-written test suite, to enhance a specific, measurable test adequacy criterion [16], [17]. This is the main goal of PANKTI.

Target methods: PANKTI focuses on specific methods, for which it will generate new tests. Developers are responsible for selecting these methods based on their test adequacy criterion of choice. For example, PANKTI can target methods in the code that are not fully covered by the test suite. We discuss target method selection in subsection II-B.

Production workload: In our study, a production workload is a sequence of inputs and interactions given to the program in production over a certain period [18]. For instance, a production workload of a PDF reader is to open and scroll over a PDF file.

Differential unit test: PANKTI is designed to generate unit tests that are small, clear, and capture a targeted test intention. The PANKTI unit tests all include an oracle that is based on the behavior observed in production. This production-based oracle can be considered as a derived test oracle per the classification of Barr *et al.* [2]. These unit tests aim to detect behavioral changes between the version monitored for test generation and upcoming revisions of the program, and are thus called "differential unit tests" by Elbaum *et al.* [14]. To sum up, PANKTI generates differential unit tests with a derived oracle based on production data.

Production object profile: When PANKTI monitors the execution of an application under consideration with a specific workload, it collects one production object profile each time one of the target methods is invoked. We define an object profile as a quadruplet $< method, receiving, parameters, result >$ as follows: given a target $method$ invocation, we monitor the $receiving$ object that the method is invoked on, with the objects or values passed as $parameters$ to the method, if any, as well as the $result$ object or value returned from the method.

Figure 1 gives an overview of the PANKTI pipeline for test generation. Our approach operates over three inputs: the source code of an application; a set of methods for which developers wish to get new tests; and one production workload. The output of PANKTI is a collection of differential unit tests for the target methods that have been invoked at least once while executing the application.

The workflow of PANKTI is structured in three phases. First, PANKTI instruments each target method with binary instrumentation, to collect runtime data about its execution context. Second, PANKTI executes the application with the input workload and collects runtime object profiles for each invocation of the target methods. Third, PANKTI generates differential unit tests for the target methods. When the target methods are selected based on a test adequacy criterion, PANKTI selects the generated tests that improve the test quality for the method, according to that criterion. The key concepts of PANKTI are illustrated in the next section, and the different steps of our pipeline are presented in detail in the following sections.

B. Selection of Target Methods

PANKTI performs generation of differential unit tests for a subset of the methods of the application, this subset is chosen by the developer and passed as input configuration to PANKTI. Focusing the generation on specific methods is important to ensure the relevance of the generated tests for the developers.

There are different ways for developers to choose a set of target methods. For example, developers might want to target the methods that implement the latest released feature, or to target the methods added or modified in the most recent commit [19]. Developers may manually specify the targets based on any combination of factors [4]. If there exists a test suite for the application, developers can pick as targets the methods that are not covered by this test suite [9], or choose targets based on test inadequacies [20]. For example, a method may be considered as a target if it is covered by the test suite, but has surviving mutants [12], or if no test fails if its body is removed [21]. The test generation process of PANKTI does not depend on how the target methods are selected.

C. Working Example

Here we illustrate the concepts of PANKTI with a concrete example, taken from PDFBox[3], an open-source application to work with PDF files [22]. Listing 1 shows a method called `getName` extracted from PDFBox. This method accepts four integer arguments, which are used as identifiers for the font name, platform, encoding, and language, respectively. It returns the name of the font corresponding to the integer identifiers, or `null` if the identifiers are not found. We say that `getName` is inadequately tested by the original test suite of PDFBox, since replacing its body with the statements `return null;`, `return "A";`, or `return "";` is undetected by the 7 existing test cases that trigger its execution [21]. This means that `getName` is weakly-tested because, in spite of being covered by multiple tests, its output is not specified by a single oracle in the test suite [23].

When we run PDFBox with a production workload, we observe that `getName` gets invoked 650 times, and we collect 563 production object profiles. One of these object profiles is

[3] https://github.com/apache/pdfbox/tree/2.0.21

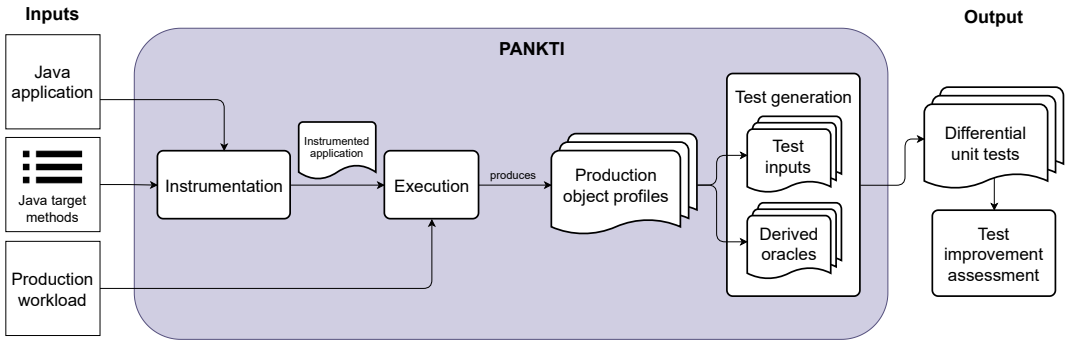


Fig. 1: The PANKTI pipeline: Transforming production workloads into differential unit tests to improve testing of target methods according to a test adequacy criterion.

```
1 String getName(int nId, int pId, int eId, int lId) {
2  Map<Integer,Map<Integer,Map<Integer,String>>> platforms =
       lookupTable.get(nId);
3  if (platforms == null) return null;

4  Map<Integer,Map<Integer,String>> encodings = platforms.get(pId);
5  if (encodings == null) return null;

6  Map<Integer,String> languages = encodings.get(eId);
7  if (languages == null) return null;

8  return languages.get(lId);
9 }
```

Listing 1: A target method for PANKTI

```
public java.lang.String getName(int, int, int, int);
  Code:
  ...
  197: invokestatic #67 // Method
       se/kth/castor/pankti/instrument/plugins/
       NamingTableAspect$TargetMethodAdvice.onBefore
  202: aload_0
  203: getfield #24 // Field lookupTable:Ljava/util/Map;
  ...
  306: invokestatic #96 // Method
       se/kth/castor/pankti/instrument/plugins/
       NamingTableAspect$TargetMethodAdvice.onReturn
  309: areturn
```

Listing 2: Excerpt showing bytecode transformation of the target method in Listing 1

Receiving object

```
<org.apache.fontbox.ttf.NamingTable>
  <tag>name</tag>
  <checkSum>3875177961</checkSum>
  <offset>298876</offset>
  <length>2955</length>
  <initialized>false</initialized>
  <font>...</font>
  <nameRecords>...<nameRecords>
  <lookupTable>...</lookupTable>
  <fontFamily>Liberation Sans</fontFamily>
  <fontSubFamily>Regular</fontSubFamily>
</org.apache.fontbox.ttf.NamingTable>
```

Parameter objects

```
<object-array>
  <int>6</int>
  <int>1</int>
  <int>0</int>
  <int>0</int>
</object-array>
```

Returned object

```
<string>
LiberationSans
</string>
```

Fig. 2: A production object profile for the target method in Listing 1

illustrated in Figure 2. It includes the *receiving* object, which is the `NamingTable` object that `getName` is invoked on, the four integers passed as *parameters* to it, and the `String` object returned as a *result* from this invocation of `getName`. All the elements of the object profile are serialized in XML format.

Listing 3 shows a differential unit test generated for `getName`, based on the production object profile illustrated in Figure 2. In the generated test, each element in the object profile is deserialized into its corresponding object. The generated assertion checks the equality between the `String` object as observed in production (expected value for this differential unit test), and the result returned from `getName` invoked on the deserialized `NamingTable` object, passing the same integer parameters.

D. Execution with Observability

A unique feature of PANKTI consists in observing the execution of a program in production to collect data for test generation. This execution with observability is articulated around the following three steps.

1) Code Instrumentation: The target methods under consideration are instrumented. The instrumentation of target methods consists of adding all the bytecode instructions required to monitor the execution context, before and just after the method invocation. For instance, the bytecode of the method `getName` in Listing 1 is transformed into Listing 2, post-instrumentation. The highlighted lines in the excerpt in Listing 2 show the instructions added to the original bytecode of `getName`. The probe before the method invocation collects a snapshot of the state of the receiving object, i.e., the object the method is being invoked on, as well as the state of any object passed as parameter to the method. The probe after the invocation, saves the object returned by the method.

PANKTI depends on the instrumentation of target methods at the bytecode level in a Java application in production. The key challenge here is to find a robust and reliable framework that can perform this instrumentation at scale, in the production environment.

2) Collection of Production Object Profiles: When the application executes under a specific workload, the instrumentation code for each target method, such as the one in Listing 2, generates one production object profile $<$

$method, receiving, parameters, result >$, each time the target method is invoked. PANKTI collects production object profiles in real time, and saves them to the disk. Production objects can be large and complex, with several levels of nesting with sub-objects. Serialization of very deeply nested objects is known to be hard [24], [25]. For this reason, PANKTI requires state-of-the-art serialization technology, which will be presented in II-G. Another crucial requirement of PANKTI is to save the objects in a human-readable format, because they are meant to be used in generated tests read by developers.

3) Selection of Object Profiles: During a pilot experiment, we made two important observations: some target methods are invoked thousands of times as the application executes; some of the serialized objects are very large. For instance, the method `toUnicode(int)` in PDFBox was invoked $30,840$ times with our selected production workload, and the `PDTrueTypeFont` object it was invoked on had nesting up to 13 levels deep. In production, there is a limited amount of space available to save object profiles. Based on these observations, PANKTI uses a threshold on the maximum size of serialized object profiles. The size threshold is one configuration parameter of PANKTI, for which the default value is 200 MB. This implies that object profiles are collected for each invocation of a target method, provided this threshold is not encountered for that method.

E. Generation of Test Cases

The test generation phase of PANKTI processes the object profiles collected in production to synthesize differential unit tests for the target methods that have been invoked at least once.

We have observed that a target method is often invoked in the same context and the set of collected production object profiles for one specific target method contains redundant data. For instance, the method `getName` in Listing 1 is invoked 650 times during execution with our selected workload. We collect production object profiles for 563 invocations, as per the size threshold discussed in the previous section. Of these 563 object profiles, 273 are unique. The first step for test generation is to select unique profiles.

Second, for each unique profile, PANKTI generates a differential unit test that follows a systematic template. First, the test case deserializes each part of the object profile: the receiver, the parameters, and the result object. Second, the test case invokes the $method$ on an object initialized with the deserialized $receiving$ and passes the set of deserialized $parameters$ to the method call. Third, we generate an assertion that expects the method invocation to return the same $result$ as observed in production. The serialized representations of objects can vary in size, depending on the complexity of the corresponding object. For the sake of readability, PANKTI includes smaller representations directly in the generated test, but generates resource files for serialized representations that are more sizable. The corresponding objects are then deserialized from these files during the execution of the generated test. This is also the methodology adopted by [26]. For example, PANKTI generated the test case shown in Listing 3, for the method

```
1 @Test
2 public void testGetName() {
3   // generated test for target NamingTable.getName
4   File fileReceiving = new File("receiving.xml");
5   NamingTable receivingObject = deserializeObject(fileReceiving);

6   String returnedObject = "<string>LiberationSans</string>";
7   String expectedObject = deserializeObject(returnedObject);

8   assertEquals(expectedObject,
9                receivingObject.getName(6, 1, 0, 0));
10 }
```

Listing 3: A generated differential unit test for the target method in Listing 1

in Listing 1, based on the production object profile illustrated in Figure 2. Lines 4 and 5 in the test fetch the $receiving$ object that was observed in production from a resource file and deserialize it. Lines 6 and 7 deserialize the $result$. Line 8 is the test oracle that asserts an equality between the $result$ that was observed in production and the value returned by the target method `getName` when it is invoked with the $parameters$ as obtained from the production object profile.

Deserialization of arbitrary objects is a hard problem [24], [25], and can sometimes result in an object that is not identical to the one observed in production. Since PANKTI fully relies on a full serialization-deserialization cycle, we need production grade serialization technology.

F. Test Improvement Assessment

The final step of the PANKTI pipeline assesses and filters the test cases generated for each target method. Here, we check three aspects: 1) that the generated test passes (see discussion in subsection V-A), 2) that it is not flaky[4], and 3) that the test improves the test adequacy criterion originally used to select the target methods.

The latter action is performed only if the application has a test suite and if the target methods were selected based on a test inadequacy criterion. In this case, the quality of the test suite before and after the test generation process is compared, per the test inadequacy criterion. If the quality is found to have improved, PANKTI is considered successful in generating a useful test.

G. Implementation

PANKTI is implemented in Java. By default, PANKTI focuses on generating tests for public and non-static methods. The focus on public methods provides a clear intention for the generated tests, similar to what developers do in the original test suite. Since PANKTI collects object profiles to generate the test inputs and the oracle, we do not target static methods in order to focus the test generation on the invocation of methods on objects and not classes.

PANKTI uses Glowroot[5] for code instrumentation and observability. Glowroot is an open-source, robust, and lightweight application performance management tool. PANKTI extends its production monitoring and instrumentation capabilities using its Plugin API.

[4] we run it 5 times per [27]
[5] https://glowroot.org/

The serialization and deserialization of production object profiles rely on the XStream library[6]. XStream is a state-of-the art library for serialization, is very robust, can handle arbitrarily complex objects, and allows the registration of custom converters. This library has also been proven to be an effective serialization tool in previous research [14], [28], [29]. Through a parameter, PANKTI can be configured to generate tests with serialized object strings in XML, JSON, and all other formats supported by XStream.

The test synthesis phase relies on Spoon [30] for test code generation. Spoon is a code analysis and transformation library for Java programs. PANKTI leverages the code generation features of Spoon to generate tests that are syntactically correct.

In the last phase, we assess the quality of the tests by compiling and running them. If the target methods are selected based on a test adequacy criterion, we determine if the PANKTI tests improve the test suite with respect to the same criterion.

PANKTI is made publicly available for facilitating reproduction and future research on the topic, at https://github.com/castor-software/pankti/.

III. Evaluation Methodology

This section describes our approach to evaluate PANKTI on real-world software systems. We discuss the criteria used to select the study subjects, their target methods and workloads, as well as the metrics computed for these projects.

A. Study Subjects

We perform a case-based research study [31] to evaluate PANKTI. We select a set of study subjects according to the following criteria: 1) The project is a real-world, open-source product implemented in Java, built with Maven, and has a test suite; 2) The project has more than 1000 stars and more than 1000 commits, which is indicative of a popular and mature project; 3) The project builds, the test suite passes, and the project can be deployed with the computing resources of our research lab; 4) A realistic production workload for the project can be executed in our laboratory environment.

We manually search for projects on GitHub that meet all of the above criteria, and select 3 of them. Jitsi/Jicofo is a component used in the Jitsi Meet videoconference system, a high-quality conferencing solution which supports audio, video, and text communication. Jicofo is responsible for maintaining the connection of the participants of a Jitsi Meet conference. PDFBox is a robust and mature project by the Apache Software Foundation for producing and reading PDF documents such as legal decisions, invoices, and contracts. This PDF application provides capabilities such as digital signatures, image conversion, etc. It includes standalone command-line tools that can be invoked on PDF and text documents [22]. Our third subject is BroadleafCommerce (henceforth referred to as Broadleaf), which is an enterprise e-commerce framework based on Spring. Broadleaf provides all necessary components for a web shopping experience, such as user and product management, a front-end website, and API services.

[6] http://x-stream.github.io/

TABLE I: Real-world projects used in our experiments

PROJECT	VERSION	#STARS	#COMMITS	LOC
Jitsi/Jicofo	stable-4857[7]	14K	1,248	28.32K
PDFBox	2.0.21[8]	1.3K	8,244	728.8K
Broadleaf	6.1.4-GA[9]	1.4K	16,964	197.5K

The version, number of stars (#STARS), number of commits (#COMMITS), and lines of code (LOC) for each of these projects are shown in Table I. The three projects have between 28K and 729K lines of code.

B. Experiments with Production Workloads

To generate differential unit tests using production data for each of our study subjects, we execute them in production conditions that are realistic, relevant, and reliable. This section describes our choice of workloads for each of the study subjects.

1) Production workloads for Jitsi/Jicofo: We hold a video-conference meeting to obtain a Jitsi/Jicofo workload. For this, we deploy the full Jitsi stack using the `docker-jitsi-meet` project. We create a meeting room for three members of the research lab for the experiment. The members join this meeting room from different IP addresses to have an hour-long meeting. During the meeting, all the participants turn on their camera and microphone. Each participant also sends at least one text message via the chat box.

2) Production workloads for PDFBox: As documentation and archival become more digitized across all domains, the Portable Document Format, or PDF, has emerged as a standard to ensure platform-independence and interoperability with multiple user environments. To select realistic and non-trivial workloads to experiment with PDFBox, we shortlist five PDF documents from GovDocs1 [32], an online corpora of PDF files made available for research and analysis. The same protocol has also been followed in previous research [33]. In order to have a good trade-off between the diversity of the files and the execution time for the experiments, we select the PDF documents based on the following criteria: 1) They have between 1 to 5 pages of text; 2) They include at least one image or photograph; and 3) Their size does not exceed 1.5 MB. The five documents that we use as part of the PDFBox workload are available as part of our replication package.

For each of the five selected documents, we invoke ten PDFBox features, which include the encryption of the PDF document with a password, decryption of an encrypted PDF document, extraction of text and images from a PDF document, etc. In summary, the workload to evaluate PANKTI on PDFBox is composed of 5×10 different invocations to the PDFBox command line tool.

3) Production workloads for Broadleaf: The workloads for Broadleaf consist of general steps to purchase products online as an end-user: a user visits the homepage of the website

[7] https://github.com/jitsi/jicofo/tree/stable/jitsi-meet_4857

[8] https://github.com/apache/pdfbox/tree/2.0.21

[9] https://github.com/BroadleafCommerce/BroadleafCommerce/tree/broadleaf-6.1.4-GA

first, then registers a new account and logs in, then views the web page to add some products into the shopping cart, and finally the user checks out the products and logs out. There are 871 HTTP requests in the workload, comprising of 865 GET requests and 6 POST requests. The GET requests are used to fetch product information and necessary files, such as JavaScript and CSS files. The POST requests are mainly for user registration, logging in, checking out products, and logging out.

C. Experiments with System Tests

System tests are typically designed to verify end-to-end uses of applications. They may exercise multiple and complex scenarios [34] or use cases [35] through the system. It is possible to generate differential unit tests from system tests, as proposed by the seminal papers of Elbaum *et al.* [14] and Saff *et al.* [36]. Inspired by this related work in the area of test improvement, we perform an experiment where we use PANKTI with system tests, instead of a production workload.

The goal of this experiment is to demonstrate the ability of PANKTI to generate unit tests by observing the execution of systems tests. So first, we need to select system tests. However, there is no strict definition of what constitutes a system test. Per previous research, we note that system tests and unit tests differ in the extent of the application they exercise, or the number of lines of code they cover [37]. Consequently, as actionable definition, we define system tests as those tests in the original test suite of our study subjects that cover at least 1500 lines of code in the application (this threshold of 1500 lines of code to identify system tests was used in previous research [38]). We measure the line coverage of each test with JaCoCo[10], in order to select the ones we consider as system tests.

We pass those system tests to PANKTI, which collects object profiles whenever target methods are invoked during their execution. The collected object profiles from system tests are then used to generate differential unit tests. The results of using PANKTI to carve differential unit tests from system tests will be discussed in subsection IV-E

D. Target Methods for The Evaluation

For our evaluation, we use "pseudo-tested methods" as the test adequacy criterion [15], [21] to select target methods in the three projects. A method is said to be pseudo-tested if it is invoked at least once when running the test suite, but no test fails if its body is removed. This means that no existing test is able to detect extreme transformations of these methods. Pseudo-testedness is a state-of-the-art criterion that developers care about, as evaluated in user studies [21]. We use the Descartes[11] tool to automatically identify pseudo-tested methods in the three study subjects.

This choice of targets for the evaluation implies a very clear way to assess the improvement provided by the PANKTI differential unit tests. A test for a specific target method is considered to be an improvement if the target method is no longer pseudo-tested.

[10]https://www.eclemma.org/jacoco/

[11]https://github.com/STAMP-project/pitest-descartes

E. Measurements

In this section, we introduce the metrics that we collect as part of our case-based study and assessment of PANKTI.

Testing metrics. First, we compute the following quantitative metrics capturing the test quality of the subjects:

1) Test suite method coverage, TMCov, represents the percentage of methods in the project that are covered by at least one test case in the original test suite.

2) Test suite line coverage, TLCov, represents the percentage of the lines of code in the project covered by at least one test case in the original test suite.

3) Workload method coverage, WMCov, determines the percentage of methods in the project that are invoked when we execute the project with our selected production workload.

4) Workload line coverage, WLCov, determines the percentage of lines of code in the project that are executed with the selected workload.

5) Target methods, #TARGET, is the number of target methods in the project. In the case of our evaluation, this is the number of pseudo-tested methods, i.e., the methods that are covered by at least one test case in the original test suite, but no test case fails if their body is replaced with a statement returning a default value; and

6) Effective target methods, #EFF_TARGET, is the number of target methods for which we use PANKTI to generate units tests. In the case of the experiments with production workloads, this is the number of pseudo-tested methods that are invoked when running the workload. In the case of the experiments with system tests, this is the number of pseudo-tested methods that are invoked when running the system tests.

Metrics TMCov and TLCov are indicators of the quality of the test suite of the project. Metrics WMCov and WLCov indicate the proportion of the project that is covered by our selected production workload. Metrics #TARGET and #EFF_TARGET are computed specifically as starting points for PANKTI. We use JaCoCo to obtain the coverage. #TARGET is computed with Descartes [21], while #EFF_TARGET is the subset of #TARGET covered during the experiments with production workloads and with system tests.

The second set of metrics we collect aim at assessing the ability of PANKTI at transforming production data into differential unit tests.

7) Target method invocations, #INVOCATIONS, is the number of invocations of an effective target method as the application executes with the production workload.

8) Collected object profiles, #COLLECTED, is the number of production object profiles for each effective target method that are captured as the application executes with the selected workload.

9) Unique object profiles, #UNIQUE, is the size of the subset of collected object profiles that includes unique combinations of the constituent elements.

10) Object profile size, SIZE is the size on disk of collected production object profiles.

11) Differential unit tests, #PANKTI_TESTS, is the number of test cases generated by PANKTI using the #UNIQUE object profiles.
12) Passing test cases, #PASSING, is the number of generated tests that pass when executed.
13) Failing test cases, #FAILING, is the number of generated tests that do not pass when executed.
14) PANKTI_STATUS (pseudo-tested or well-tested) is the classification of an effective target method, as obtained by Descartes, after the addition of the generated test to the test suite of the project. After adding the PANKTI-generated tests in the application test suite, a target method is either still pseudo-tested, or, is not pseudo-tested anymore. In the latter case, we give it the status of *well-tested*.

Metrics #INVOCATIONS, #COLLECTED, #UNIQUE, and SIZE are intermediate outputs of PANKTI as it monitors the target methods in the project in the production environment. Metrics #PANKTI_TESTS, #PASSING, #FAILING, and PANKTI_STATUS are the final outputs of PANKTI for the project, and are indicators of its success in generating differential unit tests for the target methods in the project.

IV. Experimental Results

This section discusses our findings when running PANKTI on our three study subjects. First, we present general data about the quality of the subjects' test suites. The subsequent sections discuss each case study separately.

A. Descriptive Statistics

Table II summarizes the key metrics about the original test suite and about the workload: TMCov, TLCov, WMCov, WLCov, #TARGET, and #EFF_TARGET (see subsection III-E). Let us take the case of Jitsi/Jicofo as an example. The original test suite of Jitsi/Jicofo covers 49.4% of the methods, and 46.7% of the lines in the project. Our selected production workload for Jitsi/Jicofo covers 48.9% of the methods and 46.2% of the lines in the project. We find that 29 public and non-static methods are pseudo-tested in Jitsi/Jicofo (#TARGET). This means that they are covered by at least one test in the suite, but no test is able to detect extreme mutations in these methods. When we instrument these 29 methods and observe Jitsi in production, we find that all 29 of them get invoked (#EFF_TARGET).

Per Table II, the original test suite of all the three projects varies in coverage, with a highest line coverage of 53.5%. Moreover, the coverage achieved by the workload is lower than the test coverage for all of the three cases. This observation is consistent with the findings of Wang *et al.* [9].

Pseudo-tested methods are present in all projects, confirming the results of Vera-Perez *et al.* [21] about the prevalence of such methods. These methods, covered by the original test suite, but weakly-tested, provide a false sense of trust in the test suite, which we aim at mitigating with new test cases generated by PANKTI.

On executing each project with its corresponding workload, some of these pseudo-tested methods are indeed invoked in production. This is indicated by the column "#EFF_TARGET" in Table II. This is the baseline that PANKTI aims to improve: we want to generate tests such that some of those pseudo-tested methods become well-tested: 29, 46, and 11 target methods for Jitsi/Jicofo, PDFBox, and Broadleaf, respectively.

Fig. 3: A Jitsi Meeting with PANKTI Attached

B. Case Study 1: Jitsi/Jicofo

PANKTI targets 29 pseudo-tested methods in Jitsi/Jicofo for instrumentation. Then, we run Jitsi for a videoconference during which we collect object profiles. We, the authors, perform this videoconference to discuss this paper. Figure 3 presents a screenshot of this meeting, where PANKTI was attached to the Jitsi/Jicofo component. During the one-hour meeting, target methods keep getting invoked periodically, triggering object profile collection, yet no participant observes a degradation of Jitsi's user experience such as a delay, or a crash.

We now discuss the results of PANKTI on Jitsi, presented in Table III. Table III is grouped by the study subject and sorted by the invocation count of the target methods. Each row corresponds to one effective target method, and includes the values of the metrics introduced in subsection III-E. For each of these methods, Table III shows the number of times it was invoked with the production workload (#INVOCATIONS), the number of object profiles corresponding to these invocations that are serialized (#COLLECTED), the number of collected object profiles that are unique (#UNIQUE), as well as the number of differential unit tests generated by PANKTI (#PANKTI_TESTS). Of the tests generated, the number of tests that pass (#PASSING), and the ones that do not (#FAILING), are also indicated. The Descartes classification (PANKTI_STATUS) of the target method after the addition of the generated tests to the test suite of the project is shown in the last column, it either remains the same as before, i.e., "pseudo-tested", or upgrades to "well-tested".

Rows 1 to 29 of Table III show the experimental results for Jitsi. The 29 effective target methods are invoked 152 times during the experiment. PANKTI collects 110 object profiles, 20 of which are unique. Based on the unique object profiles, PANKTI generates 20 differential unit tests in total for Jitsi/Jicofo. All of the generated tests pass and improve the quality of the Jitsi/Jicofo test suite, transforming 19/29 (65.5%) pseudo-tested methods into well-tested ones.

Let us look at one successful case. Listing 4 is the source code of method

TABLE II: Original test suite method and line coverage (TMCov, TLCov), workload method and line coverage (WMCov, WLCov), number of pseudo-tested methods (#TARGET) and number of pseudo-tested methods invoked when running the workload (#EFF_TARGET) for each study subject

PROJECT	TMCov	TLCov	WMCov	WLCov	#TARGET	#EFF_TARGET
Jitsi/Jicofo	49.4 % (667 / 1,350)	46.7 % (3,537 / 7,571)	48.9 % (660 / 1,350)	46.2 % (3,500 / 7,571)	29	29
PDFBox	54.8 % (6,049 / 11,042)	53.5 % (34,653 / 64,787)	21.6 % (2,390 / 11,042)	21.0 % (13,630 / 64,787)	138	46
Broadleaf	23.9 % (1,478 / 6,173)	23.7 % (5,846 / 24,667)	23.1 % (1,424 / 6,173)	19.2 % (4,735 / 24,667)	32	11

```
1 public int computeParticipantEgressPacketRatePps() {
2   return (numberOfSpeakers * maxPacketRatePps[0]
3         + (numberOfGlobalSenders - 2)
4         * maxPacketRatePps[1] + maxPacketRatePps[3]);
5 }
```

Listing 4: A target method called `computeParticipantEgressPacketRatePps` in Jitsi/Jicofo

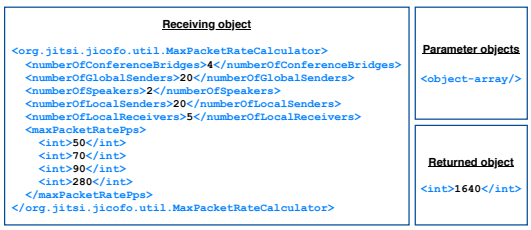


Fig. 4: A production object profile for the method `computeParticipantEgressPacketRatePps()` in Jitsi/Jicofo

`computeParticipantEgressPacketRatePps` (row 27 in Table III) in class `MaxPacketRateCalculator`. The method is covered by developer-written unit tests, but is identified as pseudo-tested: if the method body is replaced with `return -1;`, `return 0;`, or `return 1;`, no test in the original test suite fails. By observing the behavior of this method in production, PANKTI collects the object profile shown in Figure 4. A new differential unit test is then generated based on this profile, shown in Listing 5. The generated test includes an explicit assertion for the behavior of `computeParticipantEgressPacketRatePps`. After adding this generated differential unit test to the test suite, the method is not pseudo-tested anymore.

During the object profile collection phase, PANKTI collects 110 object profiles for 20 out of 29 (70.0%) methods. This corresponds to a total of 9.79 KB collected data, for object profiles that vary between 0.48 KB and 0.50 KB, with a median value of 0.49 KB. For the duration of our experiment, none of the methods reach the threshold of the file size (200 MB) for their object profile collection.

We observe that PANKTI collects no object profile for 9 methods, and does not collect all the profiles for 3 methods (rows 1, 3, and 4). This occurs when PANKTI faces extreme situations for object serialization at runtime: 1) A `ConcurrentModificationException` happens if

```
1 @Test
2 public void testCPEgressPacketRatePps() {
3   // content of the serialized receiving object from Fig. 4
4   String receivingObjectStr =
          "<org.jitsi.jicofo.util.MaxPacketRateCalculator> ...
          </org.jitsi.jicofo.util.MaxPacketRateCalculator>";

5   MaxPacketRateCalculator receivingObject =
          deserializeObject(receivingObjectStr);

6   assertEquals(1640,
          receivingObject.computeParticipantEgressPacketRatePps());
7 }
```

Listing 5: A generated differential unit test for the `computeParticipantEgressPacketRatePps` method

XStream tries to serialize an object that is being changed. This may happen if the object contains some fields which are accessed by multiple threads frequently. 2) A `"no converter specified"` exception happens if XStream does not know how to serialize some members in the object. This case only happens when XStream fails to serialize class `NetworkAddressManagerServiceImpl` in the package `net.java.sip.communicator.impl.netaddr` of Jitsi/Jicofo. This is due to the fact that this class contains fields that are related to system resources such as sockets and threads, which are fundamentally unserializable. 3) a `"security rule violation"` happens if XStream refuses to serialize a specific class which is protected by security code. Overall, this clearly show that serialization of arbitrary objects is fundamentally hard, and not yet solved by the state-of-the-art of serialization. After the profile collection phase, PANKTI successfully generates test cases for all of the covered methods. All these new test cases pass.

Highlight from the Jitsi/Jicofo Experiment

The experiment on Jitsi/Jicofo uses a production workload for videoconferencing, an essential feature of remote working and social interactions in 2020 and 2021. PANKTI successfully collects 110 object profiles for 20 methods. PANKTI generates 20 new test cases that improve the testing of 19 out of 29 target methods of Jitsi. This experiment shows that PANKTI does not perturb user experience, users can have a smooth videoconference while generating tests.

C. Case Study 2: PDFBox

There are 138 pseudo-tested methods in PDFBox. The production workload covers 46 of these methods, which all

TABLE III: Experimental results from executing production workloads of the study subjects used to evaluate PANKTI

#	CLASS_METHOD	#INVOCATIONS	#COLLECTED	#UNIQUE	#PANKTI_TESTS	#PASSING	#FAILING	PANKTI_STATUS
1	JitsiMeetConfig_isTccEnabled	20	**18** / 20 (90%)	**1** / 18 (0.05%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
2	JitsiMeetConfig_isOctoEnabled	15	**15** / 15 (100%)	**1** / 15 (0.06%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
3	JitsiMeetConfig_getMinBitrate	12	**11** / 12 (91.6%)	**1** / 11 (0.09%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
4	JitsiMeetConfig_getStartBitrate	12	**11** / 12 (91.6%)	**1** / 11 (0.09%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
5	JitsiMeetConfig_isRembEnabled	12	**12** / 12 (100%)	**1** / 12 (0.08%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
6	JitsiMeetServices_getJigasiDetector	8	**0** / 8 (0%)	0	0	0	0	**pseudo-tested**
7	JitsiMeetServices_getJibriDetector	6	**0** / 6 (0%)	0	0	0	0	**pseudo-tested**
8	JitsiMeetConfig_getMinParticipants	5	**5** / 5 (100%)	**1** / 5 (20%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
9	ColibriConferenceImpl_hasJustAllocated	4	**0** / 4 (0%)	0	0	0	0	**pseudo-tested**
10	JitsiMeetConfig_getEnforcedVideobridge	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**pseudo-tested**
11	JitsiMeetConfig_getOpusMaxAverageBitrate	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
12	JitsiMeetConfig_getStartAudioMuted	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
13	JitsiMeetConfig_getStartVideoMuted	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
14	JitsiMeetConfig_isRtxEnabled	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
15	JitsiMeetConfig_stereoEnabled	4	**4** / 4 (100%)	**1** / 4 (25%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
16	Participant_hasDtlsSupport	4	**0** / 4 (0%)	0	0	0	0	**pseudo-tested**
17	Participant_hasIceSupport	4	**0** / 4 (0%)	0	0	0	0	**pseudo-tested**
18	Participant_hasRtxSupport	4	**0** / 4 (0%)	0	0	0	0	**pseudo-tested**
19	ProtocolProviderHandler_getXmppConnection	4	**0** / 4 (0%)	0	0	0	0	**pseudo-tested**
20	JitsiMeetConfig_isLipSyncEnabled	3	**3** / 3 (100%)	**1** / 3 (33.3%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
21	JitsiMeetConfig_useRoomAsSharedDocName	3	**3** / 3 (100%)	**1** / 3 (33.3%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
22	JitsiMeetServices_getSipJibriDetector	3	**0** / 3 (0%)	0	0	0	0	**pseudo-tested**
23	JitsiMeetConfig_getAudioPacketDelay	2	**2** / 2 (100%)	**1** / 2 (50%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
24	JitsiMeetConfig_getChannelLastN	2	**2** / 2 (100%)	**1** / 2 (50%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
25	JitsiMeetConferenceImpl$BridgeSession_terminate	1	**0** / 1 (0%)	0	0	0	0	**pseudo-tested**
26	MaxPacketRateCalculator_computeEgressPacketRatePps	1	**1** / 1 (100%)	**1** / 1 (100%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
27	MaxPacketRateCalculator_computeIngressPacketRatePps	1	**1** / 1 (100%)	**1** / 1 (100%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
28	MaxPacketRateCalculator_computeParticipantEgressPacketRatePps	1	**1** / 1 (100%)	**1** / 1 (100%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
29	MaxPacketRateCalculator_computeSenderIngressPacketRatePps	1	**1** / 1 (100%)	**1** / 1 (100%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
	JICOFO_TOTAL	**152**	**110** / 152 (72.4%)	**20** / 110 (18.2%)	**20**	**20** / 20 (100%)	**0** / 20 (0%)	**well-tested : 19** / 29 **pseudo-tested : 10** / 29
30	RenderingMode_isFill	30944	**30944** / 30944 (100%)	**1** / 30944 (0.00003%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
31	PDSimpleFont_toUnicode	30840	**37** / 30840 (0.001%)	**35** / 37 (94.5%)	35	**35** / 35 (100%)	**0** / 35 (0%)	**well-tested**
32	GlyfSimpleDescript_getFlags	23471	**23471** / 23471 (100%)	**11730** / 23471 (49.9%)	11730	**11730** / 11730 (100%)	**0** / 11730 (0%)	**well-tested**
33	TTFGlyph2D_getPathForCharacterCode	15472	**34** / 15472 (0.002%)	**34** / 34 (100%)	34	**0** / 34 (0%)	**34** / 34 (100%)	**pseudo-tested**
34	TTFGlyph2D_getPathForGID	15472	**34** / 15472 (0.002%)	**34** / 34 (100%)	34	**0** /34 (0%)	**34** / 34 (100%)	**pseudo-tested**
35	PDTrueTypeFont_getPath	782	**36** / 782 (0.04%)	**36** / 36 (100%)	36	**0** / 36 (0%)	**36** / 36 (100%)	**pseudo-tested**
36	NamingTable_getName	650	**563** / 650 (86.6%)	**273** / 563 (48.5%)	273	**273** / 273 (100%)	**0** / 273 (0%)	**well-tested**
37	DefaultResourceCache_getFont	546	**29** / 546 (0.05%)	**29** / 29 (100%)	29	**5** / 29 (0.17%)	**24** / 29 (82.7%)	**pseudo-tested**
38	HorizontalMetricsTable_getLeftSideBearing	431	**360** / 431 (83.5%)	**360** / 360 (100%)	360	**360** / 360 (100%)	**0** / 360 (0%)	**well-tested**
39	TTFDataStream_readUnsignedByteArray	431	**431** / 431 (100%)	**431** / 431 (100%)	431	**431** / 431 (100%)	**0** / 431 (0%)	**well-tested**
40	HorizontalMetricsTable_getAdvanceWidth	275	**275** / 275 (100%)	**174** / 275 (63.3%)	174	**174** / 174 (100%)	**0** / 174 (0%)	**well-tested**
41	TrueTypeFont_getAdvanceWidth	275	**275** / 275 (100%)	**174** / 275 (63.3%)	174	**174** / 174 (100%)	**0** / 174 (0%)	**well-tested**
42	PDSimpleFont_hasExplicitWidth	274	**144** / 274 (52.5%)	**117** / 144 (81.2%)	117	**117** / 117 (100%)	**0** / 117 (0%)	**well-tested**
43	PDFontDescriptor_getFlags	227	**227** / 227 (100%)	**54** / 227 (23.8%)	54	**54** / 54 (100%)	**0** / 54 (0%)	**well-tested**
44	FontCache_getFont	115	**115** / 115 (100%)	**106** / 115 (92.2%)	106	**83** / 106 (78.3%)	**23** / 106 (21.7%)	**pseudo-tested**
45	DefaultResourceCache_getColorSpace	72	**25** / 72 (34.7%)	**23** / 25 (92%)	23	**4** / 23 (17.4%)	**19** / 23 (82.6%)	**pseudo-tested**
46	DefaultResourceCache_getXObject	66	**15** / 66 (22.7%)	**15** / 15 (100%)	15	**13** / 15 (86.6%)	**2** / 15 (13.3%)	**pseudo-tested**
47	PDImageXObject_getInterpolate	66	**8** / 66 (12.1%)	**8** / 8 (100%)	8	**8** / 8 (100%)	**0** / 8 (0%)	**well-tested**
48	PDImageXObject_getMask	66	**13** / 66 (19.7%)	**13** / 13 (100%)	13	**13** / 13 (100%)	**0** / 13 (0%)	**pseudo-tested**
49	PDCIDFontType2_codeToGID	58	**38** / 58 (65.5%)	**7** / 38 (18.4%)	7	**7** / 7 (100%)	**0** / 7 (0%)	**well-tested**
50	DefaultResourceCache_getExtGState	49	**24** / 49 (49%)	**22** / 24 (91.6%)	22	**4** / 22 (18.1%)	**18** / 22 (81.8%)	**pseudo-tested**
51	PDResources_getExtGState	49	**19** / 49 (38.8%)	**18** / 19 (94.7%)	18	**0** / 18 (0%)	**18** / 18 (100%)	**pseudo-tested**
52	PDExtendedGraphicsState_getStrokingOverprintControl	48	**48** / 48 (100%)	**3** / 48 (0.06%)	3	**3** / 3 (100%)	**0** / 3 (0%)	**well-tested**
53	COSParser_getEncryption	46	**38** / 46 (82.6%)	**33** / 38 (86.8%)	33	**13** / 33 (39.4%)	**20** / 33 (60.6%)	**pseudo-tested**
54	XrefTrailerResolver_getXrefType	46	**46** / 46 (100%)	**17** / 46 (36.9%)	17	**17** / 17 (100%)	**0** / 17 (0%)	**well-tested**
55	PDImageXObject_getColorKeyMask	39	**7** / 39 (17.9%)	**7** / 7 (100%)	7	**7** / 7 (100%)	**0** / 7 (0%)	**pseudo-tested**
56	PDImageXObject_getDecode	39	**7** / 39 (17.9%)	**7** / 7 (100%)	7	**7** / 7 (100%)	**0** / 7 (0%)	**pseudo-tested**
57	PDExtendedGraphicsState_getAutomaticStrokeAdjustment	33	**33** / 33 (100%)	**4** / 33 (12.1%)	4	**4** / 4 (100%)	**0** / 4 (0%)	**well-tested**
58	PDImageXObject_getOptionalContent	33	**7** / 33 (21.2%)	**7** / 7 (100%)	7	**7** / 7 (100%)	**0** / 7 (0%)	**pseudo-tested**
59	PDExtendedGraphicsState_getNonStrokingOverprintControl	24	**24** / 24 (100%)	**3** / 24 (12.5%)	3	**3** / 3 (100%)	**0** / 3 (0%)	**well-tested**
60	PDExtendedGraphicsState_getOverprintMode	24	**24** / 24 (100%)	**3** / 24 (12.5%)	3	**3** / 3 (100%)	**0** / 3 (0%)	**well-tested**
61	PDFontDescriptor_isSymbolic	24	**24** / 24 (100%)	**20** / 24 (83.3%)	20	**20** / 20 (100%)	**0** / 20 (0%)	**well-tested**
62	PDExtendedGraphicsState_getSmoothnessTolerance	21	**21** / 21 (100%)	**3** / 21 (14.3%)	3	**3** / 3 (100%)	**0** / 3 (0%)	**well-tested**
63	PDFontDescriptor_isFixedPitch	21	**21** / 21 (100%)	**6** / 21 (28.6%)	6	**6** / 6 (100%)	**0** / 6 (0%)	**well-tested**
64	PDFontDescriptor_isItalic	21	**21** / 21 (100%)	**6** / 21 (28.6%)	6	**6** / 6 (100%)	**0** / 6 (0%)	**well-tested**
65	PDFontDescriptor_isSerif	21	**21** / 21 (100%)	**6** / 21 (28.6%)	6	**6** / 6 (100%)	**0** / 6 (0%)	**well-tested**
66	PDExtendedGraphicsState_getSoftMask	20	**20** / 20 (100%)	**4** / 20 (20%)	4	**4** / 4 (100%)	**0** / 4 (0%)	**pseudo-tested**
67	PDICCBased_getDefaultDecode	16	**16** / 16 (100%)	**8** / 18 (44.4%)	8	**8** / 8 (100%)	**0** / 8 (0%)	**well-tested**
68	PDCIDSystemInfo_getOrdering	12	**12** / 12 (100%)	**1** / 12 (0.08%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
69	PDExtendedGraphicsState_getAlphaSourceFlag	12	**12** / 12 (100%)	**1** / 12 (0.08%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
70	PDExtendedGraphicsState_getNonStrokingAlphaConstant	12	**12** / 12 (100%)	**1** / 12 (0.08%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
71	PDExtendedGraphicsState_getStrokingAlphaConstant	12	**12** / 12 (100%)	**1** / 12 (0.08%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
72	PDDocumentCatalog_getViewerPreferences	8	**8** / 8 (100%)	**8** / 8 (100%)	8	**6** / 8 (75%)	**2** / 8 (25%)	**pseudo-tested**
73	PDCIDFontType2_getPath	6	**6** / 6 (100%)	**6** / 6 (100%)	6	**0** / 6 (0%)	**6** / 6 (100%)	**pseudo-tested**
74	PDCIDFont_getCIDSystemInfo	3	**3** / 3 (100%)	**1** / 3 (33.3%)	1	**0** / 1 (0%)	**1** / 1 (100%)	**pseudo-tested**
75	PDCIDSystemInfo_getRegistry	3	**3** / 3 (100%)	**1** / 3 (33.3%)	1	**1** / 1 (100%)	**0** / 1 (0%)	**well-tested**
	PDFBOX_TOTAL	**121175**	**57563** / 121175 (47.5%)	**13851** / 57563 (24%)	**13851**	**13614** / 13851 (98.2%)	**237** / 13851 (1.8%)	**well-tested : 28** / 46 **pseudo-tested : 18** / 46
76	SkuImpl_getCurrency	415	**415** / 415 (100%)	**128** / 415 (30.8%)	128	**128** / 128 (100%)	**0** / 128 (0%)	**pseudo-tested**
77	SkuImpl_getName	394	**394** / 394 (100%)	**182** / 394 (46.1%)	182	**84** / 182 (46.1%)	**98** / 182 (53.8%)	**well-tested**
78	SkuImpl_isTaxable	24	**24** / 24 (100%)	**12** / 24 (50%)	12	**12** / 12 (100%)	**0** / 12 (0%)	**well-tested**
79	OrderImpl_finalizeItemPrices	6	**6** / 6 (100%)	**6** / 6 (100%)	6	**6** / 6 (100%)	**0** / 6 (0%)	**well-tested**
80	OrderImpl_getHasOrderAdjustments	6	**6** / 6 (100%)	**6** / 6 (100%)	6	**6** / 6 (100%)	**0** / 6 (0%)	**well-tested**
81	DynamicSkuPrices_getPrice	4	**4** / 4 (100%)	**2** / 4 (50%)	2	**2** / 2 (100%)	**0** / 2 (0%)	**well-tested**
82	FixedPrice...Provider_canCalculateCostForFulfillmentGroup	4	**4** / 4 (100%)	**2** / 4 (50%)	2	**2** / 2 (100%)	**0** / 2 (0%)	**well-tested**
83	SkuImpl_getBaseRetailPrice	4	**4** / 4 (100%)	**4** / 4 (100%)	4	**0** / 4 (0%)	**4** / 4 (100%)	**pseudo-tested**
84	SkuImpl_getBaseSalePrice	4	**4** / 4 (100%)	**4** / 4 (100%)	4	**4** / 4 (100%)	**0** / 4 (0%)	**pseudo-tested**
85	SkuImpl_getPriceData	4	**4** / 4 (100%)	**4** / 4 (100%)	4	**0** / 4 (0%)	**4** / 4 (100%)	**pseudo-tested**
86	FixedPriceFulfillmentOptionImpl_getPrice	2	**2** / 2 (100%)	**1** / 2 (50%)	1	**0** / 1 (0%)	**1** / 1 (100%)	**pseudo-tested**
	BROADLEAF_TOTAL	**867**	**867** / 867 (100%)	**351** / 867 (40.5%)	**351**	**244** / 351 (69.5%)	**107** / 351 (30.5%)	**well-tested : 6** / 11 **pseudo-tested : 5** / 11
	TOTAL	**122194**	**58540** / 122194 (47.9%)	**14222** / 58540 (24.3%)	**14222**	**13878** / 14222 (97.6%)	**344** / 14222 (2.4%)	**well-tested : 53** / 86 **pseudo-tested : 33** / 86

```
1 public boolean isFill() {
2   return this == FILL || this == FILL_STROKE || this == FILL_CLIP
        || this == FILL_STROKE_CLIP;
3 }
```

Listing 6: A target method called `isFill` in PDFBox

become effective target methods for PANKTI. These methods are included in rows 30 to 75 in Table III. From the 121,175 total invocations of these target methods in production, which occur throughout the course of our experiment, we collect 57,563 production object profiles.

The collected object profiles amount to 5.5 GB of disk space. We notice that for 19 of the 46 target methods, PANKTI does not collect all the object profiles. This means that the size threshold of 200 MB is encountered for these 19 methods. The distribution of the sizes of the collected object profiles is illustrated in Figure 5. From the figure, we see that most of the collected object profiles for PDFBox range from 0 to 10 MB in size. The smallest object profile we collect is 158 bytes in size, with the largest one being nearly 55 MB. Larger object profiles correspond to complex objects with many levels of nesting, the maximum level of nesting we observe in a collected object profile is 36. As explained in subsubsection II-D3, we limit the total storage size for object profiles; to trade between the value of the collected profiles and the performance of PANKTI. In the extreme cases, such as those in rows 31, 33, and 34 in Table III, PANKTI must store only a fraction of the profiles.

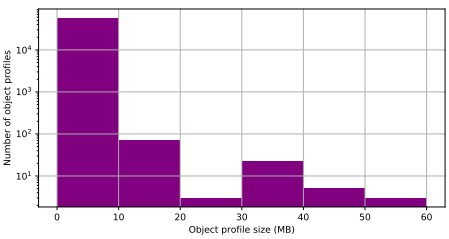


Fig. 5: The distribution of the collected object profile sizes for PDFBox over 57,563 profiles from 46 methods

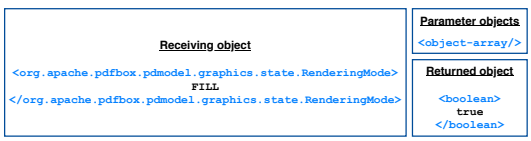


Fig. 6: A production object profile for the method `isFill()` in PDFBox

The set of 57,563 object profiles includes 13,851 profiles that are unique. Per the algorithm presented in subsection II-E, PANKTI generates one test method corresponding to each of these profiles. The number of generated tests ranges between a minimum of 1 for 7 target methods to a maximum of 11,730 for 1 target method. In total, we execute 13,851 differential unit tests generated by PANKTI. 13,614 (98.3%) of these tests pass while 237 tests fail. We discuss the reasons for the failure of the generated tests in subsection V-A.

```
1 @Test
2 public void testIsFill() {
3   String receivingObjectStr =
4   "<org.apache.pdfbox.pdmodel.graphics.state.RenderingMode>
5   FILL</org.apache.pdfbox.pdmodel.graphics.state.RenderingMode>";

6   RenderingMode receivingObject =
        deserializeObject(receivingObjectStr);

7   assertEquals(true, receivingObject.isFill());
8 }
```

Listing 7: A generated differential unit test for the `isFill` method

Thanks to these generated tests, 28 of the 46 target methods (60.9%) switch from being pseudo-tested to well-tested. For example, the method called `isFill` in the class `RenderingMode` (row 30 in Table III) is one method that becomes well-tested after the addition of a PANKTI-generated test to the test suite of PDFBox.

Let us now discuss the case of the heavily-invoked method `isFill`, shown in Listing 6. It takes no parameter, and returns a boolean value, depending on certain values of the rendering mode for a PDF document. It is invoked 30,944 times by our production workload. We collect all 30,944 object profiles for this method, of 158 bytes each. Since all these object profiles are the same, shown in Figure 6, PANKTI generates one single test method for it. Its status upgrades from pseudo-tested to well-tested with the addition of this new test. The generated test is shown in Listing 7.

We formatted the generated test to be consistent with the structure of test methods in the PDFBox codebase, and suggested it to the developers of PDFBox in the form of a pull request[12] on GitHub. The pull request was accepted by a developer, and the new test case is now part of the test suite for PDFBox. This indicates that PANKTI can capture relevant information in production and turn them into valid test inputs and oracles.

Highlight from the PDFBox Experiment

With a PDF manipulation workload for PDFBox, PANKTI generates differential unit tests that improve the testing of 28 out of 46 target pseudo-tested methods. This case study shows that the PANKTI monitoring and test generation pipeline scales to thousands of method invocations and object profiles. Furthermore, this case study validates the design decision of thresholding the number and size of collected object profiles.

D. Case Study 3: Broadleaf

Descartes finds 32 pseudo-tested methods when running the original test suite of the Broadleaf e-commerce application. Next, we deploy Broadleaf together with PANKTI and execute it with the typical e-commerce workload described in subsection III-B. PANKTI finds that 11 out of the 32 pseudo-tested methods are executed by our workload. These 11 effective target methods are shown in rows 76 to 86 in Table III. They are invoked 867 times during the experiment.

[12] https://github.com/apache/pdfbox/pull/88

```
1 @Test
2 public void testGetHasOrderAdjustments() {
3   File fileReceiving = new File("receiving.xml");
4   OrderImpl receivingObj = deserializeObject(fileReceiving);
5   assertEquals(false, receivingObj.getHasOrderAdjustments());
6 }
```

Listing 8: A generated differential unit test for the `getHasOrderAdjustments` method

The size threshold of 200 MB is not encountered for any of these methods. We observe that target methods are invoked throughout the duration of the experiment. Consequently, the object profiles for all invocations are successfully serialized to files by PANKTI. During the test generation phase, PANKTI identifies 351 unique object profiles from the collected ones. All of the unique object profiles are transformed into unit tests without any exception.

When executing the 351 generated differential unit tests, 244 (69.5%) of them pass, while the other 107 (30.5%) fail. There are several reasons why some generated tests may fail, which will be discussed in subsection V-A. The passing tests improve the test quality for 6 out of the 11 (54.5%) target methods. This means that PANKTI generates tests that better specify the behavior of these 6 methods.

Let us take row 80 in Table III as an example. In Broadleaf's original test suite, the method `getHasOrderAdjustments` in class `OrderImpl` is pseudo-tested. If the body of this method is replaced by `return true;` or `return false;`, none of the 6 test cases that reach it fail, meaning that even though the method is covered, its correct behavior is not specified by the existing tests. In production, PANKTI collects 6 unique object profiles for this method. Listing 8 presents one of the test cases generated according to the object profiles. In the generated test, the receiving object is deserialized from the profile (since the profile is large, the object profile is read from a file for the sake of test readability) (line 4). According to the collected production data, the method invocation is expected to return `false`. Thus `assertEquals(false, ...)` is generated to verify the actual return value of method `getHasOrderAdjustments` (line 5). After the addition of this newly generated differential unit test to the test suite of Broadleaf, Descartes does not detect `getHasOrderAdjustments` as a pseudo-tested method anymore. This implies that the values collected by observing the interactions of a real user with the website can contribute to the improvement of the overall quality of the test suite of Broadleaf.

Highlight from the Broadleaf Experiment

PANKTI collects 351 unique object profiles while running the production workload on the e-commerce application Broadleaf, and successfully generates a test case for each profile. In total, 244 test cases improve the test thoroughness for 6 out of the 11 target pseudo-tested methods. Overall, this case study shows that PANKTI works well in a typical web, HTTP-based architecture which is representative of many enterprise applications.

E. Experimental Results with System Tests

Table IV summarizes the results of the assessment of PANKTI's ability to trace the execution of system tests in order to generate differential unit tests, per the protocol of subsection III-C. For each of the three study subjects. the number of system tests we identify is given in the column #SYSTEM_TESTS. #EFF_TARGET represents the number of target pseudo-tested methods invoked during the execution of these system tests. The number of differential unit tests generated using the object profiles collected by PANKTI is signified by #PANKTI_TESTS. Of these, the tests that pass are represented as #PASSING, and the ones that do not, as #FAILING. In column PANKTI_STATUS we provide the status of the invoked target methods after adding the PANKTI-generated differential unit tests to the test suite of the project. They either remain pseudo-tested, or become well-tested.

The table indicates that there are 11, 150, and 6 system tests in Jicofo, PDFBox, and Broadleaf, respectively. PANKTI instruments 29, 138, and 32 pseudo-tested methods in these projects. Of these instrumented methods, 29, 111, and 26 are invoked during the execution of the system tests, and are the effective targets for test generation. PANKTI successfully generates 20, 18, 291, and 692 differential unit tests for these target methods. Of these tests, 100%, 96.5%, and 98.8% pass. As a result of the tests generated by PANKTI, 62% (18 of 29), 59.5% (66 of 111), and 46.1% (12 of 26) of targets in Jicofo, PDFBox, and Broadleaf are no longer pseudo-tested.

We note that the set of methods invoked during our experiments with production workloads differs from the ones invoked with system tests, i.e., for the same #TARGET, #EFF_TARGET is different for the two experiments. We observe that the production workload for PDFBox invokes 2 methods which are never reached by system tests. One of these methods becomes well-tested as a result of the tests generated by PANKTI. For Broadleaf, the production workload reaches 3 methods not invoked at all by system tests, of which 2 methods become well-tested due to the addition of the new tests. For Jicofo, both the production workload and the system tests cover the same set of methods. These observations confirm that system tests can miss some behavior exercised in production [9]. Consequently, a production workload is an essential complement to system tests for test improvement, as it provides additional test generation targets for PANKTI.

In total, PANKTI generates tests for 166 out of the 199 instrumented pseudo-tested methods, using 167 system tests across the three study subjects. 96 of these methods (57.8%) become well-tested as a result of the tests automatically generated by PANKTI. This experiment demonstrates that PANKTI can generate differential unit tests from system tests.

V. Discussion

We now discuss the key aspects of PANKTI.

A. When PANKTI-generated Tests Fail

From Table III, we observe that a total of 344 PANKTI-generated unit tests fail (2.4% of the generated tests). We manually analyze a random sample of them to understand the

TABLE IV: Experimental results from executing system tests of the study subjects used to evaluate PANKTI

PROJECT	#SYSTEM_TESTS	#EFF_TARGET	#PANKTI_TESTS	#PASSING	#FAILING	PANKTI_STATUS
Jicofo	11	**29** / 29	20	**20** / 20	**0** / 20	**well-tested: 18** / 29 **pseudo-tested: 11** / 29
PDFBox	150	**111** / 138	18,291	**17,655** / 18,291	**636** / 18,291	**well-tested: 66** / 111 **pseudo-tested: 45** / 111
Broadleaf	6	**26** / 32	692	**684** / 692	**8** / 692	**well-tested: 12** / 26 **pseudo-tested: 14** / 26
TOTAL	167	**166** / 199	19,003	**18,359** / 19,003	**644** / 19,003	**well-tested: 96** / 166 **pseudo-tested: 70** / 166

causes. First, we find that no test generated for methods that return a primitive type, an array of a primitive type, a wrapper object for a primitive type, or an object of an enum type fail. For the tests that fail in PDFBox and Broadleaf, all methods return complex objects. Our manual analysis reveals four main reasons for a failure in a PANKTI-generated test.

Comparison with the `equals` method can fail for arbitrary objects. An `assertEquals` JUnit assertion internally invokes the `equals` method on the two objects being compared. According to the default implementation of `equals`, a value of `true` is returned for two objects if they have the same reference or address in memory. To perform a deep comparison on the two objects, the behavior of `equals` must be overridden [39]. As this is not guaranteed for the arbitrary objects we serialize during our experiments, the equality assertions, even for two objects that are otherwise internally identical, are prone to failure if the `equals` method is not, or partially, implemented. This is the case for the target methods of Broadleaf in row 85, and PDFBox in rows 51, 72, and 73 of Table III.

An external component or a cache is not available during testing. Tests can fail due to the unavailability of external components at testing time (while being available in production). For example, the method `getName` in Broadleaf (row 77 in Table III) returns the name of a product as a `java.lang.String` object. When PANKTI collects object profiles in the production environment, the method `getName` returns the name of a product provided through requests to an external translation service. However, during test execution, the method `getName` always returns the default English name of a product because the translation service is not activated. Tests may also fail due to the unavailability of internal caches. The target methods for PDFBox in rows 37, 44, 45, 46, and 50 in Table III access objects from in-memory caches implemented using `java.lang.ref.SoftReference`, which are not available in the testing environment, causing the assertions to fail.

The target methods return objects containing transient fields. The objects being returned by the target methods for PDFBox in rows 33, 34, 35, 53, and 73 of Table III contain transient fields which are not serialized, by definition. Examples of such objects include `java.awt.geom.GeneralPath` and `java.util.IdentityHashMap`. This leads to failing assertions during the comparison of the returned object during the method invocation in the generated test.

The class declaring the target method overrides the serialization behavior. XStream has different converters for different kinds of classes. If a class defines its own serialization behavior, for example, by implementing the `java.io.Externalizable` interface, XStream follows it. If the serialization or the deserialization behavior is not well designed, it can possibly lead to an assertion failure. The tests for method `getBaseRetailPrice` in Broadleaf (row 83 in Table III) fail for this reason. The method returns an object whose type is `Money`. `Money` customizes its deserialization such that a `java.math.BigDecimal` field is converted from a `float` value. However, this changes the scale of the `BigDecimal`, which makes the assertion fail in the generated test.

B. Cases where the Test Suite is not Improved

From Table III, we see that PANKTI generates at least one differential unit test that passes for 15 target methods, yet their status does not upgrade from pseudo-tested to tested. We analyze these cases manually. For those 15 target methods, PANKTI generates valid differential unit tests that check the behavior of these target methods when they return `null`, which is still a weak assertion.

We run Descartes, to determine the status of the target methods after adding the PANKTI-generated tests. For each of these 15 methods, Descartes performs one transformation on the methods: replace the body with `return null;`. The PANKTI-generated tests do not distinguish this transformation from the normal behavior of the method, and thus do not fail. This is a corner case of the default test adequacy criterion of PANKTI based on pseudo-tested methods. PANKTI generates valid, relevant test cases, yet the extreme case where Descartes generates a single variant goes unnoticed.

C. Overhead of Generating Tests from Production Workloads

This section reports a thorough performance evaluation of the deployment overhead of PANKTI for the three study subjects. For PDFBox, we compute the average CPU and memory utilization of 10 normal executions of the workload defined in subsubsection III-B2, 10 executions of this workload which we monitor with Glowroot only, and 10 executions with PANKTI fully attached (Glowroot, together with the instrumentation and serialization). We find that CPU usage is 3.7% during normal execution, and 21% with Glowroot attached. We also observe that attaching PANKTI does not introduce any additional increase in CPU usage. Memory usage is 65 MB during

normal execution, 180 MB while execution of the workload with Glowroot, and 557 MB after attaching PANKTI.

Overhead is an important aspect for interactive client-facing applications such as Jitsi/Jicofo (videoconference) and Broadleaf (e-commerce), because it may degrade user experience. For a Jitsi videoconference, without PANKTI, we observe an average CPU and memory usage of 0.1% and 338 MB, respectively. When attaching PANKTI, we observe 5.5% CPU usage and 902 MB memory. These values drop to 0.4% and 688 MB, respectively, when ignoring the target methods that raise exceptions related to serialization (discussed in subsection IV-B). Packet loss rate remains stable at zero for all experiments, implying that users do not experience any side effects on video or audio quality from lost packets due to PANKTI. For Broadleaf, the average CPU and memory usage during normal execution are 16.4% and 1081 MB, respectively, and 41.8% and 1130 MB with PANKTI attached. To determine the impact on user experience, we measure the average response time for HTTP requests with `apache-utils`[13] and find that it increases by an average of 185 milliseconds. The results from these experiments can also be found at https://github.com/castor-software/pankti-experiments.

PANKTI builds on top of Glowroot, a widely-adopted, state-of-the-art monitoring solution for Java projects. Glowroot constitutes a significant part of the overall PANKTI overhead, yet, it is still acceptable for a standard server running in a real production scenario [40], and this computation price is paid by many applications in the world, in order to access the state-of-the-art monitoring provided by Glowroot. It is also important to note that these values of overhead are subject to change depending on the workload, the set of target methods, and the threshold for object profile collection, all of which are completely configurable factors.

D. Coverage Change with PANKTI

The code coverage ratio of the three case studies discussed in section III slightly increases after adding the tests generated by PANKTI to the original test suites. For Jicofo, the PANKTI differential unit tests cover five new lines of code and six new branches. For PDFBox, the addition of the new tests results in the coverage of four more methods in the application. For Broadleaf, the new tests increase test suite coverage by one line and two branches.

This outcome is to be expected because the test adequacy criterion used in our evaluation is pseudo-tested methods (see subsection III-D). Recall that, by construction, these target methods are already covered by the original test suite of the three projects. For this reason, the only coverage increase may happen in a few uncovered branches in covered methods. To this extent, code coverage can be considered as irrelevant. Instead, we fully focus on the quality of the oracle, and assess the improvement provided by PANKTI tests with regards to the oracle: still pseudo-tested or not, after adding the PANKTI tests. An interesting area of future work would be to study the impact of PANKTI-generated tests on coverage, when the targets for test generation are also selected based on coverage.

[13] https://httpd.apache.org/docs/2.4/programs/ab.html

E. Privacy Implications

As regulations, such as the General Data Protection Regulation (GDPR) in the European Union, gain traction, stricter policies are being enforced to address issues concerning the storage and processing of user data. One technical challenge induced by these regulations consists in finding a good balance between effective monitoring of software systems in production, and secure and lawful handling of user data [41]. PANKTI has privacy implications, since the production workload and object profiles may contain sensitive information, such as user names and passwords. We mitigate a part of this risk with one explicit input that specifies a selection of target methods to be instrumented: the list of target methods can be audited and adjusted by developers first, according to privacy and regulations. In this way, PANKTI would focus its instrumentation and monitoring on non-sensitive methods only. Without involving developers, it is also possible to protect users' privacy with the state-of-the-art of automated privacy-preservation techniques [42], [43]. Overall, similar to related work on production monitoring, PANKTI has to trade-off between its capabilities and compliance with data protection laws [44].

F. Structure and Readability of the PANKTI Differential Unit Tests

Readability is an important aspect of automatically generated tests [45], [46]. A test produced by an automated generation tool is more likely to be seamlessly integrated into an existing test suite if developers can understand it. To this end, we engineer PANKTI to ensure that generated tests are targeted and focused. For our three study subjects, each of the tests generated by PANKTI contains a clear test intention, represented in the form of a single invocation of the target method with a unique input, and a single assertion on equality of the output of the invocation and the oracle. This systematic template and precise intention ensure that these tests are inherently well-structured and easy to understand. The qualitative feedback obtained on the PANKTI test given as pull-request (see subsection IV-C) tends to validate this.

As described in subsection II-E, the tests generated by PANKTI rely on production object profiles which are deserialized from the XML format during the execution of the test. Despite the well-defined structure of the tests themselves, we appreciate that these profiles can often be lengthy and potentially challenging to comprehend. As future work, we plan to conduct user studies involving developers in order to analyze the readability and debuggability [47] of the tests generated by PANKTI.

G. Threats to Validity

The main internal threat to validity comes from the libraries that PANKTI uses for the implementation. In the current version, XStream is used as the serialization and deserialization library. Though XStream is the state-of-the-art tool for object serialization, it sometimes fails for some special types of objects if it is not customized. For example, objects that are based on threads or thread data can not be serialized

by default[14]. Yet, a serialization failure rarely happens for our experiments: we observe this only once in Jicofo. The workaround is engineering: it is possible to improve the ability of XStream's serialization by registering customized object converters.

The threat to external validity is related to the breadth of considered application domains. We mitigate the external threat to validity by evaluating PANKTI with three Java projects, as discussed in section III. The main strength is that they are real-world applications that cover different production workloads and diverse production environments. We look forward to applying PANKTI with our industry partners to further improve external validity.

VI. Related Work

PANKTI contributes to the field of automatic generation and automatic improvement of test suites. The generation of relevant test inputs [48] is a key challenge in both these domains, which has been addressed through symbolic [49], model-based [50], or search-based techniques [51]. The novelty of PANKTI is twofold: first, to collect data in production and automatically turn them into test inputs; second, to target specific parts of the application code that are weakly-tested and of interest for the developers, in order to generate test cases that are valuable for the quality of the test suite.

The closest work to PANKTI is a recently developed tool called Replica, by Wang and Orso [52]. It traces production behavior as a sequence of method calls and uses these traces to spot behaviors triggered in production but not covered by the test suite. Replica then uses a guided symbolic execution of the program to "mimic" this behavior and to generate inputs for untested behavior. The primary difference with PANKTI is that we generate test oracles from the actual observations made in production, and not from symbolic execution. PANKTI also collects a different type of trace compared to Replica: object profiles instead of sequences of method calls. Focusing on object profiles, we can generate test data that recreate production conditions.

A. Generating tests from execution traces

Thummalapenta *et al.* [53] mine program execution traces, which include method calls, arguments, and return values, to generate parameterized unit tests implemented as regression tests with Pex [54]. Marchetto *et al.* [55] define a methodology to generate Selenium tests from event logs for web applications. Sampath *et al.* [56] apply incremental concept analysis to cluster similar test cases, generated from user-session based testing of web applications. Several works generate test cases that reproduce failures [57], [58], [29]. In particular, Artzi *et al.* propose ReCrash [15] to turn runtime observations into unit tests that reproduce failures [29]. ReCrash works in a similar fashion as PANKTI in that it serializes object states observed during execution. However, unlike PANKTI, its goal is not to generate tests for target methods that need better testing, but to generate tests that raise the same exceptions as observed in production. In particular, ReCrash has no feature to create derived oracles. More recently, Utting *et al.* [59] apply machine learning to user and test execution traces in order to identify test inadequacies and generate new tests for usage scenarios that are missing from the test suite. Křikava and Vitek [26] record function calls, including arguments, in execution traces of software packages developed in the R language, and extract unit tests from them with the goal of improving coverage. Similar to these studies, PANKTI relies on the observation of an executing application for test generation. Compared to these tools, the key novelty of PANKTI is the execution representation using object profiles, which has never been proposed so far. Moreover, PANKTI allows developers to choose which methods to target for test generation.

B. Test suite improvement

Danglot *et al.* [38] and Baudry *et al.* [60] propose to transform existing test methods into new ones that globally improve the coverage and the mutation score of the test suite [61]. Tillman and Schulte [62] suggest the use of symbolic execution to generalize traditional unit tests into parameterized unit tests and instantiating these PUTs to obtain concrete tests for higher test coverage. Saff *et al.* [36] develop a test refactoring technique to automatically generate unit tests from bulky system tests. Harder *et al.* [8] compute the operational difference based on the behavior of a program observed during execution in semantic terms with Daikon [63] to keep only those test cases that add to the abstraction. This results in a test suite that is minimal but better at detecting faults than a suite with high coverage. Chen *et al.* [64] address the issues related to behavioral backward incompatibilities (BBIs), and propose the early detection of BBIs in libraries across their client projects by prioritizing and executing the clients' tests. The goal of these studies, like ours, is to improve the test quality of the project. The key differences are that PANKTI bases its test input and derived oracle generation process on observations collected in production, and that it targets test generation on a subset of methods that the developers consider as relevant for test improvement.

C. Handling the oracle problem

The oracle problem in software testing refers to the identification of the desired behavior of a program unit [65]. In their survey, Barr *et al.* [2] investigate four aspects in the literature that address the oracle problem, namely specified, derived, and implicit oracles, and the lack of automated test oracles. Gay *et al.* [66] combine monitoring the execution of test cases, with mutation testing, in order to automatically select oracle data. Bertolino *et al.* [67] execute tests in-vivo, as an application executes, effectively leveraging production data as oracles and allowing the discovery of corner-cases that may otherwise be rare in the testing environment. The oracle can be automated, taking the domain into account. For instance, the ADVISOR tool of Genç *et al.* [68] automates the test oracle in systems with a graphical/visual output, which can otherwise be very inaccurate due to contextual differences between an

[14] https://x-stream.github.io/faq.html#Serialization_types
[15] http://groups.csail.mit.edu/pag/reCrash/

output image and a reference image. PANKTI approaches the oracle problem by automatically synthesizing derived oracles from production observations, which is novel to the best our knowledge.

Our work is different from the technique of Elbaum *et al.* [14], who also generate derived oracles: they use system tests whereas PANKTI uses production workloads; their approach does not allow the set of target methods to be configured according to a developer-defined criterion. Moreover, PANKTI can generate tests for methods that are never tested, while the technique of Elbaum *et al.* generates test cases exclusively for methods that are covered by system-level tests

D. Capture and replay

Joshi and Orso [69] present a capture and replay tool, and propose potential applications, including the generation of regression tests. Their tool captures selectively at the boundary of a subset of the application under study as it executes. Steven *et al.* [70] design a tool called JRapture that captures the sequence of interactions between an executing Java program and components on the host system such as files, or events on graphical user interfaces. These sequences can then be replayed for observation-based testing. GenUTest [28] is a capture and replay tool which logs method calls and the sequence of their occurrence, in a medium-sized, executing instrumented program. The arguments passed to and values returned from methods are serialized into logs, which are utilized for the generation of unit tests and mocks for methods, including test assertions. Saieva *et al.* [71] generate ad-hoc tests that replay recorded user execution traces in order to test candidate patches for critical security bugs. The uniqueness of PANKTI is that object profiles are captured not for the purpose of replaying the same sequence of operations on the system, but for specifying the behavior of target methods, through focused differential unit tests, with derived test oracles. This is in contrast to capture and replay systems, which typically lack oracles.

VII. Conclusion

This paper has introduced PANKTI, a tool that observes Java programs in production to automatically generate differential unit tests. PANKTI observes specific methods that are weakly-tested according to a test adequacy criterion and introduces a novel technique based on the collection of object profiles. We have conducted experiments with three sizeable, popular, multi-domain, open-source Java projects to assess PANKTI's ability at monitoring production and at improving real-world test suites. PANKTI successfully generates differential unit tests that improve the testing of 53 of the 86 (61.6%) methods targeted across our three study subjects. This shows that PANKTI is able to generate tests for real-world Java software.

In follow-up studies, we will broaden the scope of test improvement to other weakly-tested parts. In particular, one can focus on methods that are not reached by any existing test but executed in the field, the existence of which has been demonstrated by Wang *et al.* [9] and Gittens *et al.* [72]. Also, we wish to explore test input minimization by utilizing partial object profiles instead of the whole. Our second thread for future work consists in extending PANKTI to generate integration- or system-level tests, by considering arbitrary sequences of method invocations across different classes.

References

[1] T. Y. Chen and R. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 3, pp. 1–27, 2008.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[3] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "How the experience of development teams relates to assertion density of test classes," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, pp. 223–234, IEEE, 2019.

[4] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1213–1228, 2017.

[5] G. Grano, C. De Iaco, F. Palomba, and H. C. Gall, "Pizza versus pinsa: On the perception and measurability of unit test code quality," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, pp. 336–347, 2020.

[6] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, 2019.

[7] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *European Conference on Object-Oriented Programming*, pp. 380–403, Springer, 2006.

[8] M. Harder, J. Mellen, and M. D. Ernst, "Improving test suites via operational abstraction," in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 60–71, IEEE, 2003.

[9] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 321–332, IEEE, 2017.

[10] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 302–313, 2013.

[11] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proc. of the International Conference on Automated Software Engineering, ASE*, pp. 53–63, 2018.

[12] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pp. 163–171, 2018.

[13] Y. Ma, J. Offutt, and Y. R. Kwon, "Mujava: a mutation system for java," in *Proc. of the International Conference on Software Engineering (ICSE)*, pp. 827–830, 2006.

[14] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 253–264, 2006.

[15] R. Niedermayr, E. Juergens, and S. Wagner, "Will my tests tell me if i break this code?," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, pp. 23–29, IEEE, 2016.

[16] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th international conference on software engineering*, pp. 435–445, 2014.

[17] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 214–224, 2015.

[18] L. Zhang and M. Monperrus, "Tripleagent: Monitoring, perturbation and failure-obliviousness for automated resilience improvement in java applications," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, 2019.

[19] B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry, "An approach and benchmark to detect behavioral changes of commits in continuous integration," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2379–2415, 2020.

[20] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 237–249, IEEE, 2020.

[21] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1195–1225, 2019.

[22] S. Butler, J. Gamalielsson, B. Lundell, C. Brax, A. Mattsson, T. Gustavsson, J. Feist, and E. Lönroth, "Maintaining interoperability in open source software: A case study of the apache pdfbox project," *Journal of Systems and Software*, vol. 159, p. 110452, 2020.

[23] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 90–99, 2011.

[24] E. Costlow, "The State of Java Serialization." https://www.infoq.com/articles/java-serialization-aug18/ (accessed 2020-10-14).

[25] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pp. 177–182, 2012.

[26] F. Křikava and J. Vitek, "Tests from traces: automated unit test extraction for r," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 232–241, 2018.

[27] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 433–444, IEEE, 2018.

[28] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "Genutest: a unit test and mock aspect generation tool," *International journal on software tools for technology transfer*, vol. 11, no. 4, p. 273, 2009.

[29] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *European conference on object-oriented programming*, pp. 542–565, Springer, 2008.

[30] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. update for oadoi on Nov 02 2018.

[31] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative inquiry*, vol. 12, no. 2, pp. 219–245, 2006.

[32] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[33] T. Kuchta, T. Lutellier, E. Wong, L. Tan, and C. Cadar, "On the correctness of electronic documents: studying, finding, and localizing inconsistency bugs in pdf readers and files," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3187–3220, 2018.

[34] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. Briand, "Log-based slicing for system-level test cases," in *2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[35] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 international symposium on software testing and analysis*, pp. 385–396, 2015.

[36] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 114–123, 2005.

[37] M. Jorde, S. Elbaum, and M. B. Dwyer, "Increasing test granularity by aggregating unit tests," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 9–18, 2008.

[38] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with dspot: a study with ten mature open-source projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2603–2635, 2019.

[39] B. Weidig, "Equality and Comparison in Java: Pitfalls and Best Practices." Link to Medium post (accessed 2020-11-19).

[40] O. Cornejo, D. Briola, D. Micucci, and L. Mariani, "In-the-field monitoring of functional calls: Is it feasible?," *Journal of Systems and Software*, vol. 163, p. 110523, 2020.

[41] F. Menges, T. Latzo, M. Vielberth, S. Sobola, H. C. Pöhls, B. Taubmann, J. Köstler, A. Puchta, F. Freiling, H. P. Reiser, *et al.*, "Towards gdpr-compliant data processing in modern siem systems," *Computers & Security*, vol. 103, p. 102165, 2021.

[42] H. Zhang, S. Latif, R. Bassily, and A. Rountev, "Differentially-private control-flow node coverage for software usage analysis," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (S. Capkun and F. Roesner, eds.), pp. 1021–1038, USENIX Association, 2020.

[43] T. Pulls, R. Peeters, and K. Wouters, "Distributed privacy-preserving transparency logging," in *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, (New York, NY, USA), p. 83–94, Association for Computing Machinery, 2013.

[44] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram, "Analyzing the impact of {GDPR} on storage systems," in *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[45] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pp. 5–14, 2016.

[46] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 107–118, 2015.

[47] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 151–158, 2004.

[48] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[49] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[50] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[51] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163, IEEE, 2011.

[52] Q. Wang and A. Orso, "Improving testing by mimicking user behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 488–498, IEEE, 2020.

[53] S. Thummalapenta, J. De Halleux, N. Tillmann, and S. Wadsworth, "Dygen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *International Conference on Tests and Proofs*, pp. 77–93, Springer, 2010.

[54] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*, pp. 134–153, Springer, 2008.

[55] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 121–130, IEEE, 2008.

[56] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald, "Applying concept analysis to user-session-based testing of web applications," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 643–658, 2007.

[57] P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, and A. van Deursen, "Search-based crash reproduction using behavioural model seeding," *Softw. Test. Verification Reliab.*, vol. 30, no. 3, 2020.

[58] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: let existing test cases help," in *Proc. of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pp. 910–913, ACM, 2015.

[59] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, and F. Bouquet, "Identifying and generating missing tests using machine learning on execution traces," in *Proc. of the International Conference On Artificial Intelligence Testing*, pp. 83–90, 2020.

[60] B. Baudry, F. Fleurey, J. Jézéquel, and Y. L. Traon, "Genes and bacteria for automatic test cases optimization in the .net environment," in *Proc. of the Int. Symposium on Software Reliability Engineering (ISSRE)*, pp. 195–206, 2002.

[61] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[62] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," *IEEE software*, vol. 23, no. 4, pp. 38–47, 2006.

[63] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[64] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 112–124, 2020.

[65] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *Proc. of the International Conference on Software Engineering (ICSE)*, pp. 391–400, 2011.

[66] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Automated oracle data selection support," *IEEE Trans. Software Eng.*, vol. 41, no. 11, pp. 1119–1137, 2015.

[67] A. Bertolino, G. De Angelis, B. Miranda, and P. Tonella, "Run java applications and test them in-vivo meantime," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 454–459, IEEE, 2020.

[68] A. E. Genç, H. Sözer, M. F. Kıraç, and B. Aktemur, "Advisor: An adjustable framework for test oracle automation of visual output systems," *IEEE Transactions on Reliability*, 2019.

[69] S. Joshi and A. Orso, "Scarpe: A technique and tool for selective capture and replay of program executions," in *2007 IEEE International Conference on Software Maintenance*, pp. 234–243, IEEE, 2007.

[70] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jrapture: A capture/replay tool for observation-based testing," in *Proc. of ISSTA*, pp. 158–167, 2000.

[71] A. Saieva, S. Singh, and G. Kaiser, "Ad hoc test generation through binary rewriting," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 115–126, IEEE, 2020.

[72] M. Gittens, K. Romanufa, D. Godwin, and J. Racicot, "All code coverage is not created equal: a case study in prioritized code coverage," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, pp. 11–es, 2006.

Harvesting Production GraphQL Queries to Detect Schema Faults

Louise Zetterlund[†], Deepika Tiwari[*], Martin Monperrus[*], and Benoit Baudry[*]

[†]Redeye AB, Sweden
[*]KTH Royal Institute of Technology, Sweden

***Abstract*—GraphQL is a new paradigm to design web APIs. Despite its growing popularity, there are few techniques to verify the implementation of a GraphQL API. We present a new testing approach based on GraphQL queries that are logged while users interact with an application in production. Our core motivation is that production queries capture real usages of the application, and are known to trigger behavior that may not be tested by developers. For each logged query, a test is generated to assert the validity of the GraphQL response with respect to the schema. We implement our approach in a tool called AutoGraphQL, and evaluate it on two real-world case studies that are diverse in their domain and technology stack: an open-source e-commerce application implemented in Python called Saleor, and an industrial case study which is a PHP-based finance website called Frontapp. AutoGraphQL successfully generates test cases for the two applications. The generated tests cover 26.9% of the Saleor schema, including parts of the API not exercised by the original test suite, as well as 48.7% of the Frontapp schema, detecting 8 schema faults, thanks to production queries.**

***Index Terms*—GraphQL, production monitoring, automated test generation, test oracle, API testing, schema**

I. Introduction

Web APIs consist of programmable endpoints to interact with software systems. They can be implemented in different ways, including the well known REST [1] and SOAP [2], [3] paradigms. GraphQL is a new way to define web APIs invented by Facebook in 2015 [4]. A GraphQL API implementation consists of a schema that specifies the data structures and operations exposed by the API, as well as a server that implements the logic to handle API requests, resolving them into actual data. The clients of a GraphQL API, typically a browser or an app, send requests that specify the data they want to retrieve. The performance [5], [6] and flexibility [7] of GraphQL have contributed to its rapid adoption in the industry [8], [9].

While GraphQL offers significant benefits to develop web APIs, correctly implementing it remains a challenge. Specifically, a bug in the server may cause a GraphQL query to be resolved into data that is incompatible with the properties defined in the schema. We designate this kind of fault as a schema fault. Let us consider the example of *Saleor*, an e-commerce platform that exposes a GraphQL API. A user reported an error when trying to create a new product without assigning it to a category[1]. This issue was identified as a schema fault and fixed by the maintainers, because the API implementation contradicted the *Saleor* GraphQL schema. Schema faults are the focus of testing techniques for other systems specified using schemas, such as databases [10], [11], or OpenAPI REST APIs [12]–[14]. However, there has been little work to detect schema faults in GraphQL APIs. Only one approach, by Karlsson *et al.* [15] targets them, by generating GraphQL queries randomly and using them as inputs in property-based tests with the goal of exercising more of the GraphQL schema.

In this work, we propose to harvest GraphQL queries from an application in production, and use them as inputs for test generation. Our motivation is that test cases generated from production GraphQL queries assess the behavior of the application with respect to real API usages. Moreover, it has been shown that production data can lead to valuable test cases that invoke behavior untested by developer-written tests [16], [17]. We implement our technique in a tool called AutoGraphQL, which operates in two phases. The first phase involves monitoring an application in production and logging every unique GraphQL query. In the second phase, AutoGraphQL generates one test for each query logged in the monitoring phase. Each generated test includes the required oracles to assess whether the format of the response is consistent with the GraphQL schema.

We evaluate AutoGraphQL on one open-source and one closed-source case study. Our open-source case study is an e-commerce platform called *Saleor*. Our closed-source industrial case study, *Frontapp*, is the primary website of Redeye AB, an equity research and investment banking company based in Stockholm, Sweden. AutoGraphQL harvests 334 and 24,049 unique GraphQL queries in production, for Saleor and Frontapp, respectively. Our tool successfully generates one test for each logged query. The generated tests exercise 26.9% of the GraphQL schema in Saleor, including parts of the schema not exercised by the original, developer-written test suite. The tests generated by AutoGraphQL for Frontapp exercise 48.7% of the schema and detect 8 schema faults.

Our evaluation of AutoGraphQL with two diverse case studies demonstrates that it can successfully generate tests with GraphQL queries harvested from production. The generated tests complement developer-written tests by triggering untested behavior, and are able to discover schema faults in the implementation of the GraphQL API. To sum up, our

[1] https://github.com/mirumee/saleor/issues/5589

contributions are as follows:

- A novel technique to harvest GraphQL queries from production and use them to generate test cases with oracles tailored for the detection of schema faults
- The evaluation of our technique with one industrial and one open-source case study, which demonstrates that the generated tests trigger untested behavior, and discover schema faults
- An open-source implementation of our methodology in a tool called AutoGraphQL, as well as a publicly available dataset for reproducibility at https://github.com/castor-software/autographql/

The rest of this paper is organized as follows: section II discusses GraphQL, section III introduces AutoGraphQL, and section IV describes the methodology we use to evaluate it. We present the results from this evaluation in section V, and discuss some aspects of AutoGraphQL in section VI. section VII includes related work and section VIII concludes the paper.

II. Background

This section introduces GraphQL, a specification for web APIs, as well as its implementation.

A. GraphQL APIs

GraphQL is a new paradigm to build web APIs. A GraphQL API consists of one schema that defines the data structures that are available through the API, and a set of requests, or 'queries', that can be made against the schema. The implementation of the API is composed of so-called 'resolvers' which map the information requested by the queries to actual data from the underlying database or storage of the application.

GraphQL requests are typically triggered from clients such as the frontend of an application, and are handled by a server at the backend. Unlike REST APIs [7], [18], the requests are not centered around resources [1]. Instead, they are structured around operations. There are two types of requests in GraphQL: queries and mutations, defined as follows. A request that only fetches data, such as the details of a product on a website, is called a `Query`; a request that changes, or "mutates" data, such as adding or updating a shipping address, is called a `Mutation`. Both kinds of requests are sent to a GraphQL endpoint exposed by the application backend, where they are resolved. The corresponding responses are sent back to the frontend, typically as JSON.

B. GraphQL Schemas

A GraphQL schema serves as a contract between the frontend and the backend of the application [19]. A schema is specified with the strongly-typed Schema Definition Language (SDL), which is defined in the GraphQL specification[2]. It includes declarations of object, enum, interface, and union types. The types themselves are composed of fields that define their properties. These fields may be a scalar, such as `Int`, `String`, or `ID`, other object types defined in the schema, or an array thereof. An exclamation mark `!` represents non-nullable fields. In addition to the type declarations, the schema also defines the `Query` and `Mutation` operations that can be performed on them.

```
interface Node {
  id: ID!
}

enum VideoTypeEnum {
  ANIMATION
  LIVE_ACTION
  SCREENCAST
}

type Video implements Node {
  id: ID!
  title: String!
  url: String!
  videoType: VideoTypeEnum
  teaser: Teaser
}

type Teaser {
  title: String!
  subTitle: String
  publishedOnSite: Boolean
  url: String!
  duration: Float
}

type Query {
  video(id: ID!): Video
  teasers(first: Int!):
       [Teaser]
}
```

Listing 1: An excerpt from the GraphQL schema of Frontapp

Listing 1 is an excerpt from the GraphQL schema of Frontapp, the primary website of a company called Redeye AB[3], employing one of the authors. The schema defines an `interface` called `Node` which has a non-nullable `id` of type `ID`. The schema also defines two object types. The `Video` type represents a video published on Frontapp. It implements `Node` and it has non-nullable `String` fields that specify its `title` and its `url`. The field `videoType` expresses the kind of a video as one of the values enlisted in the enumeration type `VideoTypeEnum`. A video may also have a `teaser` of type `Teaser`, which itself has non-nullable `String` fields for its `title` and `url`, and a nullable `subTitle`. A teaser also has a `duration` expressed as a `Float`. The `Boolean` field `publishedOnSite` determines if the teaser has been published on Frontapp.

`Query` is a special GraphQL type that defines the entry-points of all GraphQL queries that fetch data. This excerpt of the Frontapp schema defines two entry-points, `video` and `teasers`. A `Video` object can be fetched through a non-nullable `id` argument, through the `video` entry-point. The `teasers` entry-point returns a list of the first *n* `Teaser`-type objects, based on the value of *n* provided as the non-nullable `Int` argument to the variable `first`.

GraphQL allows the requesting entity to explicitly specify, in a single declarative GraphQL query [20], the data or fields required in the response. Listing 2 shows a query made against the schema defined in Listing 1. The query is given an explicit name, called its *operation name*, which is `GetTeasers`. This query is generated by the interactions of end-users as they browse through the videos published on the Frontapp website. This interaction would trigger the `teasers` entry-point and fetch a list of objects of type `Teaser`. The query requests for the `title`, `subTitle`, and `url` of the first `2` teasers. The

[2] https://spec.graphql.org

[3] https://www.redeye.se

```
query GetTeasers {
  teasers(first: 2) {
    title
    subTitle
    url
    __typename
  }
}
```

Listing 2: A production GraphQL query made against the schema defined in Listing 1

```
{
  "data": {
    "teasers": [
      {
        "title": "Finance 101",
        "subTitle": "The basics of finance",
        "url": "https://youtu.be/dQw4w9WgXcQ",
        "__typename": "Teaser"
      },
      {
        "title": "Development 101",
        "subTitle": null,
        "url": "https://youtu.be/jNQXAC9IVRw",
        "__typename": "Teaser"
      }
    ]
  }
}
```

Listing 3: The response for the query in Listing 2

meta-field `__typename`, wherever used in a query, specifies the type of the object at that point in the query.

C. GraphQL Resolvers

Resolvers are functions to map each field requested in incoming queries with actual data in the application. The resolvers are not written in GraphQL, they may be implemented in any programming language supported by the underlying GraphQL engine, including Java, JavaScript, PHP, Python, and others[4]. Therefore, GraphQL API implementations can evolve [7] while providing stable APIs.

Listing 4 shows a resolver for Frontapp, written in PHP. The resolver fetches the first *n* `Teaser` objects, from the `teaserRepository`, which is the component that interacts with the Frontapp database. The resolver then prepares the response, with values for all the fields requested by the query in Listing 2 fetched from the database. The response is a list of teaser objects with their `title`, `subTitle`, and `url`. We present the response in Listing 3. It contains only the fields explicitly requested in the query, for the `first` 2 `teasers`, including their `title`, `subtitle`, and `url`, and with their `__typename` being `Teaser`.

III. AutoGraphQL

This section describes AutoGraphQL, a tool that automatically generates tests for the GraphQL backend of an application. We first discuss schema faults, which are the targets for the tests generated by AutoGraphQL. Then, we present an overview of AutoGraphQL, and describe the phases

[4]https://graphql.org/code/#language-support

```
1 @@ -0 +2 @@
2 public function resolveTeasers(int $first) {
3   $teasers = $this->teaserRepository->findMatching($first);
4   $data = [];
5   foreach ($teasers as $teaser) {
6     $newTeaser = new \stdClass();
7     $newTeaser->title = $teaser->getTitle();
8     $newTeaser->subTitle = $teaser->getSubTitle();
9 +   if ($teaser->isPublishedOnSite()) {
10      $newTeaser->url = $teaser->getUrl();
11 +   }
12    $data[] = ["teaser" => $newTeaser];
13  }
14  return (array_column($data, "teaser"));
15 }
```

Listing 4: A resolver that fetches a list of the first *n* `teasers`, with a schema fault that has just been introduced

in which it operates. We conclude this section by discussing the implementation of the tool.

A. Schema Faults

In this work, we aim at detecting faults in the implementation of the GraphQL resolvers, leading them to return data with a format that does not conform to the schema. We call these faults "schema faults". They occur in GraphQL APIs when valid queries get resolved into invalid responses, as a result of incorrect mapping between the fields requested and the actual data storage in the application. This response may be sent to the client without the error being explicitly identified as such (say HTTP 5xx or a JSON error object). Such invalid responses basically break the interface contract between the server and the client as specified in the schema. This kind of fault is common, for example, a user of the e-commerce platform Saleor reported an issue[5], confirmed by a developer, due to a schema fault.

In order to detect schema faults, AutoGraphQL automatically generates test oracles, derived from the schema. These test oracles determine that the data returned by the API is well-formed, with respect to the schema. For example, the `Boolean` field `publishedOnSite` for a `Teaser` is defined as nullable in the schema in Listing 1. If the condition on line 9 is introduced in the resolver presented in Listing 4, the `url` of the teaser object will be resolved to null if `publishedOnSite` is false or null. This contradicts the schema which specifies that the `url` of a `teaser` cannot be null, and is therefore a bug in the implementation of the resolver. This kind of fault would be detected by AutoGraphQL.

B. Overview of AutoGraphQL

AutoGraphQL generates tests that (i) exercise the GraphQL API implementation, and (ii) assess that the data returned as a resolution to a GraphQL query conforms to the schema. AutoGraphQL operates in two phases, illustrated in Figure 1. The first phase consists in monitoring the application in production, in order to collect the queries that are performed by users, as well as their arguments. This data collection process is performed for a given amount of time, decided by

[5]https://github.com/mirumee/saleor/issues/6750

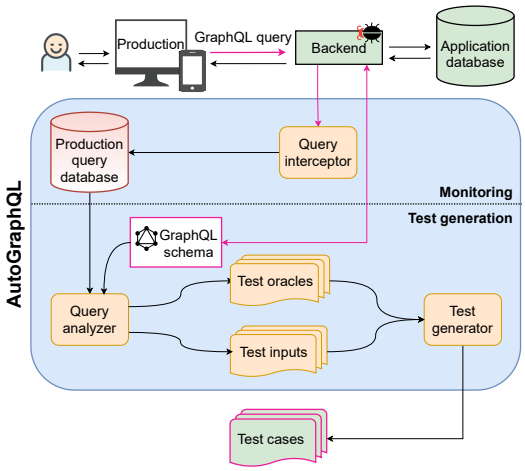


Fig. 1: Overview of AutoGraphQL

developers. The second phase of AutoGraphQL is triggered by developers and consists in analyzing the data that was observed in production to turn them into test cases. AutoGraphQL automatically extracts test inputs from the monitored data, as well as the corresponding oracles from the GraphQL schema. The tests generated by AutoGraphQL use a single GraphQL query as the test input and verify the format of the response to the query. The following two subsections discuss the details of each phase of test generation with AutoGraphQL.

C. Monitoring in Production

Monitoring GraphQL queries in production constitutes the first phase of AutoGraphQL, and is illustrated in the top-half of Figure 1. AutoGraphQL's *query interceptor* monitors the requests sent from the frontend of an application to its backend. It intercepts incoming GraphQL query requests, and the arguments with which they were invoked, and logs them into a database. It also aggregates metadata about these queries, including the number of times a specific query request was invoked or when it was last invoked. The output from this phase is a database of GraphQL queries logged from production.

Listing 5 presents an example of a logged query. The keys `query` and `variables` represent the actual query executed as well as the argument passed to it, respectively. In this case, the query is the same as in Listing 2, and the value of `2` is passed as argument for `first`, to fetch the first 2 `teasers`. The entry also includes the operation name for the query (`operation_name`), which is `GetTeasers`. In order to gather statistics about the queries triggered in production, we also save timestamps for when they are first logged (`created_at`) and when they are logged most recently (`updated_at`). Moreover, during our experiments, we observe that a query may frequently be invoked with

```
{
  "query":
    "query GetTeasers($first: Int!) {
      teasers(first: $first) {
        title
        subTitle
        url
        __typename
    }
  }",
  "variables": {
    "first": 2
  },
  "operation_name": "GetTeasers",
  "created_at": "2021-03-03 08:57:46",
  "updated_at": "2021-05-05 16:55:19",
  "times_called": 301016
}
```

Listing 5: A logged GraphQL query from production

the same arguments. We create one entry for each unique combination of query and arguments, and record the frequency of its occurrence with the `times_called` field. The value of $301,016$ means that this combination of query and argument occurred as many times, in production, during the course of our experiment.

D. Test Generation

The second phase of AutoGraphQL, presented in the bottom-half of Figure 1, is triggered by developers whenever they want to generate tests after a period of monitoring. It involves automatically fetching the GraphQL schema of the application from the configured GraphQL endpoint, and using this schema in conjunction with the queries logged in the monitoring phase, in order to produce the inputs and oracles for the generated tests. The output from this phase is the test suite generated from the logged production queries. We now discuss the two components that are responsible for analyzing the logged queries and using them to generate tests.

1) Query Analyzer: The goal of the query analyzer is to use the GraphQL schema of an application and the queries logged in the query database to generate test inputs and their corresponding oracles. Given a logged GraphQL query, such as the one in Listing 5, the analyzer extracts the test input, which is the combination of `query` and its associated `variables`, as well as the `operation_name` given to the query. Next, the query analyzer produces two sets of oracles. We summarize them in Table I, together with their implementation as assertions in the PHPUnit framework[6]. The first set of oracles verify the format of the response, specifically i) its HTTP status code, ii) the validity of the JSON text, and iii) that it does not contain a JSON error object because of an invalid request. The second set of oracles are specific to the schema and i) verify that the response contains all the data requested by the query, and ii) map each object and field requested by the query with the properties defined for it in the schema. These properties include its type, its kind, i.e., whether it is an object, enum, list, or interface, and its nullability.

[6]https://phpunit.de

TABLE I: The oracles generated by AutoGraphQL depending on the response

CATEGORY	ORACLE	IMPLEMENTATION AS PHPUnit ASSERTION
Format	HTTP status code of the response is 200	`assertEquals(200, ...)`
	Response to query is well-formed, valid JSON	`json_decode` doesn't throw exceptions; `assertIsArray(...)`
	Response does not contain a JSON error object	`assertArrayNotHasKey('errors', ...)`
Schema	Response contains all requested fields	`assertArrayHasKey(...)`
	Correct kind of each element in response	`assertIsArray(...)` `assertContains(...)` `assertEquals(...)`
	Correct type of each element in response	`assertIsString(...)` `assertIsBool(...)` `assertIsNumeric(...)` `assertIsInt(...)` `assertEquals(...)`
	Nullability-contract of each field in response	`assertNotNull(...)`

For example, a subset of the oracles produced for the query in Listing 5 is that the response would contain a field called `title` (`assertArrayHasKey('title', ...)`), which is a non-null (`assertNotNull(...)`) `String` type object (`assertIsString(...)`).

2) Test Generator: The test generator uses the test input and test oracles produced by the query analyzer in order to generate a valid and executable test case that verifies the implementation of the GraphQL API for the application. The output of the test generator is one test case for each logged query. By default, AutoGraphQL generates tests in PHP, using the PHPUnit framework as test driver.

Listing 6 shows the test generated for the query in Listing 5. The test fetches the response to the query (lines 10 to 23) by sending it as an HTTP request to the GraphQL endpoint of an application (lines 25, 26). After verifying its HTTP status code (line 28), the test decodes the response and verifies that it is well-formed JSON (lines 30, 31). Next, the assertion on line 33 ensures that the response does not contain an error due to an invalid query, due to the query trying to fetch data that does not exist, or an exception being raised during query resolution. Lines 36 to 51 contain the assertions produced by the test generator using the oracles derived from the schema. We use the assertion available in PHPUnit to check the validity of collections: `assertIsArray` verifies that `teasers` is a list. Next, `for` each of the items within `teasers`, `assertEquals` verifies that its `__typename` is `Teaser`. `assertArrayHasKey` assertions verify that each of the `teaser` objects in the list has a `title`, a `subTitle`, and a `url`, as requested by the query. The assertions for the `title` and the `url` of a `Teaser` are `assertNotNull` since they are defined as non-nullable, per the schema in Listing 2. Moreover, `assertIsString` verifies that the `title` and `url`, and `if` present, the `subTitle` of a `teaser` are all strings. When this test is executed, assuming the server returns the response shown in Listing 3, 22 assertions are evaluated in total. A failure in any of the assertions in a generated test causes the test to fail, which could be indicative of a schema fault. On the other hand, a generated test that passes can serve as a regression test.

E. Challenges

We now discuss the challenges of test generation with AutoGraphQL.

Query Interception: In order for the query interceptor of AutoGraphQL to monitor and log incoming GraphQL queries, it must be tailored to fit the technology stack of an application. For example, the configuration of the query interceptor used by an application with a backend implemented in Python would differ from the one implemented in Ruby. This is a potentially significant engineering effort.

Testing Database: The execution of the generated test suite requires a running application server with a testing database. This is typically provided by a staging environment. The state of the staging database may have an impact on the execution of the generated tests. Thus an important engineering challenge is to be able to re-initialize a clean staging database before running the AutoGraphQL tests.

F. Implementation

AutoGraphQL is implemented in Python. The query analyzer uses the GraphQLParser of graphql-py[7] to map the elements of a query with the schema and produce test oracles. By default, AutoGraphQL populates a template with the test input and oracles in order to produce tests in the PHPUnit framework. This allows the properties of each node to be expressed as PHPUnit assertions, which serve as oracles in the generated test. Jinja2[8] is the templating language used to render the assertions into PHPUnit test files. We choose PHPUnit for the generated tests, since PHP is a popular server-side language for the implementation of web APIs [21]. AutoGraphQL can generate tests for applications that do not use PHP, or even be extended to support any testing framework, since the generated tests only interact with the HTTP GraphQL endpoint of the application.

IV. Evaluation Methodology

This section describes our two real-world case studies, *Saleor* and *Frontapp*. We also present our experimental setup, including the configuration of AutoGraphQL with the two case studies, their production workloads, as well as the metrics used to evaluate the effectiveness of AutoGraphQL in generating tests for them.

[7] https://github.com/ivelum/graphql-py
[8] https://jinja.palletsprojects.com/en/2.11.x/

```php
<?php declare(strict_types=1);

namespace GraphQL;
use PHPUnit\Framework\TestCase;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use Symfony\Component\HttpFoundation\Request;

class GetTeasersTest extends WebTestCase {
  public function testGraphQL() {
    $client = static::createClient();

    /* Use the details from the logged query */
    $query = <<<'JSON'
    {
      "query": "query GetTeasers($first: Int!) {
       teasers(first: $first) {
         title
         subTitle
         url
         __typename
       }
      }",
      "variables": { "first": 2 },
      "operationName": "GetTeasers"
    }
    JSON;

    /* Make an HTTP request with the query */
    $client->request('POST', '/graphql/', [], [],
        ["CONTENT_TYPE" => 'application/json'], $query);
    $response = $client->getResponse();

    /* Verify the HTTP status code of the response */
    $this->assertEquals(200, $response->getStatusCode());

    /* Decode the response and verify that it contains valid
        JSON */
    $responseArray = json_decode($response->getContent(),
        true);
    $this->assertIsArray($responseArray, 'Response is not
        valid JSON');

    /* Verify that the response does not have errors */
    $this->assertArrayNotHasKey('errors', $responseArray,
        'Response contains errors');
    $responseContent = $responseArray['data'];

    /* Verify the properties of the response payload per the
        schema */
    $this->assertArrayHasKey('teasers', $responseContent);
    if ($responseContent['teasers']) {
      $this->assertIsArray($responseContent['teasers']);
      for($i = 0; $i < count($responseContent['teasers']);
          $i++) {
        if ($responseContent['teasers'][$i]) {
          $this->assertEquals('Teaser' ,
              $responseContent['teasers'][$i]['__typename']);
          $this->assertArrayHasKey('title',
              $responseContent['teasers'][$i]);
          $this->assertNotNull(
              $responseContent['teasers'][$i]['title']);
          $this->assertIsString(
              $responseContent['teasers'][$i]['title']);
          $this->assertArrayHasKey('subTitle',
              $responseContent['teasers'][$i]);
          if ($responseContent['teasers'][$i]['subTitle']) {
            $this->assertIsString(
                $responseContent['teasers'][$i]['subTitle']);
          }
          $this->assertArrayHasKey('url',
              $responseContent['teasers'][$i]);
          $this->assertNotNull(
              $responseContent['teasers'][$i]['url']);
          $this->assertIsString(
              $responseContent['teasers'][$i]['url']);
        }
      }
    }
  }
}
```

Listing 6: A PHPUnit test generated using the logged query in Listing 5, based on the schema in Listing 1

TABLE II: Case studies for the evaluation of AutoGraphQL

PROJECT	LOC	COMMITS	LANGUAGE	DOMAIN
Saleor	691K	17.4K	Python	E-commerce
Frontapp	154K	7K	PHP	Finance

A. Case Studies

We use one open-source and one industrial project as case studies in order to evaluate the effectiveness of AutoGraphQL. We describe these projects and some relevant metrics below.

1) Saleor: Saleor is a widely-used, open-source e-commerce platform, maintained by more than 170 contributors. The project has more than $13K$ stars on GitHub. It is a well-documented and mature project, that can be deployed with Docker. Saleor is crafted with modern technologies, such as Django[9], PostgreSQL, Redis, React, and TypeScript. It offers both a storefront for customers to browse through a catalog of products and make purchases, as well as a dashboard for administrators to manage products, users, and orders. Incoming requests from both of these frontends are handled by a GraphQL server implemented as part of the core component of Saleor. The test suite of Saleor contains automated tests for both the backend and the two frontend components. We choose Saleor as a case study in order to have reproducible experiments with an open-source project, and to demonstrate the versatility of AutoGraphQL in generating tests that target the GraphQL implementation of an application, regardless of the underlying backend technology.

Table II summarizes the key characteristics of the case studies: the number of lines of code and of commits, the language implementing the GraphQL API and the domain of the case study. We use the latest stable release of Saleor, version 2.11, for our experiments. As mentioned in the table, this version contains $691K$ lines of code and the backend is in Python.

2) Frontapp: Frontapp is the primary website of our industrial partner, Redeye AB. Frontapp contains articles, financial analyses, tools, and video streams of events hosted by Redeye. The site is implemented in Symfony[10], a web application framework for PHP projects, and in JavaScript. Its GraphQL API is connected to multiple data sources and receives approximately $64K$ requests daily. Frontapp has been in production for more than 7 years. There is no automated test for the application, and it is tested manually by the QA team before major versions are released.

As presented in Table II, more than 20 Redeye developers have contributed about $7K$ commits to Frontapp. The application contains nearly $154K$ lines of code (LOC), as measured on February 09, 2021, and the GraphQL API is implemented in PHP.

[9] https://www.djangoproject.com/
[10] https://symfony.com/

B. Experiments

This section describes the experimental protocol followed and the metrics used to evaluate AutoGraphQL with Frontapp and Saleor.

1) Query Interceptor Configuration: As described in subsection III-E, the query interceptor of AutoGraphQL is specific to a given software stack. In order to conduct experiments with Saleor, we extend it with an agent, implemented as a GraphQL middleware [22]. This agent allows the query interceptor of AutoGraphQL to access queries that arrive at the GraphQL endpoint of Saleor, `/graphql/`, and log them into the query database. For our experiments with Frontapp, we configure a PHP event listener that triggers the query interceptor, which then logs all queries arriving on the GraphQL endpoint, which is also `/graphql/`.

2) Production Workloads: AutoGraphQL generates tests for queries that are triggered by user actions as part of interactions in production, during the monitoring phase of AutoGraphQL. We define such a production workload for each case study, as follows.

For our experiments with Saleor, we deploy the e-commerce application in a local server in our laboratory. In order to produce a realistic production workload, one of the authors interacted with the components on the frontend to perform typical operations related to e-commerce websites, such as browsing through the catalog of products, searching for specific products from the search bar, viewing the web-page for a product, and making orders. Additionally, the author performed administrative actions such as fetching the list of registered customers and orders, or searching for a specific customer or order. The experiment was carried out over the duration of nearly 3 hours.

The production workload for Frontapp consists of the interactions of actual end-users with the system. We do not ask the end-users to perform any specific operations, and simply log the queries that are generated as they browse through the website, reading articles or using its search feature, etc. We log these queries for a period of 33 days.

3) Metrics for Evaluation: In order to gauge the effectiveness of the tests generated by AutoGraphQL, we adopt the concept of schema coverage introduced by Karlsson *et al.* [15]. Compared to traditional code coverage, schema coverage is a more relevant metric to assess the tests generated by AutoGraphQL since they are intended to directly target the GraphQL schema.

Schema Coverage: We consider a GraphQL schema to be composed of a set of tuples of the form $\{Object_o, Field_n\}$, by combining all *Object o*, defined as a `type` or an `interface` in the schema, with each of its n fields. A query is said to reach a tuple $\{o, f\}$ if it requests for a field f defined in the object o. This is determined statically by analyzing the Abstract Syntax Tree of the query. The schema coverage (SCHEMA_COV) of a test, or the test suite, generated by AutoGraphQL is then defined as the number of tuples in the schema that are reached by the query, or set of queries, invoked by the test(s) (COVERED_TUPLES), divided by the total number of tuples in the schema (SCHEMA_TUPLES). This is presented in Equation 1.

$$\text{SCHEMA\_COV} = \frac{\text{COVERED\_TUPLES}}{\text{SCHEMA\_TUPLES}} \tag{1}$$

A schema coverage of 0% means that the test suite of a project does not invoke any queries that cover a tuple in the schema. On the other hand, a schema coverage of 100% would imply that all the tuples in the schema are covered by the test suite. For example, the query in Listing 2 covers 4 tuples, {*Query, teasers*}, {*Teaser, title*}, {*Teaser, subTitle*}, *and* {*Teaser, url*}, of the 13 tuples of the schema in Listing 1. The schema coverage of the corresponding test in Listing 6 is therefore 30.8%.

Metrics: We collect and report the following metrics for Frontapp and Saleor, based on their schema, logged production queries, test generation, and test execution:

1. TYPES is the number of types defined in the schema.
2. ENTRY_POINTS is the number of entry-points defined in the `Query` type of the schema.
3. UNIQUE_QUERIES is the number of unique combinations of queries and arguments logged during the experiment, and consequently, the number of tests generated.
4. ASSERTIONS_EVALUATED is the total number of assertions evaluated on executing the generated tests.
5. PASSING is the number of generated tests that pass.
6. FAILING is the number of generated tests that do not pass.
7. SCHEMA_FAULTS is the number of bugs found by the generated tests.
8. SCHEMA_TUPLES is the number of tuples obtained from the schema.
9. COVERED_TUPLES is the number of tuples of the schema reached by the generated tests.
10. SCHEMA_COV_GENERATED is the schema coverage achieved with the test suite generated by AutoGraphQL, per Equation 1.

All metrics are integer quantities, except for SCHEMA_COV_GENERATED which is expressed as a percentage.

V. Evaluation Results

This section presents the results obtained during our experiments with the two case studies. We summarize the results for all metrics introduced in subsubsection IV-B3 in Table III.

A. Case Study 1: Saleor

As presented in Table III, the GraphQL schema of Saleor defines 460 TYPES and 69 query ENTRY_POINTS. Based on the production workload defined in subsubsection IV-B2, AutoGraphQL logs 334 UNIQUE_QUERIES, and generates one test for each of them. We successfully execute all of these tests. The generated test suite triggers 43 of the 69 query entry-points in the schema. These tests cover 506 tuples (COVERED_TUPLES) out of the 1884

TABLE III: Results from the evaluation of AutoGraphQL on the two case studies

#	Metric	Saleor	Frontapp
1	TYPES	460	92
2	ENTRY_POINTS	69	23
3	UNIQUE_QUERIES	334	24,049
4	ASSERTIONS_EVALUATED	88,668	8,727,519
5	PASSING	334	23,892
6	FAILING	0	157
7	SCHEMA_FAULTS	0	8
8	SCHEMA_TUPLES	1884	875
9	COVERED_TUPLES	506	426
10	SCHEMA_COV_GENERATED	26.9%	48.7%

SCHEMA_TUPLES in Saleor. This results in a value of 26.9% for SCHEMA_COV_GENERATED.

The execution of the 334 test cases triggers the evaluation of 88,668 assertions (ASSERTIONS_EVALUATED). The difference between the number of tests and the number of assertions evaluated within the tests is because some assertions are made inside loops to verify the properties of elements that are lists, as illustrated on line 39 of Listing 6. Each of the 88,668 assertions verifies one expected property about the returned data, per the GraphQL schema. None of the 88,668 assertion evaluations fail, meaning that the generated test cases based on our selected production workload, do not reveal a schema fault in version 2.11 of Saleor's GraphQL resolvers. This is to be expected given the popularity and maturity of Saleor.

Saleor has a solid test suite written by the developers. Now we want to assess the complementarity of the original tests and the test cases generated by AutoGraphQL. In particular, we consider two metrics: SCHEMA_COV_ORIGINAL is the schema coverage achieved with the GraphQL requests triggered by the original test suite. DISTINCT_TUPLES is the number of schema tuples not covered by the original test suite but covered by the test suite generated by AutoGraphQL. A non-zero value for DISTINCT_TUPLES would imply that AutoGraphQL is able to generate valuable new tests.

The original test suite of Saleor includes 5405 test cases, which trigger 2340 requests, of which 1227 are queries and 1113 are mutations. Table IV shows the number of tuples involved in those tests. This original test suite covers 1429 of the 1884 tuples, resulting in a value of 75.8% for SCHEMA_COV_ORIGINAL. The AutoGraphQL test suite covers 506 tuples, including 483 tuples covered by the original as well as the generated test suite. Most importantly, the tests generated by AutoGraphQL using production queries cover 23 DISTINCT_TUPLES in the Saleor schema that are never covered by the original test suite, including one query entry-point. This confirms the findings of Wang *et al.* [17] and Tiwari *et al.* [16] that in-house tests can miss behavior that is exercised in production. These unique tuples covered by the generated tests complement the existing test cases, and the generated tests would contribute to the prevention of regression bugs in the resolvers that handle these tuples. We also note that 432 of the 1884 tuples in the schema are covered neither by the original tests, nor by the generated tests, showing that comprehensive schema testing is hard. We will discuss the possible reasons why parts of the schema are not covered by the generated tests in more detail in section VI.

TABLE IV: Coverage of Saleor schema tuples with original and generated tests

	COVERED_TUPLES
Original test suite	**1429** / 1884
AutoGraphQL-generated test suite	**506** / 1884
Intersection of AutoGraphQL and original test suites	**483** / 1884
Tuples only covered by AutoGraphQL tests (DISTINCT_TUPLES)	**23** / 1884

Highlight from the Saleor experiment

With the 334 GraphQL queries harvested during our experiments with Saleor, AutoGraphQL generates 334 test cases. These tests cover 26.9% of the schema, including 23 tuples in the schema that have not been covered by the developers in the original test suite. This reveals that the AutoGraphQL tests complement the original test suite with respect to the capability of detecting schema faults in GraphQL resolvers.

B. Case Study 2: Frontapp

From Table III, we see that the schema of Frontapp defines 92 TYPES and 23 query ENTRY_POINTS. The Frontapp schema does not define `Mutation` operations. The production workload of Frontapp, described in subsubsection IV-B2, observed over a monitoring period of 33 days, results in AutoGraphQL harvesting and storing 24,049 UNIQUE_QUERIES. The query most frequently invoked during this period was executed 301,016 times and is presented in Listing 5. Using the logged queries, AutoGraphQL generates 24,049 PHPUnit tests, all of which are successfully executed. Frontapp has 875 SCHEMA_TUPLES of which 426 are covered by the generated tests (COVERED_TUPLES), resulting in a SCHEMA_COV_GENERATED of 48.7%. The generated tests trigger 19 of the 23 entry-points in the schema. The number of ASSERTIONS_EVALUATED on the execution of the generated tests is 8,727,519.

Of the 24,049 generated tests, 157 fail. The developers at Redeye confirmed that these failures are caused by 8 distinct SCHEMA_FAULTS in Frontapp. The difference between the number of failures and the number of schema faults is the result of some test cases triggering the same query entry-points, and therefore, the same bugs. These schema faults are caused due to incorrect assumptions about the properties of

```
@@ -0 +3 @@
+ if (is_array($source['authorIds']) &&
+ count($source['authorIds']) > 0) {
    if (id::isValid($source['authorIds'][0])) {
      $firstAuthor = $this->personRepository->findById(
        id::fromString($source['authorIds'][0])
      );
      if ($firstAuthor && $firstAuthor->getTitle()) {
        $authorTitle = $firstAuthor->getTitle();
      }
    }
+ }
```

Listing 7: A schema fault discovered in Frontapp, and its resolution

objects, causing them to contradict the properties defined in the schema. For example, a nullable variable was sent to a resolver that could not handle a null input, or an element was collected in a non-nullable array without being checked for null first, or a resolver returned a different type than was stated in the schema.

Let us now look at an example of a schema fault found by a generated test. Listing 7 shows a bug located within a resolver defined in Frontapp. This bug was caused because the field `authorIds`, defined as non-nullable in the Frontapp schema, actually had the value of `null`, causing an exception to be raised (line 4). It was discovered due to a failing assertion in a test generated by AutoGraphQL, specifically the assertion that checks if the response has an `errors` field. The bug was consequently fixed by Frontapp developers by adding the highlighted check (lines 1 and 2) to ensure that `authorIds` is indeed not null before performing further computations on it.

As mentioned in subsection IV-A, we note that Frontapp does not have automated tests. Thus, the developers at Redeye proved to be interested in the AutoGraphQL test cases, which complement their manual QA activities. Furthermore, it is a possibility to push the AutoGraphQL tests in a repository with continuous integration, and to run them regularly to identify regressions or new schema faults as the application continues to evolve.

Highlight from the Frontapp experiment

AutoGraphQL harvests 24,049 GraphQL queries that are triggered due to interactions of Frontapp end-users in production, and generates as many tests. The generated test suite achieves a schema coverage of 48.7% and discovers 8 schema faults. Those faults have subsequently been fixed by the developers. This validates the capability of AutoGraphQL to automatically generate valuable tests that detect faults, from GraphQL queries observed in production.

VI. Discussion

We now reflect on some interesting aspects of AutoGraphQL.

Test Minimization and Prioritization: Test generation with AutoGraphQL is systematic in that each unique query harvested from production is used as input for the generation of one test. Thus, the number of harvested queries determines the size of the generated test suite, and consequently its execution time. For example, the 334 generated tests for Saleor are executed in 34 seconds, while the 24,049 tests generated for Frontapp take 114 hours to execute. In situations where the execution time is critical, such as in a continuous integration pipeline, it would be useful to minimize the generated test suite, as well as prioritize the execution of the tests [23], [24]. As described in subsection III-C, the query interceptor of AutoGraphQL aggregates meta-data about each query logged during the monitoring phase, including the number of times it was observed, as well as timestamps for when it was first and last invoked. This information may be used by developers to filter a subset of queries to use as inputs for test generation. For example, a developer may generate tests using the queries triggered at least 500 times within the last 3 days, and execute these tests in a prioritized fashion, based on criteria such as their schema coverage.

Mutation Requests: State of the art techniques for GraphQL, including cost analysis [5], [25], formal analysis [26], and GraphQL test generation [27] only support query requests, with the exception of [15] which provides support for the generation of mutation requests. Outside the academic literature, *Schemathesis* is an open-source tool that uses the GraphQL schema to generate property-based tests, it also only supports query requests[11]. However, we note that mutation operations are an equally integral part of GraphQL APIs. For example, the Saleor schema defines 222 mutation entry-points. Thus, there is a clear need for research on mutation requests. This is an interesting and challenging research endeavour because mutation requests have side-effects on the application database, which may result on breaking tests depending on test ordering and breakage of various assumptions on the application state.

GraphQL Schema Evolution and Test Generation: A GraphQL schema and the implementation of the corresponding resolvers may evolve at a different pace. Some parts of the schema may correspond to functionality that is slated to be deprecated, such as the field in a type that is to be replaced by another field. The schema might also specify elements that are yet to be implemented as part of a future release. For example, the developers at Redeye confirm that the Frontapp schema specifies more elements than those which can be handled by the current resolvers. This is due to the fact that Redeye began the process of migrating Frontapp from a REST API to a GraphQL API in 2019 [8]. This impacts the schema coverage achieved with the tests generated by AutoGraphQL, since there are tuples in the schema that are unreachable by design. For the same reason, the AutoGraphQL tests may become outdated and even break when the schema evolves. Therefore, an important direction for research in automated test generation for GraphQL is to understand how the tests may be evolved to address the schema evolution [28]. One approach to evolve these tests could involve repairing the generated test

[11] https://schemathesis.readthedocs.io/en/stable/graphql.html

suite [29].

VII. Related Work

This section discusses closely related work in the areas of test generation for web APIs, for data schemas, and based on production.

A. Test Generation for Web APIs

Currently, only two studies propose automated test generation strategies for GraphQL APIs. Vargas *et al.* [27] mutate GraphQL queries in existing tests in order to amplify them [30]. Karlsson *et al.* [15] produce randomly-generated queries and arguments based on the GraphQL schema, and use them as inputs in property-based tests. AutoGraphQL differs from these approaches because the inputs for test generation are not existing test queries or randomly generated ones, but queries that are harvested from production.

Several studies propose black-box test generation approaches for REST APIs [12]. In addition to the HTTP status code of the response, the oracles are often derived from OpenAPI/Swagger specifications describing the API. The parameters used as test inputs may be derived from the API specification [14], or produced randomly [13], [31]. Recently, deep learning models have been proposed to determine the validity of these inputs [32]. The generated tests can assess the robustness of the API through invalid requests [33], detect regressions across API versions [34], verify the data dependencies among sequences of requests [35], or verify the constraints imposed on their parameters [36]. Metamorphic relations among requests may also serve as the oracle [37]. EvoMaster [38] is a search-based, white-box approach to generate tests for RESTful web services. The technique is based on an evolutionary algorithm which rewards code coverage and fault-finding ability, the latter being determined by HTTP status codes. AutoGraphQL is a black-box approach, that is fundamentally different from these test generation techniques because it is tailored to GraphQL APIs, and uses GraphQL schemas and requests monitored in production.

B. Test Generation for Data Schemas

Traditional databases are also defined with a schema, as is GraphQL. Several studies use database schemas in the context of testing. For example, Khalek and Khurshid [39] use SQL schemas to generate SQL queries, test data, and oracles verifying the result of query execution, with the goal of testing database engines. McMinn *et al.* [40] and Alsharif *et al.* [10] propose search-based approaches that use the schema to generate test data for covering database integrity constraints. QAGen by Binnig *et al.* [41] generates meaningful test inputs based on the schema. XML schemas have also been used to produce XML instances automatically, which may be used as inputs for testing web services [42]–[45]. AutoGraphQL relates to this domain, but in a new technological context, that of web APIs and GraphQL: it uses the GraphQL schema of an application to produce oracles in the generated tests that verify the format of the GraphQL responses.

C. Test Generation Based on Production

A few studies propose test generation strategies using information obtained from production. For example, Oracle Database Replay [46] and Snowtrail [47] capture production queries made against databases, and replay them in order to detect regressions. Marchetto *et al.* [48] use event logs to generate Selenium tests for web applications. Hammoudi *et al.* [29] incrementally repair tests for web applications generated from record-replay tools. Tiwari *et al.* [16] monitor methods of interest in production in order to generate differential unit tests. Thummalapenta *et al.* [49] use execution traces to generate parameterized unit tests. ReCrash by Artzi *et al.* [50] reproduces failures through unit tests generated from runtime observations. AutoGraphQL is the first tool that harvests GraphQL queries from production to use as inputs for the generation of test cases.

VIII. Conclusion

GraphQL is a new way to specify web APIs. Though it continues to gain widespread adoption, few studies propose automated test generation strategies that target GraphQL API implementations. This paper introduces AutoGraphQL, the first tool that leverages production GraphQL queries to automatically generate test cases. The goal of the generated tests is to detect schema faults through oracles that verify that the response to a query conforms with the GraphQL schema.

We present the evaluation of AutoGraphQL on one open-source and one industrial case study, called Saleor and Frontapp: AutoGraphQL successfully generates tests for both projects. The tests generated for Saleor exercise 26.9% of the schema and cover regions in the GraphQL schema that are not covered by its original test suite. The tests generated for Frontapp exercise 48.7% of the schema and reveal 8 distinct schema faults. These experiments demonstrate that AutoGraphQL is capable of generating tests for untested behavior, as well as detecting errors that occur in the production environment.

An important future direction for AutoGraphQL is to analyze how these tests may be incorporated into a continuous integration pipeline. This would require the generated test suite to be minimized, and for the tests to run in a prioritized fashion. It would also be useful to understand how these tests may be evolved as a result of changes made to the API.

IX. Acknowledgements

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] R. T. Fielding and R. N. Taylor, *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[2] A. Davis and D. Zhang, "A comparative study of soap and dcom," *Journal of Systems and Software*, vol. 76, no. 2, pp. 157–169, 2005.

[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to soap, wsdl, and uddi," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[4] L. Byron, "GraphQL: A data query language." https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/ (accessed 2021-07-01).

[5] A. Cha, E. Wittern, G. Baudart, J. C. Davis, L. Mandel, and J. A. Laredo, "A principled approach to graphql query cost analysis," in *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 257–268, 2020.

[6] M. Seabra, M. F. Nazário, and G. Pinto, "Rest or graphql? a performance comparative study," in *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 123–132, 2019.

[7] G. Brito and M. T. Valente, "Rest vs graphql: A controlled experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 81–91, 2020.

[8] G. Brito, T. Mombach, and M. T. Valente, "Migrating to graphql: A practical assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 140–150, IEEE, 2019.

[9] S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "Can graphql replace rest? a study of their efficiency and viability," in *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pp. 10–17, IEEE, 2021.

[10] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "Domino: Fast and effective test data generation for relational database schemas," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 12–22, IEEE, 2018.

[11] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, "Schemaanalyst: Search-based test data generation for relational database schemas," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 586–590, IEEE, 2016.

[12] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Empirical comparison of black-box test case generation tools for restful apis," in *21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2021.

[13] S. Karlsson, A. Čaušević, and D. Sundmark, "Quickrest: Property-based test generation of openapi-described restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 131–141, IEEE, 2020.

[14] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, "Automatic generation of test cases for rest apis: A specification-based approach," in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[15] S. Karlsson, A. Čaušević, and D. Sundmark, "Automatic property-based testing of graphql apis," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 1–10, 2021.

[16] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Transactions on Reliability*, pp. 1–17, 2021.

[17] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 321–332, 2017.

[18] P. Erlandsson and J. Remes, "Performance comparison: Between graphql, rest & soap," Master's thesis, University of Skövde, 2020.

[19] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of graphql schemas," in *International Conference on Service-Oriented Computing*, pp. 3–19, Springer, 2019.

[20] M. Cederlund, "Performance of frameworks for declarative data fetching: An evaluation of falcor and relay+ graphql," Master's thesis, Kungliga Tekniska Högskolan, 2016.

[21] J. Imtiaz, M. Z. Iqbal, *et al.*, "An automated model-based approach to repair test suites of evolving web applications," *Journal of Systems and Software*, vol. 171, p. 110841, 2021.

[22] N. Burk, "Open Sourcing GraphQL Middleware - Library to Simplify Your Resolvers." https://www.prisma.io/blog/graphql-middleware-zie3iphithxy (accessed 2021-07-13).

[23] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: continuous prioritization for continuous integration," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 688–698, 2018.

[24] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020.

[25] G. Mavroudeas, G. Baudart, A. Cha, M. Hirzel, J. A. Laredo, M. Magdon-Ismail, L. Mandel, and E. Wittern, "Learning graphql query costs (extended version)," *arXiv preprint arXiv:2108.11139*, 2021.

[26] O. Hartig and J. Pérez, "Semantics and complexity of graphql," in *Proceedings of the 2018 World Wide Web Conference*, pp. 1155–1164, 2018.

[27] D. M. Vargas, A. F. Blanco, A. C. Vidaurre, J. P. S. Alcocer, M. M. Torres, A. Bergel, and S. Ducasse, "Deviation testing: A test case generation technique for graphql apis," in *11th International Workshop on Smalltalk Technologies (IWST)*, pp. 1–9, 2018.

[28] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st international conference on software testing, verification, and validation*, pp. 220–229, IEEE, 2008.

[29] M. Hammoudi, G. Rothermel, and A. Stocco, "Waterfall: An incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 751–762, 2016.

[30] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[31] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *ICSE 2019*, November 2019.

[32] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, and A. Ruiz-Cortés, "Deep learning-based prediction of test input validity for restful apis," in *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pp. 9–16, 2021.

[33] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of rest services," *IEEE Access*, vol. 9, pp. 24738–24754, 2021.

[34] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 312–323, 2020.

[35] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 142–152, IEEE, 2020.

[36] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: Black-box constraint-based testing of restful web apis," in *International Conference on Service-Oriented Computing*, pp. 459–475, Springer, 2020.

[37] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of restful web apis," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2017.

[38] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, Jan. 2019.

[39] S. Abdul Khalek and S. Khurshid, "Automated sql query generation for systematic testing of database engines," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), p. 329–332, Association for Computing Machinery, 2010.

[40] P. McMinn, C. J. Wright, C. Kinneer, C. J. McCurdy, M. Camara, and G. M. Kapfhammer, "Schemaanalyst: Search-based test data generation for relational database schemas," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 586–590, 2016.

[41] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: generating query-aware test databases," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 341–352, 2007.

[42] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Automatic test data generation for xml schema-based partition testing," in *Second International Workshop on Automation of Software Test (AST '07)*, pp. 4–4, 2007.

[43] J. M. Almendros-Jiménez and A. Becerra-Terón, "Xquery testing from xml schema based random test cases," in *Database and expert systems applications*, pp. 268–282, Springer, 2015.

[44] D. Petrova-Antonova, K. Kuncheva, and S. Ilieva, "Automatic generation of test data for xml schema-based testing of web services," in *2015 10th*

International Joint Conference on Software Technologies (ICSOFT), vol. 1, pp. 1–8, IEEE, 2015.

[45] S. C. Lee and J. Offutt, "Generating test cases for xml-based web component interactions using mutation analysis," in *Proceedings 12th International Symposium on Software Reliability Engineering*, pp. 200–209, IEEE, 2001.

[46] Y. Wang, S. Buranawatanachoke, R. Colle, K. Dias, L. Galanis, S. Papadomanolakis, and U. Shaft, "Real application testing with database replay," in *Proceedings of the Second International Workshop on Testing Database Systems*, pp. 1–6, 2009.

[47] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee, "Snowtrail: Testing with production queries on a cloud database," in *Proceedings of the Workshop on Testing Database Systems*, pp. 1–6, 2018.

[48] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 121–130, IEEE, 2008.

[49] S. Thummalapenta, J. De Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *International Conference on Tests and Proofs*, pp. 77–93, Springer, 2010.

[50] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *European conference on object-oriented programming*, pp. 542–565, Springer, 2008.

PROZE: Generating Parameterized Unit Tests Informed by Runtime Data

Deepika Tiwari*, Yogya Gamage†, Martin Monperrus*, and Benoit Baudry†

*KTH Royal Institute of Technology, *{deepikat, monperrus}@kth.se*
†Université de Montréal, *{yogya.gamage, benoit.baudry}@umontreal.ca*

***Abstract*—Typically, a conventional unit test (CUT) verifies the expected behavior of the unit under test through one specific input / output pair. In contrast, a parameterized unit test (PUT) receives a set of inputs as arguments, and contains assertions that are expected to hold true for all these inputs. PUTs increase test quality, as they assess correctness on a broad scope of inputs and behaviors. However, defining assertions over a set of inputs is a hard task for developers, which limits the adoption of PUTs in practice. In this paper, we address the problem of finding oracles for PUTs that hold over multiple inputs.**

We design a system called PROZE, that generates PUTs by identifying developer-written assertions that are valid for more than one test input. We implement our approach as a two-step methodology: first, at runtime, we collect inputs for a target method that is invoked within a CUT; next, we isolate the valid assertions of the CUT to be used within a PUT.

We evaluate our approach against 5 real-world Java modules, and collect valid inputs for 128 target methods, from test and field executions. We generate 2,287 PUTs, which invoke the target methods with a significantly larger number of test inputs than the original CUTs. We execute the PUTs and find 217 that provably demonstrate that their oracles hold for a larger range of inputs than envisioned by the developers. From a testing theory perspective, our results show that developers express assertions within CUTs, which actually hold beyond one particular input.

I. Introduction

Within all serious software projects, developers write hundreds of unit tests that automatically verify the expected behaviors of individual components in their systems [12]. As part of their testing effort, developers spend time deciding on concrete pairs of test input and oracle [3]. The input brings the system to the desired testable state through a series of actions, such as method invocations. The oracle verifies the expected behavior, and is typically expressed through assertion statements. Conventional unit tests (CUTs) assess the behavior of the program with respect to one specific input - output pair.

Parameterized unit tests (PUTs) are a more advanced form of tests. Unlike CUTs, PUTs evaluate multiple test inputs against the oracle [26], exercising more behaviors of the unit under test [7, 22]. Writing PUTs is notoriously hard for developers, which is why they are not widely used in practice [23, 31]. The major blockers for writing PUTs are envisioning the appropriate oracle and defining representative inputs [9]. In this paper, our key insights are that certain assertions already present in conventional unit tests are suitable for parameterized unit tests [20], and that runtime data can be utilized within parameterized unit tests [33].

We present PROZE, a novel system that analyzes CUTs in order to generate PUTs. PROZE automatically detects assertions that are suitable for PUTs, according to the following methodology. First, for a target method that is directly invoked by a CUT, it captures additional input arguments from its invocations at runtime, across test and field executions of the program. Next, PROZE transforms the candidate CUT into PUTs that invoke the target method with its arguments captured at runtime. Each generated PUT contains a developer written assertion that is evaluated against all captured inputs. We categorize each PUT output by PROZE as: 1) strongly-coupled if it is valid only for the original test input and cannot generalize to other inputs; 2) decoupled if it is valid for all observed input values; and 3) falsifiably-coupled if it is valid for a subset of input values and fails for others.

We evaluate PROZE against 5 modules from real-world Java projects. PROZE monitors the execution of the developer-written test suite of each module, as well as its execution under a usage workload, to capture new test inputs, and synthesize parameterized unit tests. From our experiments with the 5 modules, we observe that PROZE successfully generates 2,287 PUTs which compile and can be run with off-the-shelf test harnesses. We find 217 PUTs that are provably valid over a broader range of inputs.

PROZE is fundamentally novel. We are aware of only a few papers exploring the field of PUT generation: Fraser *et al.* concentrate on symbolic pre- and postconditions [7], Thummalapenta *et al.* look at simple tests [23], and Elbaum *et al.*. consider system tests [6]. Our work is the first to generalize CUTs into PUTs for real world, complex tests, using runtime data. Our key contributions are:

- PROZE, a novel methodology to generate parameterized unit tests by identifying developer-written assertions that hold over multiple inputs collected at runtime.
- Evidence that 1) some real-world assertions hold for multiple inputs beyond the original ones, and 2) PROZE is able to generate executable parameterized unit tests that increase input space coverage by several orders of magnitude.
- A publicly available implementation of the approach, which is able to generate parameterized unit tests for modern Java software and testing frameworks.

```java
// CUT with an oracle coupled to the input
@Test
public void testOneElement() {
  PeriodicTable table = PeriodicTable.create();
  Element element = table.getElement("He");
  assertEquals("Helium", element.getFullName());
  assertEquals(4, element.calculateMassNumber());
  assertTrue(table.getGroup18().contains(element));
}
.....

// PUT with a more general oracle
@ParameterizedTest
@MethodSource("provideSymbol")
public void testManyElements(String symbol) {
  PeriodicTable table = PeriodicTable.create();
  Element element = table.getElement(symbol);
  int atomicNum = element.getAtomicNumber();
  int massNum = element.calculateMassNumber();
  int numNeutrons = element.getNeutrons();
  assertEquals(numNeutrons, (massNum - atomicNum));
  assertTrue(atomicNum < massNum);
  assertTrue(table.getGroup18().contains(element));
}

// Argument provider for the PUT
private static Stream<Arguments> provideSymbol() {
  return Stream.of(
    Arguments.of("He"),
    Arguments.of("Ne"),
    Arguments.of("Ar"),
    Arguments.of("Kr"),
    Arguments.of("Xe"),
    Arguments.of("Rn"),
    Arguments.of("Og")
  );
}
```

Listing 1. `testOneElement` is a conventional unit test (CUT) with three assertions evaluated against a single input. `testManyElements` is a parameterized unit test (PUT) which receives a `String` argument from the argument provider `provideSymbol`. Its three assertions hold over the 7 unique inputs supplied by `provideSymbol`.

II. Background

We now introduce and illustrate the two main concepts for our work: conventional unit tests and parameterized unit tests. We reuse the terminology of conventional vs. parameterized introduced in the seminal series of works by de Halleux, Tillman, and Xie [35].

A. Conventional Unit Tests (CUT)

A CUT is a developer-written test case that typically invokes the unit under test with a single input state, and verifies the expected output state through an oracle [35]. One single CUT can test the behavior of multiple methods that belong to the unit under test, and can define multiple assertions to check the behavior of these methods for a specific test input [34].

Lines 1 to 9 of Listing 1 present the CUT `testOneElement`, annotated with the `@Test` JUnit annotation. The method call `getElement` within the CUT (line 5) fetches the element with the symbol `"He"`, from an instance of `PeriodicTable`. The CUT contains three assertions, verifying the expected properties related to the element. The first and second assertions verify its full name and its mass number, respectively. These two assertions are specific to the given input, and will not hold if `getElement` is invoked with any other `String` argument. The third assertion verifies that the element is contained within the set of elements in Group 18 of the `table`. This is a more general assertion, which will hold for all Group 18 elements.

B. Parameterized Unit Tests (PUT)

A PUT is a test case that receives one or more parameters [11], and that invokes the unit under test with multiple inputs corresponding to the input parameters. A PUT typically expresses a more general oracle than a CUT, by design, since it must hold over a set of behaviors of the unit under test. The parameters of a PUT are automatically assigned values through an *argument provider*. The oracle is evaluated each time the PUT is invoked with a new argument.

Argument provider: An argument provider can be an array of values or a helper method that returns values. The argument provider for a PUT returns values that are of the same type as the type of the PUT's parameters. The PUT is invoked as many times as there are values in the argument provider.

Consider the method `testManyElements` of Listing 1 (lines 12 to 24). The JUnit annotation `@ParameterizedTest` declares that this method is a PUT [22]. Next, the annotation `@MethodSource` links it to the argument provider method `provideSymbol` (lines 26 to 37). When the PUT is run, its `String` parameter `symbol` is assigned one value supplied by `provideSymbol`, resulting in 7 unique runs of the PUT in total. Within the PUT, the method `getElement` gets invoked with the `symbol` (line 17), and returns the corresponding element from the `table`. None of the three assertions in the PUT are specific to the input. The first and second assertions verify properties related to the atomic and mass numbers of the element, computed within the test. The third assertion is the same as the third assertion of `testOneElement`, and checks that the element is contained within Group 18 of the `table`. All three assertions hold over all the 7 `String` input arguments supplied by the argument provider.

In this work, we hypothesize that some developer-written CUTs include an oracle that is meant to be valid over more than one test input [20]. We see this with the third assertion of the CUT `testOneElement`, which holds for 6 more inputs in addition to the original one. We aim at automatically retrieving such assertions, to harness them within a PUT.

III. PROZE

This section describes PROZE, a technique that generates parameterized unit tests informed by runtime data.

A. Overview

We summarize our approach in Figure 1. PROZE accepts as input 1) the source and test code, including all developer-written CUTs, of a project, and 2) a workload for the project, which automates calling end-user functionalities (see subsection IV-B). The latter is used for triggering runtime usage with realistic values. The final output of PROZE is a

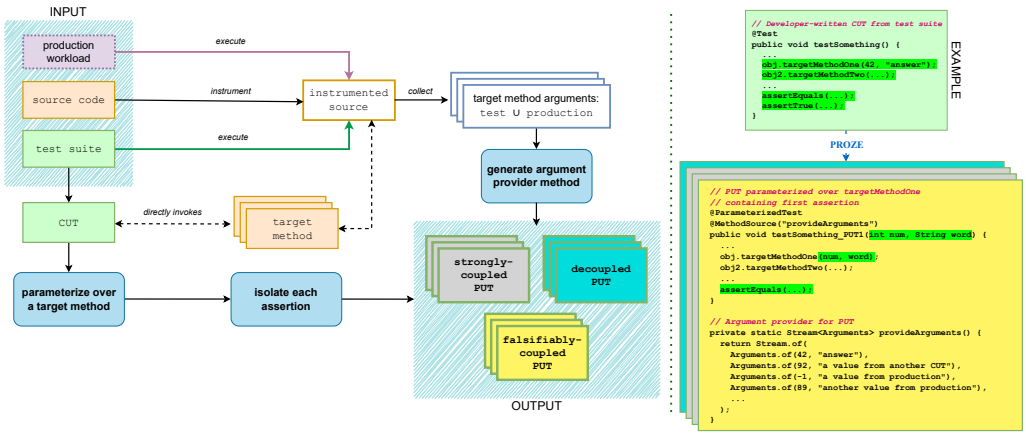


Fig. 1. An overview of PROZE, a novel technique for the automated transformation of a developer-written conventional unit test (CUT) into parameterized unit tests (PUTs). Given a project and its original test suite composed of a set of CUTs, PROZE generates a set of PUTs, which are classified as either strongly-coupled, decoupled or falsifiably-coupled, depending on how much the oracle is tied to the original input. The CUT on the top right is transformed into corresponding PUTs on the bottom right. Each PUT is fed with arguments collected at runtime with bespoke monitoring.

set of parameterized unit tests and their corresponding argument providers. The PUTs are classified as strongly-coupled, decoupled, falsifiably-coupled depending on whether the oracle in the PUT is tied to the original input, or if it holds for a larger set of inputs. Figure 1 presents an example of this transformation for the CUT `testSomething` into a set of corresponding PUTs. Each PUT derived from the CUT focuses on a single method and contains one of the CUT's assertions. We highlight one of these PUTs, which contains the first assertion from the original CUT, and has parameter types `int, String`, corresponding to the parameter types of the method `targetMethodOne(int, String)`. In the following subsections, we present details of the automated transformation of developer-written CUTs into PUTs that harness one assertion each.

B. Capturing PUT inputs

A PUT has an argument provider, which enumerates the inputs to be used in the test. Hence, the first step of PROZE is to collect these possible inputs. For this task, PROZE sources inputs by monitoring test and production executions. Since monitoring is costly, PROZE first identifies a set of target methods [11] to instrument so as to monitor their invocations at runtime and capture valid arguments for them.

Selecting and instrumenting target methods: Given the input project, PROZE statically analyzes all the CUTs within its test suite (i.e., all methods annotated with `@Test`). As depicted in Figure 1, PROZE uses this analysis to map each CUT to the set of methods and constructors that it directly invokes. PROZE selects *target methods* that accept arguments of primitive and/or `String` types, because of implementation constraints on argument providers. All CUTs that invoke at least one candidate target method will be tentatively transformed into PUTs by PROZE.

Next, PROZE instruments each target method, to capture the input arguments for each of its invocations during the execution of the program. The instrumentation consists of instructions that trigger the serialization of the input arguments of a target method each time it gets invoked at runtime, as well as the stack trace. These input arguments will later be used inside the argument providers of the generated PUTs.

Capturing target method arguments: In order to collect a set of input arguments for each target method, the instrumented version of the program under test is executed. PROZE monitors the execution of the developer-written test suite. This enables it to 'transplant' the test inputs that appear in other tests, within the generated parameterized unit tests for the method.

Additionally, our key insight is that we can use the same instrumentation to monitor the execution of the program when it is exercised with an end-to-end workload, possibly in production. This allows us to collect a wide range of realistic inputs for a target method. As illustrated in Figure 1, PROZE computes the union of all arguments passed to a target method during the execution of the test suite and during the execution of the end-to-end workload.

C. Automatic generation of PUTs

At this stage, PROZE generates a set of PUTs for each original CUT. There are three key steps that PROZE takes for this transformation, presented as blue boxes in Figure 1.

1) Parameterize over a target method: We design PROZE to generate one test class per target method invoked by a CUT. The values passed to the parameters of a PUT are assigned as the arguments to a single target method. We refer to the PUT as

being *parameterized over the target method*. This is illustrated in the example of Figure 1, where the two arguments to the PUT `testSomething_PUT1` are assigned as the arguments of `targetMethodOne`.

2) Isolate each assertion: A key design decision of PROZE is that each PUT has one single assertion, extracted from the original CUT. When the original CUT has multiple assertions, they verify different aspects of the expected behavior. Isolating each assertion within its own PUT is essential for a clear test intention through one single assertion, and to enable individual assessment of each assertion against the captured inputs.

As a result, for a CUT that invokes α target methods and that specifies β assertions, PROZE generates $\alpha \times \beta$ PUTs. Each PUT has parameters that are of the same type as the parameters of the target method it parameterizes over, and it invokes the target method with the values passed to these parameters. We see this in the example of Figure 1, where the PUT has parameter type `int, String`, corresponding to the target method `targetMethodOne(int, String)`, and contains the first assertion from `testSomething`.

3) Generate argument provider method: PROZE generates one argument provider method in the test class, which is in charge of providing concrete values to the parameters of the PUTs. For our example CUT `testSomething` of Figure 1, PROZE generates 2 test classes and 4 PUTs in total (2 target methods *times* 2 assertions). Specifically, for each of the target methods `targetMethodOne` and `targetMethodTwo`, PROZE generates one test class with two PUTs, one for each assertion. The PUTs parameterized over `targetMethodOne` take a pair of `int` and `String` parameters, corresponding to the parameter signature of the target method.

PROZE uses the union of the arguments of a target method captured at runtime, in order to construct the argument provider. The argument provider is implemented as a method that is linked to the PUT through a specific annotation, such as `@MethodSource` for JUnit, or with `@DataProvider` for TestNG. As we see with the method `provideArguments` in the example of Figure 1, this method returns an array of values, each of which corresponds to one argument that is passed to the PUT. When the PUT is executed, this provider sequentially supplies it with one input argument.

D. Classes of generated PUTs

Once all the PUTs have been generated, PROZE executes them. As its final output, PROZE categorizes each generated PUT as:

- strongly-coupled if the PUT passes only when supplied with all the original arguments of the target method, and no additional arguments supplied by the provider. In other words, the assertion is coupled to exactly the test inputs defined in the original CUT, and it does not hold over a larger set of input arguments. This represents cases where the transformation of the CUT into a PUT parameterized over the target method is technically feasible, yet not useful due to the lack of an appropriate oracle.
- decoupled if the PUT passes for all the arguments that are supplied to it by the argument provider, including original and non-original arguments of the target method. This means that the assertion holds for all inputs to the target method contained in the captured union. The assertions that fall in this category are from two distinct categories: cases of parameterization where the oracle is indeed correct over all captured inputs (e.g., the PUT in Listing 1), and cases where the assertion is not coupled to the target method's behavior. The latter is not interesting for testing, and the corresponding PUTs are not relevant.
- falsifiably-coupled if the PUT passes when it is supplied with all the original arguments of the target method, as well as a subset of the additional arguments supplied by the provider. The latter includes target method arguments that are not present in the original CUT from which the PUT is derived. For a PUT that falls in this category, it is important that it fails for some of the arguments supplied by the provider, as this indicates that the assertion is sensitive to the input passed to the target method. Such cases represent assertions that hold over a larger set of inputs [20], yet can successfully distinguish between valid and invalid inputs. These PUTs are the most interesting, as they provide sound evidence that an existing oracle can assess the behavior of a target method over a large set of inputs. These PUTs can be effectively used to increase the coverage of the target's input space.

In addition to these three categories, we also find cases where the PUT fails when it is supplied with any of the original arguments of the target method, regardless of the other inputs supplied by the argument provider. Such ill-formed cases are due to side-effects from modifying target method invocations and removing assertion statements during PUT generation, per the core algorithm of PROZE. They are filtered out and not delivered as the output by PROZE.

We note that our categorization of test assertions as strongly-coupled, falsifiably-coupled, or decoupled is novel, it captures an essential facet of the oracle problem [3] in the context of PUT generation.

E. Illustrative example

Listing 2 presents an excerpt of a CUT called `testRadioButtons`, written by the developers of PDFBOX, a notable open-source PDF manipulation library [5]. The test loads a PDF document which contains a form, and fetches the form field containing a group of radio buttons (lines 3 to 6). The radio button field can be set to one of three options, `"a"`, `"b"`, or `"c"`. Next, on line 7, the method `setValue` is invoked with the `String` value `"b"`, to select the option `"b"` of the button, also updating its visual appearance. This modified PDF is saved and closed (lines 8 to 10), and then reloaded as a new document (line 12). After fetching the same radio button field, the two assertions verify that the selected value of the button is still `"b"` (line 15), and that there is only one export value corresponding to the selected value (line 16).

```
1 @Test
2 public void testRadioButtons() {
3   File pdf1 = new File("target/RadioBtns.pdf");
4   PDDocument doc1 = PDDocument.load(pdf1);
5   PDAcroForm form1 = doc1
        .getDocumentCatalog().getAcroForm();
6   PDRadioButton radioButton1 = (PDRadioButton)
        form1.getField("MyRadioButton");
7   radioButton1.setValue("b");
8   File pdf2 = new File("target/RadioBtns-mod.pdf");
9   doc1.save(pdf2);
10  doc1.close();

12  PDDocument doc2 = PDDocument.load(pdf2);
13  PDAcroForm form2 = doc2
        .getDocumentCatalog().getAcroForm();
14  PDRadioButton radioButton2 = (PDRadioButton)
        form2.getField("MyRadioButton");
15  assertEquals("b", radioButton2.getValue());
16  assertEquals(1, radioButton2
        .getSelectedExportValues().size());
17  doc2.close();
18 }
```

Listing 2. `testRadioButtons` is a developer-written CUT in the PDFBox project. The target method `setValue` is invoked within the test with a `String` argument (line 7). We present an excerpt of this CUT, containing two of its twelve assertions.

```
1 @ParameterizedTest
2 @MethodSource("provideArgs")
3 public void testRadioButtons_PUT(String value) {
4   File pdf1 = new File("target/RadioBtns.pdf");
5   PDDocument doc1 = PDDocument.load(pdf1);
6   PDAcroForm form1 = doc1
        .getDocumentCatalog().getAcroForm();
7   PDRadioButton radioButton1 = (PDRadioButton)
        form1.getField("MyRadioButton");
8   radioButton1.setValue(value);
9   File pdf2 = new File("target/RadioBtns-mod.pdf");
10  doc1.save(pdf2);
11  doc1.close();

13  PDDocument doc2 = PDDocument.load(new
        File("target/RadioButtons-modified.pdf"));
14  PDAcroForm acroForm2 = doc2
        .getDocumentCatalog().getAcroForm();
15  PDRadioButton radioButton2 = (PDRadioButton)
        acroForm2.getField("MyRadioButton");
16  assertEquals(1, radioButton2
        .getSelectedExportValues().size());
17  doc2.close();
18 }

20 // Provide PUT arguments from the captured union
21 private static Stream<Arguments> provideArgs() {
22   return java.util.stream.Stream.of(
23     Arguments.of("Off"),
24     ...
25     Arguments.of("Yes"),
26     Arguments.of("b"),
27     Arguments.of("c")
28   );
29 }
```

Listing 3. PROZE generates this parameterized unit test from the CUT `testRadioButtons` of Listing 2. It takes a `String` parameter called `value`. The target method `setValue` is invoked within the PUT with `value` (line 8). The PUT contains a single assertion (line 16). The method `provideArgs` is the argument provider for the PUT, supplying 12 unique arguments to the PUT from data captured at runtime.

PROZE statically analyzes `testRadioButtons`, identifies the method `setValue` as a target method, and consequently instruments this method to capture its `String` arguments at runtime. During the execution of the test suite of PDFBox, as well as its execution under a production workload, PROZE monitors the invocations of `setValue`, capturing the 12 unique `String` arguments that `setValue` gets invoked with. This set of arguments contains the `String` `"b"` that `setValue` is invoked with in the CUT. We refer to `"b"` as the *original argument* of `setValue` within the CUT.

After capturing this union of input arguments at runtime, PROZE generates two PUTs for `testRadioButtons` that are parameterized over `setValue`, one for each of the two assertions in Listing 2. Lines 1 to 18 of Listing 3 present the PUT that includes the second assertion of the CUT (line 16 of Listing 2). The PUT is annotated with the JUnit5 `@ParameterizedTest` annotation. This PUT has a single `String` parameter corresponding to the `String` parameter type of `setValue`, highlighted on line 3, and it contains one assertion at line 16. PROZE encapsulates the 12 captured `String` arguments of `setValue` into the data provider `provideArgs` (lines 20 to 29 in Listing 3). This data provider is linked to the generated PUT through the `@MethodSource` JUnit5 annotation (line 2). When executed, the PUT accepts each argument sequentially from the provider and assigns it as the argument for the invocation of `setValue` (highlighted on line 8). This results in 12 unique executions of the test, each with a distinct argument supplied to the PUT by `provideArgs`. PROZE classifies this PUT as falsifiably-coupled with `setValue`, since it passes for 2 of the 12 arguments it is invoked with, i.e., the original argument `"b"` as well as one additional argument `"c"`. On the other hand, the generated PUT containing the assertion on line 15 of Listing 2 is classified as strongly-coupled, as it passes only with `"b"`.

F. Implementation details

PROZE is implemented in Java. It is implemented using open-source libraries, including Spoon [19] for code analysis and generation, and Glowroot [29] for dynamic instrumentation and monitoring. PROZE generates PUTs in two of the most widely used test frameworks in Java [14], JUnit5 and TestNG. Our implementation is publicly available at https://github.com/ASSERT-KTH/proze/.

IV. Experimental Methodology

This section describes our methodology to assess the ability of PROZE to transform developer-written CUTs into PUTs, and in particular to identify the assertions that are suitable for parameterization.

A. Research questions

We answer the following research questions as part of our evaluation.

RQ1 [*PUT arguments*]: To what extent does PROZE collect and increase the input space coverage for the target methods?

RQ2 [*PUT classification*]: To what extent does PROZE extend the range of the oracle over the input space?

RQ3 [*PUT representativeness*]: How do the PUTs generated by PROZE relate to the testing practices of developers?

B. Dataset: Applications and workloads

Table I presents details of the five Java modules we use for the evaluation of our approach, including the link to the exact version we use for our experiments (Commit SHA), and the number of lines of Java source code (LoC, counted with `cloc`). We also report the number of developer-written CUTs in their test suite (TOTAL CUTS), and the TOTAL ASSERTIONS across these CUTs.

TABLE I
DESCRIPTION OF THE FIVE JAVA MODULES USED AS CASE STUDIES IN OUR EVALUATION.

MODULE	SHA	LOC	TOTAL CUTS	TOTAL ASSERTIONS
`fontbox`	`8876e8e`	16,815	25	156
`xmpbox`	`8876e8e`	7,181	54	222
`pdfbox`	`8876e8e`	84,764	429	1,633
`query`	`3ea38dd`	2,472	33	42
`sso`	`3ea38dd`	13,520	83	152
TOTAL			**624**	**2,205**

PDFBOX: The first three modules we consider belong to PDFBox, a popular open-source project by the Apache Software Foundation, which supports operations for working with PDF documents [5]. We experiment with three modules of PDFBOX version 2.0.24, called `fontbox`, `xmpbox`, and `pdfbox`. The `fontbox` module (first row of Table I) handles fonts within PDF documents. It has more than 16K lines of Java source code, and its test suite contains 25 developer-written CUTs implemented as JUnit tests. The total number of assertions across these 25 CUTs is 156. The second module, `xmpbox`, is responsible for managing metadata within PDF documents, based on the eXtensible Metadata Platform (XMP) specification. Its source code comprises of 7K lines of Java code, and the 54 CUTs in its test suite contain 222 assertions in total. The core module of the PDFBOX project is `pdfbox`, which is the third one we experiment with. With upwards of 84K lines of Java source code, `pdfbox` is the largest module in our dataset. Its test suite has 429 developer-written CUTs containing 1,633 assertion statements in total.

We run the test suite of `pdfbox` in order to capture the arguments with which its target methods are invoked at runtime. We also exercise `pdfbox` with the production workload from [29], performing operations that are typical of PDF documents, including encryption, decryption, and conversion to/from text and images, on 5 real-world PDF documents sourced from [8].

WSO2-SAML: The second project we consider for our evaluation is `identity-inbound-auth-saml` (henceforth referred to as WSO2-SAML). It is an extension of the WSO2 identity server, which provides identity and access management across commercial applications worldwide. The WSO2-SAML project contains functionalities for performing user authentication using the Security Assertion Markup Language (SAML) protocol [21]. We work with two modules within WSO2-SAML version 5.11.41. The first module, `query` (row 4 of Table I), handles application authentication- and authorization-related queries. It has nearly 2.5K lines of Java code, and 33 developer-written TestNG CUTs. The `sso` module supports Single SignOn (SSO) for SAML. There are 13.5K lines of Java code within this module, and 83 CUTs with 152 assertions.

PROZE instruments each target method called by the CUTs across `query` and `sso` with the goal of capturing their input arguments. These arguments are captured while the test suite of WSO2-SAML is run and when the field workload is executed. For this experiment, we use three sample web applications for WSO2-SAML: `pickup-dispatch`, `pickup-manager`, and `travelocity`. We first deploy WSO2-SAML and register the three sample applications using the GUI of the WSO2 identity server. Then we configure SSO and Single LogOut (SLO) for all three applications. We exercise SSO by logging into `pickup-dispatch` and consequently accessing `pickup-manager` and `travelocity`. Similarly, we perform SLO by logging out from one application. We also modify the configuration of the applications, such as sharing user attributes (i.e., *claims* in SAML), including name, country, and email, and enabling claim-sharing between the three applications.

C. Protocol

We run each of the five modules per their test and field workloads detailed in subsection IV-B, while PROZE captures arguments for the target methods it identifies within their developer-written CUTs. For a target method, if PROZE captures more than one argument from different execution contexts, i.e., test and field, it generates PUTs by transforming each CUT that invokes the method, as well as the corresponding argument providers. Per section III, each generated PUT is parameterized over a single target method, and contains one of the CUT's assertions. Based on this methodology, this subsection presents the protocol we employ to answer the three research questions listed in subsection IV-A.

RQ1 [PUT arguments]: As the answer to our first question, we report the number of target methods for which PROZE successfully captures a union of arguments at runtime over test and field executions, and transforms the CUTs that invoke them into PUTs. As argued by Kuhn *et al.* [15], adequately covering the input space within tests has a positive relationship with test quality. Therefore, for each target method, the key metric for this RQ is the number of unique arguments PROZE captures for it. These arguments are encapsulated within the argument provider for each PUT that is parameterized over the target method. This set of arguments represents the increase in the input space coverage of the target method.

RQ2 [PUT classification]: PROZE outputs a set of PUTs which are classified as either strongly-coupled, decoupled or falsifiably-coupled. To answer RQ2, we report the number of PUTs in each of these categories. Per subsection III-D,

TABLE II
RQ1: PROZE CAPTURES RUNTIME ARGUMENTS (#CAP. ARGS) FOR TARGET METHODS ACROSS THE 5 MODULES, GENERATING PUTS FROM THE CUTS THAT DIRECTLY INVOKE THEM WITH #ORIG. ARGS.

MODULE	TARGET METHODS	CUTS	#ORIG. ARGS	PUTS	#CAP. ARGS
fontbox	**12**	**15**	**med. 2**	**188**	**med. 60**
EXAMPLES	getLeftSideBearing	2	8	15	205
	getGlyph	3	5	14	187
	getGlyphId	1	2	8	68
xmpbox	**27**	**36**	**med. 1**	**594**	**med. 5**
EXAMPLES	createText	6	7	32	738
	addQualifiedBagValue	2	3	10	630
	addUnqualifiedSeqValue	3	4	30	117
pdfbox	**55**	**155**	**med. 1**	**1,388**	**med. 8**
EXAMPLES	encode	2	2	12	2,694
	getStringWidth	2	3	6	2,351
	getPDFName	13	1,016	64	1,193
query	**2**	**3**	**med. 1**	**8**	**med. 3**
EXAMPLES	getServiceProviderConfig	1	1	3	3
	getIssuer	2	2	5	3
sso	**32**	**39**	**med. 1**	**109**	**med. 6**
EXAMPLES	unmarshall	2	2	3	35
	decodeForPost	1	1	1	24
	encode	1	1	2	23
TOTAL	**128**	**248**	**2,739**	**2,287**	**13,021**

a falsifiably-coupled PUT provides concrete evidence that its oracle is valid over a larger range of test inputs than the original input of the CUT from which it is derived.

RQ3 [PUT representativeness]: For this RQ, we present a qualitative analysis of the three most notable techniques for the automatic generation of PUTs, with respect to the PUTs generated by PROZE. The three related works we consider for this analysis, [7], [11], and [36] employ diverse methodologies for automatically generating PUTs. We summarize their approaches and reason about the representativeness of the PUTs generated by them against those generated by PROZE.

V. EXPERIMENTAL RESULTS

This section presents the answers to the research questions introduced in subsection IV-A, based on our experiments with the 5 modules described in subsection IV-B, and using the protocol detailed in subsection IV-C.

A. RQ1: PUT arguments

Table II summarizes the results for our first RQ. For each MODULE, the highlighted rows present the total number of TARGET METHODS, the total number of CUTS that directly invoke them, and the median number of original arguments passed to these methods from the CUTS (#ORIG. ARGS). Next, we give the number of PUTS that PROZE generates from these CUTs, and the median number of arguments captured (#CAP. ARGS) at runtime for the target methods in the module. Under each module, we list examples of target methods for which PROZE captures a large number of arguments, including the number of CUTs that invoke them, the number of original arguments that they use to do so, the number of PUTs generated by PROZE for the target method, as well as the number of arguments captured for it.

Per the methodology described in section III, PROZE statically analyzes each of the 624 CUTs across the 5 modules in Table I, to select target methods, and instruments these methods to capture their arguments at runtime. However, not all of these 624 CUTs can be transformed into PUTs. First, not all of them contain appropriate target methods per the criteria explained in subsection III-B. Second, even if a CUT contains a target method, the method may not be invoked at runtime. Overall, we see that PROZE captures data for 128 target methods across test and field executions, which are directly invoked by 248 of the 624 CUTs in the 5 modules. This means that these 128 target methods are invoked in the field and/or by at least one developer-written test. Furthermore, for each of these 128 methods, PROZE captures at least one runtime argument in addition to the original argument, from different sources. In total, PROZE captures 13,021 arguments at runtime for the 128 target methods, which are passed to 2,287 generated PUTs.

The first few rows of Table II show the results for the three PDFBOX modules, `fontbox`, `xmpbox`, and `pdfbox`. PROZE generates 2,170 PUTs in total for the 94 target methods across these three modules. The PUTs are derived from the 206 CUTs that invoke the 94 methods directly. We see the largest numbers for the `pdfbox` module, owing to its large size, as highlighted in Table I. The 55 target methods in `pdfbox` are invoked by 155 CUTs, from which PROZE generates 1,388 PUTs. The median number of original arguments passed to these methods from the CUTs is 1, whereas the median number of arguments captured by PROZE for these 55 methods is 8. This means that PROZE increases the coverage of the input space of 22 methods only by a few values. For example, one PUT invokes the method `setFlyScale(float)` with the original argument passed by the CUT `saveAndReadTransitions`, and one more argument captured when running the whole test suite and workload. On the other hand, PROZE significantly increases the input space coverage for the other 23 methods. For example, for the method `encode(int)`, PROZE captures 2,694 arguments, the maximum among the target methods in our dataset. There are two CUTs that directly invoke `encode` with one original argument each. PROZE generates 12 PUTs from these 2 CUTs. Each PUT takes an `int` parameter that is supplied values from the argument provider method, which includes these 2,694 captured arguments. PROZE augments the input space coverage of `encode` by three orders of magnitude.

The results obtained for the two modules, `query` and `sso`, of WSO2-SAML are given in the next set of rows in Table II. Within WSO2-SAML, the maximum number of arguments is observed for the target method `unmarshall` which is called directly by 2 CUTs `testMarshall` and `testUnmarshall`, with 1 original argument each. PROZE generates 3 PUTs for `umarshall`, along with a data provider that generates 35 stringified XMLs. Overall, PROZE generates 8 PUTs for the `query` module and 109 PUTs for the `sso` module, with their argument providers having a median of 3 and 6 arguments, respectively.

RQ1: PUT arguments

PROZE captures 13,021 arguments at runtime for the 128 methods across our five study subjects. These arguments are encapsulated within argument provider methods for the 2,287 generated PUTs. Invoking the target methods with these arguments within the PUTs increases their input space coverage, sometimes up to several orders of magnitude.

B. RQ2: PUT classification

To answer this RQ, we execute each of the 2,287 PUTs generated by PROZE, and delve into our PUT classification protocol described in subsection IV-C. For the five modules, Table III presents the number of PUTs that pass at least with the original arguments, and fall into each of the three categories, strongly-coupled, falsifiably-coupled, and decoupled. We disregard 373 ill-formed PUTs per subsection III-D for this RQ.

TABLE III
RQ2: PUT CLASSIFICATION AS STRONGLY-COUPLED, FALSIFIABLY-COUPLED, OR DECOUPLED WITH THE TARGET METHOD INVOCATION.

MODULE	PUTS	STRONGLY -COUPLED	FALSIFIABLY -COUPLED	DECOUPLED
fontbox	135	31	12	92
xmpbox	543	94	40	409
pdfbox	1,150	510	161	479
query	7	0	0	7
sso	79	51	4	24
TOTAL	**1,914**	**686**	**217**	**1,011**

Let us consider a CUT from xmpbox called testBagManagement (lines 1 to 17 of Listing 4). This CUT invokes the target method createText with an argument containing 4 String values (line 8), and contains 4 assertions (lines 11, 12, 15, and 16). The original argument for createText within this CUT is null, "rdf", "li", "valueOne". In addition to this original argument, PROZE captures 737 unique arguments with which createText is invoked at runtime. Using this union of captured arguments, PROZE generates the argument provider method provideCreateTextArgs (lines 31 to 39 in Listing 4). Furthermore, PROZE generates 4 PUTs for testBagManagement, one for each of its 4 assertions. Since the PUTs are parameterized over createText, they accept 4 String values (line 23) from the argument provider method. For brevity, we showcase an excerpt that is common to these 4 PUTs (lines 20 to 29), specifically the invocation of createText with the PUT parameters on line 25. Note that each of the 4 generated PUTs contains a single assertion from the original CUT.

strongly-coupled cases: The two PUTs containing the first and third assertion of testBagManagement (lines 11 and 15, respectively in Listing 4) only pass when the argument they receive from provideCreateTextArgs is the original argument (on line 34). We classify them as strongly-coupled to the original argument of createText. Per Table III, we find 686 such PUTs for which the original argument is critical for the assertion to pass. In other words, the oracle is completely bound to the original test input.

decoupled case: Next, the PUT containing the assertion on line 12 of testBagManagement holds over all 738 arguments provided by provideCreateTextArgs in the generated parameterized unit test. We categorize this PUT as decoupled from the argument of createText, as the invocation of createText with all arguments pass. This means PROZE is not able to prove that the argument propagates to the assertion, and that the oracle is coupled to the PUT

```
1 @Test
2 public void testBagManagement() {
3   XMPMetadata parent =
        XMPMetadata.createXMPMetadata();
4   XMPSchema schem = new XMPSchema(parent, "nsURI",
        "nsSchem");
5   String bagName = "BAGTEST";
6   String value1 = "valueOne";
7   String value2 = "valueTwo";
8   schem.addBagValue(bagName,
        schem.getMetadata().getTypeMapping()
        .createText(null, "rdf", "li", value1));
9   schem.addQualifiedBagValue(bagName, value2);
10  List<String> values =
        schem.getUnqualifiedBagValueList(bagName);
11  assertEquals(value1, values.get(0));
12  assertEquals(value2, values.get(1));
13  schem.removeUnqualifiedBagValue(bagName, value1);
14  List<String> values2 =
        schem.getUnqualifiedBagValueList(bagName);
15  assertEquals(1, values2.size());
16  assertEquals(value2, values2.get(0));
17 }
18 ..................................................................

20 // Generated PUT parameterized over createText
21 @ParameterizedTest
22 @MethodSource("provideCreateTextArgs")
23 public void testBagManagement_PUT_N(String ns,
       String prefix, String propName, String value) {
24   ...
25   schem.addBagValue(bagName,
         schem.getMetadata().getTypeMapping()
         .createText(ns, prefix, propName, value));
26   ...
27   // one of lines 11, 12, 15, or 16 per PUT
28   ...
29 }

31 // Argument provider with 738 captured inputs
32 static Stream<Arguments> provideCreateTextArgs() {
33   return java.util.stream.Stream.of(
34     Arguments.of(null, "rdf", "li", "valueOne"),
35     ...
36     Arguments.of(null, "pdf", "PDFVersion", "1.4"),
37     Arguments.of("nsURI", "nsSchem", "li",
           "valueTwo")
38   );
39 }
```

Listing 4. The CUT testBagManagement in xmpbox has 4 assertions, and calls target method createText. PROZE captures 738 arguments for createText, and generates the argument provider method provideCreateTextArgs. This provider supplies arguments to 4 generated PUTs parameterized over createText. The parts common to the 4 generated PUTs are included in testBagManagement_PUT_N.

inputs. In Table III, we report 1,011 such PUTs that contain an assertion that is likely independent of the arguments passed to the target method. To be conservative, we assume that all of them are in this situation, but we note that it is theoretically possible that the assertion still verifies an interesting behavior over all inputs passed to the PUT (cf. Listing 1).

falsifiably-coupled case: Finally, the PUT containing the assertion on line 16 of `testBagManagement` is valid for the original argument, and for one additional argument, `"nsURI", "nsSchem", "li", "valueTwo"` (line 37 of Listing 4), but invalid for the other ones. This makes this PUT falsifiably-coupled to the target method `createText`, with a proof that the oracle is coupled to the PUT input. There are 217 such PUTs for which PROZE has discovered at least one additional valid input by monitoring runtime executions. We find the maximum increase in input space coverage for the target method `encode` in `pdfbox`, for which PROZE generates 12 PUTs (Table II), of which 10 are falsifiably-coupled to the arguments of `encode`, each passing for 2,692 of the 2,694 captured inputs.

Falsifiably-coupled PUTs are ready to be turned into PUTs that can be added in the original test suite. These PUTs include an assertion that demonstrably assesses the behavior of the target method, with an argument provider that feeds realistic values to the PUT. In PROZE, the final steps to make a falsifiably-coupled PUT ready to be incorporated into the suite are: (i) limit the provider to deliver only valid arguments, by removing the falsifying arguments from the initial provider; (ii) if multiple PUTs use the same argument providers, merge the assertions of these different PUTs into one PUT.

RQ2: PUT classification

PROZE generates 217 PUTs that include an assertion that is falsifiably-coupled to the arguments supplied to a target method, i.e., it is proven to successfully discriminate the behavior of the target method depending on the arguments passed to the PUT. To our knowledge, PROZE is the first technique that is able to generate PUTs using realistic inputs with the guarantee of a valid oracle.

C. RQ3: Test representativeness

Table IV summarizes our insights from the analysis of the PUTs generated by PROZE with respect to those generated by [7], [11], and [36].

With the exception of BASILISK [11] which focuses on C programs, all the studies do PUT generation for Java projects. Per the INPUT row, PROZE and MR-SCOUT [36] are the only techniques that use developer-written CUTs from the existing test suite of a project as the seed for their PUT generation efforts. This guarantees that developer intentions propagate to the generated PUTs, making them understandable for developers. The TEST INPUTS row presents the methodology employed by each technique to provide multiple inputs to the PUTs. The PUTs generated by Fraser and Zeller [7] include symbolically generated inputs, while BASILISK uses random inputs generated from fuzzing, and MR-SCOUT uses inputs generated through search-based techniques. PROZE is novel in this respect, as it captures actual usages of target methods during test and field executions. PUTs have previously not been automatically generated with realistic test inputs collected at runtime. The following row presents the kind of TEST ORACLE included in the generated tests. MR-SCOUT and PROZE are the only techniques that reuse developer-written assertions in the generated PUTs. Preserving the developers' assertions as part of the generated PUTs contributes to their readability with respect to the test intention. Likewise, the tests generated by MR-SCOUT and PROZE inherit TEST SCENARIOS from original developer-written CUTs. The goal of a codified MR generated by MR-SCOUT is to express a metamorphic relation that exists between the initial and subsequent states of an object. On the other hand, within a PROZE PUT, an existing assertion is evaluated against a larger set of inputs than in the original CUT. The concept of extending the range of an oracle over more inputs within a PUT is both new and powerful. PROZE is also the only technique to generate PUTs that are native JUnit5 (Listing 3 and Listing 4) and TestNG parameterized tests, the two most popular TEST FRAMEWORKS in Java [14]. The invocation of the PUT with distinct arguments is fully compatible with existing test runners. Compared to the three most related techniques for PUT generation, we argue that the PUTs generated by PROZE are more representative of real-world tests and testing practices. First, PROZE transforms a developer-written CUT into a PUT for a target method, which guarantees that the PUT represents a real-world test scenario. Second, the inputs to the PUT are concrete values captured at runtime, and supplied to the PUT explicitly through an argument provider method. Finally, the assertion in the PUT is not implicit or inferred, but explicit and adopted directly from the developer-written CUT, capturing a clear and scoped test intention.

RQ3: PUT representativeness

PROZE derives PUTs from developer-written CUTs, which guarantees that the PUTs have a clear test intention. PROZE exploits best-in-class testing frameworks, and the generated tests are representative of current testing practices. The PROZE PUTs are the first ever to be informed by runtime data from test and field executions.

D. Threats to validity

A threat to the internal validity of our evaluation arises from the implementation of PROZE, which may contain bugs. The limited number of study subjects [13] in our evaluation has implications on the external validity of our findings.

VI. RELATED WORK

Since they were first proposed by Tillmann and Schulte [26], PUTs have received considerable attention in the literature. The empirical study on open-source .NET projects by Lam *et al.* [16] identifies common implementation patterns and code

TABLE IV
RQ3: QUALITATIVE ANALYSIS OF PROZE IN THE TESTING PROCESS, COMPARED TO THREE PREVIOUS STUDIES ON AUTOMATED PUT GENERATION

	FRASER & ZELLER, 2011 [7]	BASILISK, 2019 [11]	MR-SCOUT, 2024 [36]	PROZE
LANGUAGE	Java	C	Java	Java
INPUT	source class	manually-written system tests + system tests generated with fuzzing	developer-written test suite	developer-written test suite + workload
TEST INPUTS	symbolic preconditions through mutation	random values from fuzzing system test inputs	instances of class under test through search-based approach	union of arguments from test and field invocations
TEST ORACLE	symbolic postconditions	implicit assertion (failure), increase in branch coverage	assertion with metamorphic relation (MR)	developer-written assertion
TEST SCENARIO	optimized subset of relevant assumptions, sequence of method calls, assertions	carved from system test	multiple invocations of methods from the same class with different arguments	derived from developer-written CUT
TEST FRAMEWORK	`assume`, `assert` clauses	functions in C	parameterized methods called by JUnit5 tests	full-fledged JUnit5 + TestNG tests
EXAMPLE	Section 5.4 of [7]	running example in [11]	Figure 4 of [36]	Listing 3, Listing 4

smells in PUTs written with the PEX framework [25]. Several studies have also proposed techniques for automatically generating PUTs. UnitMeister and AxiomMeister [24] use symbolic execution techniques for generalizing CUTs in order to produce a new set of PUTs [27]. The technique proposed by Thummalapenta and colleagues [23] for retrofitting CUTs into PUTs requires significant manual effort to create generalized oracles that are valid over all inputs. Compared to these studies, PROZE is fully automated, it does not require manual intervention to generalize the oracle or to generate PUTs.

Tsukamoto *et al.* [31] propose AutoPUT to automate the retrofitting of PUTs from developer-written CUTs. AutoPUT identifies a set of CUTs with similar Abstract Syntax Trees (ASTs) and consolidates them into a single PUT implemented as JUnit4 `@Theory` methods. A similar approach is employed by Azamnouri *et al.* [2] to convert CUTs into PUTs as a means of compressing the test suite. Menéndez *et al.* [18] propose the concept of diversified focused testing, which involves generating parameterized test data generators. Unlike these approaches, our goal with PROZE is not to compress or diversify the test suite. We aim to discover CUT oracles that are well-suited for PUTs, thereby leveraging the substantial benefits that PUTs offer over CUTs.

Theory-based [20] and property-based [17, 10] testing are closely related concepts to PUTs [32]. A theory or a property is a general oracle that holds for any input data. A key practical challenge developers face with these kinds of tests is generating realistic input data similar to the ones they would observe in the real world [9]. In contrast, PUTs oracles do not apply universally to all possible inputs. They are specific to the selected set of realistic inputs provided by the argument provider. Moreover, PROZE generates argument providers that include data which both satisfy and falsify the oracle.

A key novelty of our work is our focus on realistic data captured from test and field executions. Several studies employ runtime data for CUT generation [33, 29, 28, 1], in order to increase the input space coverage and representativeness of the test inputs. Yet, there is no literature on using runtime data in the context of PUTs.

VII. CONCLUSION

We introduce PROZE, a novel technique to automatically transform CUTs into PUTs, leveraging original developer-written oracles. To our knowledge, this is the first study on the automated generation of PUTs using realistic data observed during both test and field executions. Our evaluation of PROZE on five Java modules from two widely-used open-source Java projects demonstrates that PROZE successfully handles real-world tests. PROZE captures 13,021 runtime arguments for 128 target methods during the test and field executions of these five applications. Consequently, PROZE generates 2,287 PUTs from the 248 CUTs that directly invoke these target methods. Our results show that PROZE significantly enhances the input space coverage for these methods. Notably, 217 PUTs are ready to be used in the projects' test suite, as they have a strong oracle that holds for a wide range of test inputs, and that can discriminate the behavior of the target method under valid and invalid inputs.

We envision that the argument providers of parameterized unit tests can include calls to faking libraries [4]. These libraries provide reusable data for test inputs, such as names, addresses or *lorem ipsum*s. We believe they can also be used to increase the coverage of input spaces, including with humorous data [30]. In the future, we wish to assess the effectiveness of realistic runtime data and humorous fakes on the overall quality of test suites.

ACKNOWLEDGEMENTS

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Chains project funded by the Swedish Foundation for Strategic Research (SSF), IVADO and NSERC.

References

[1] Mehrdad Abdi and Serge Demeyer. “Test Transplantation through Dynamic Test Slicing”. In: *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2022.

[2] Aidin Azamnouri and Samad Paydar. “Compressing Automatically Generated Unit Test Suites Through Test Parameterization”. In: *Proceedings of FSEN 2021*. 2021.

[3] Earl T Barr et al. “The oracle problem in software testing: A survey”. In: *IEEE transactions on software engineering* 41.5 (2014).

[4] Benoit Baudry et al. “Generative AI to Generate Test Data Generators”. In: *IEEE Software* (2024).

[5] Simon Butler et al. “Maintaining interoperability in open source software: A case study of the Apache PDFBox project”. In: *Journal of Systems and Software* 159 (2020).

[6] Sebastian Elbaum et al. “Carving and Replaying Differential Unit Test Cases from System Test Cases”. In: *IEEE Transactions on Software Engineering* 35.1 (2009).

[7] Gordon Fraser and Andreas Zeller. “Generating parameterized unit tests”. In: *Proceedings of the 2011 international symposium on software testing and analysis*. 2011.

[8] Simson Garfinkel et al. “Bringing science to digital forensics with standardized forensic corpora”. In: *digital investigation* 6 (2009).

[9] Harrison Goldstein et al. “Property-Based Testing in Practice”. In: *Proceedings of ICSE*. 2024.

[10] Zac Hatfield-Dodds. “Falsify your Software: validating scientific code with property-based testing.” In: *SciPy*. 2020.

[11] Alexander Kampmann and Andreas Zeller. “Carving parameterized unit tests”. In: *Proceedings of ICSE-Companion*. 2019.

[12] Katja Karhu et al. “Empirical observations on software testing automation”. In: *Proceedings of ICST*. 2009.

[14] Dong Jae Kim et al. “Studying test annotation maintenance in the wild”. In: *Proceedings of ICSE*. 2021.

[15] Rick Kuhn et al. “Input Space Coverage Matters”. In: *Computer* 53.1 (2020).

[16] Wing Lam et al. “A Characteristic Study of Parameterized Unit Tests in. NET Open Source Projects”. In: *Proceedings of ECOOP*. 2018.

[17] David R MacIver, Zac Hatfield-Dodds, et al. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (2019).

[18] Héctor D Menéndez et al. “Diversifying focused testing for unit testing”. In: *ACM Transactions on Software Engineering and Methodology* 30.4 (2021).

[19] Renaud Pawlak et al. “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”. In: *Software: Practice and Experience* 46 (2015).

[20] David Saff, Marat Boshernitsan, and Michael D. Ernst. *Theories in practice: Easy-to-write specifications that catch bugs*. Tech. rep. MIT Computer Science and Artificial Intelligence Laboratory, 2008.

[21] Michael Schwartz et al. “SAML”. In: *Securing the Perimeter: Deploying Identity and Access Management with Free Open Source Software* (2018).

[22] Elvys Soares et al. “Refactoring test smells with junit 5: Why should developers keep up-to-date?” In: *IEEE Transactions on Software Engineering* 49.3 (2022).

[23] Suresh Thummalapenta et al. “Retrofitting unit tests for parameterized unit testing”. In: *Proceedings of FASE*. 2011.

[24] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. “Discovering likely method specifications”. In: *Proceedings of ICFEM*. 2006.

[25] Nikolai Tillmann and Jonathan De Halleux. “Pex–white box test generation for. net”. In: *Proceedings of TAP*. 2008.

[26] Nikolai Tillmann and Wolfram Schulte. “Parameterized unit tests”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005).

[27] Nikolai Tillmann and Wolfram Schulte. “Unit tests reloaded: Parameterized unit testing with symbolic execution”. In: *IEEE software* 23.4 (2006).

[28] Deepika Tiwari, Martin Monperrus, and Benoit Baudry. “Mimicking production behavior with generated mocks”. In: *arXiv preprint arXiv:2208.01321* (2022).

[29] Deepika Tiwari et al. “Production Monitoring to Improve Test Suites”. In: *IEEE Transactions on Reliability* 71.3 (2022).

[30] Deepika Tiwari et al. “With great humor comes great developer engagement”. In: *Proceedings of ICSE: Software Engineering in Society*. 2024.

[31] Keita Tsukamoto, Yuta Maezawa, and Shinichi Honiden. “AutoPUT: an automated technique for retrofitting closed unit tests into parameterized unit tests”. In: *Proceedings of SAC*. 2018.

[32] Vasudev Vikram, Caroline Lemieux, and Rohan Padhye. “Can large language models write good property-based tests?” In: *arXiv preprint arXiv:2307.04346* (2023).

[33] Qianqian Wang, Yuriy Brun, and Alessandro Orso. “Behavioral execution comparison: Are tests representative of field behavior?” In: *Proceedings of ICST*. 2017.

[34] Robert White and Jens Krinke. “TCTracer: Establishing test-to-code traceability links using dynamic and static techniques”. In: *Empirical Software Engineering* 27.3 (2022).

[35] Tao Xie et al. “Mutation analysis of parameterized unit tests”. In: *Proceedings of ICST Workshops*. 2009.

[36] Congying Xu et al. “MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases”. In: *ACM Transactions on Software Engineering and Methodology* (2024).

Mimicking Production Behavior with Generated Mocks

Deepika Tiwari, Martin Monperrus, and Benoit Baudry
KTH Royal Institute of Technology, Stockholm, Sweden

{deepikat, monperrus, baudry}@kth.se

Abstract—Mocking allows testing program units in isolation. A developer who writes tests with mocks faces two challenges: design realistic interactions between a unit and its environment; and understand the expected impact of these interactions on the behavior of the unit. In this paper, we propose to monitor an application in production to generate tests that mimic realistic execution scenarios through mocks. Our approach operates in three phases. First, we instrument a set of target methods for which we want to generate tests, as well as the methods that they invoke, which we refer to as mockable method calls. Second, in production, we collect data about the context in which target methods are invoked, as well as the parameters and the returned value for each mockable method call. Third, offline, we analyze the production data to generate test cases with realistic inputs and mock interactions. The approach is automated and implemented in an open-source tool called RICK. We evaluate our approach with three real-world, open-source Java applications. RICK monitors the invocation of 128 methods in production across the three applications and captures their behavior. Based on this captured data, RICK generates test cases that include realistic initial states and test inputs, as well as mocks and stubs. All the generated test cases are executable, and 52.4% of them successfully mimic the complete execution context of the target methods observed in production. The mock-based oracles are also effective at detecting regressions within the target methods, complementing each other in their fault-finding ability. We interview 5 developers from the industry who confirm the relevance of using production observations to design mocks and stubs. Our experimental findings clearly demonstrate the feasibility and added value of generating mocks from production interactions.

Index Terms—Mocks, production monitoring, mock-based oracles, test generation

1 Introduction

Software testing is an expensive, yet indispensable, activity. It is done to verify that the system as a whole, as well as the individual modules that compose it, behave as expected. The latter is achieved through unit testing. When a unit interacts with others, or with external components, such as the file system, database, or the network, it becomes challenging to test it in isolation. As a solution to this problem, developers rely on a technique called mocking [1]. Mocking allows a unit to be tested on its own, by replacing dependent objects that are irrelevant to its functionality, with "fake" implementations. There are several advantages of mocking, such as faster test executions, fewer side-effects, and quicker fault localization [2].

Despite their advantages, using mocks within tests requires considerable engineering effort. Developers must first identify the components that may be replaced with mocks [3]. They must also determine how these mocks behave when triggered with a certain input, i.e., how they are stubbed [4], [5]. In order to address these challenges, several techniques have been developed to automatically generate mocks. For example, mocks have been generated through search-based algorithms by including contextual information, such as the state of the environment, in the search space for input data generation [6], [7]. Dynamic symbolic execution has been used to define the behavior of mocks through generated mock classes [8]. System test executions can be monitored to derive unit tests that use mocks [9]. This preliminary research has validated the concept of mock generation. However, the test inputs and the mocks produced by these techniques are either synthetic or manually written by developers per their assumptions of how the system behaves. These approaches do not guarantee that the generated mocks reflect realistic behaviors as observed in production contexts.

The fundamental premise of mocking is to replace a real object with a fake one that mimics it [10], [11]. This implies that, for it to be useful, the behavior of the mock should resemble, as closely as possible, that of a real object [12]. Our key insight is to derive realistic behavior from real behavior, i.e., to generate mocks from production usage. This builds upon the fact that 1) the behavior of an application in production is the reference one [13], and 2) this production behavior can be used for test generation [14], [15]. Given a set of methods of interest for test generation, we monitor them in production. As a result of this monitoring, we capture realistic execution contexts to generate tests for each target method, where external objects are replaced with mocks, and stubbed based on production states.

In this paper, we propose RICK, an approach for mimicking production behavior with generated mocks. RICK monitors an application executing in production with the goal of generating valuable tests. The intention of the generated tests is to verify the behavior of the methods, where the reference behavior captured in the oracle is the one from production. The interactions of this method with other external objects are mocked. Within each generated

test, the data captured from production is expressed as rich serialized test inputs. Each generated test has a mock-based oracle, which verifies distinct aspects of the invocation of the target method and its interactions with mock methods, such as the object returned from the target method, the parameters with which the mock methods are called, as well as the frequency and sequence of these mock method calls.

Our approach for the synthesis of mocks is based on two fundamentally novel concepts. First, the mocks are stubbed with data and interactions observed in production. Second, our three mock-based oracles enable powerful behavior verification. A key benefit of this latter point is that, in addition to checking the output of a method directly with a straightforward assertion, we also verify the actions that should occur within the method.

We evaluate the capabilities of RICK with three open-source Java applications: a map-based routing application called GRAPHHOPPER, a feature-rich graph analysis and visualization tool called GEPHI, and a utility library for working with PDF documents called PDFBOX. We target 212 methods across the three applications, which get invoked as the applications execute in typical production scenarios. RICK generates 294 tests for 128 of these methods. Within each generated test, RICK recreates execution states that mimic production ones with objects that range in size from 37 bytes to 39 megabytes. When we execute the tests, we find that 68 of the 128 methods (53.1%) have at least one passing test that recreates real usage conditions, and 154 of the 294 (52.4%) tests successfully recreate the complete production execution context. These results indicate that RICK is capable of monitoring applications in production, capturing realistic behavior for target methods, and transforming it into tests that mimic the behavior of the methods, while isolating it from its interactions with external objects. Furthermore, through mutation analysis, we determine that the generated tests are effective at detecting regressions within the target methods. The mock-based oracles contained in the generated tests complement each other with respect to their ability to detect bugs. To assess the quality of the RICK-generated tests, we interview 5 software developers from different sectors of the IT industry. All of them find the collection of production values to be relevant to generate realistic mocks. Moreover, they appreciate the structure and the understandability of the tests generated by RICK.

To sum up, the key contributions of this paper are:

- The novel concept of the automated generation of mocks, stubs, and oracles using data collected from production.
- A comprehensive methodology for generating tests that mimic complex production interactions through mocks, by capturing receiving objects, parameters, returned values, and method invocations, for a method under test, while an application executes.
- An evaluation of the approach on 3 widely-used, large, open-source Java applications, demonstrating the feasibility and benefits of generating mocks.
- A publicly available tool called RICK implementing the approach for Java, and a replication package for future research on this novel topic[1].

1. https://zenodo.org/doi/10.5281/zenodo.6914463

The rest of the paper is organized as follows. section 2 presents the background on mocking and mock objects. We describe how RICK generates tests and mocks in section 3. Next, section 4 discusses the methodology we follow for our experiments, applying RICK to real-world Java projects, followed by the results of these experiments in section 5. section 7 presents closely related work, and section 8 concludes the paper.

2 BACKGROUND

This section summarizes the key concepts of mock objects, and how they are used in practice. We also discuss the challenges of using mocks within tests.

2.1 Mock Objects

Software comprises of individual modules or units. These units interact with each other, as well as with external libraries, for example, to send emails, transfer data over the network, or perform database operations. This facilitates modular development, as different teams can work in parallel on implementing distinct functionalities of the system. The modules are then composed together, in order to achieve use cases. Yet, a disadvantage of this coupling is that testing each unit in isolation from others is not straightforward. *Mocking* was proposed as a solution to this problem [1]. It is a mechanism to replace real objects with skeletal implementations that mimic them [2]. Mocking allows individual functionalities to be tested independently. The process of unit testing with mocks is faster and more focused [3]. Since the test intention is to verify the behavior of one individual unit, mocking can facilitate fault localization. External objects, with interactions that are complex to set up within a test, can be replaced with mocks [16]. Furthermore, a test can be made more reliable by using mocks to replace external, potentially unreliable or non-deterministic, components [6], [7]. Mock objects typically behave in specific, pre-determined ways through a process called *stubbing* [5], [10]. For example a method called `getAnswer` invoked on a mock object can be stubbed to return a value of `42`, without the actual invocation of `getAnswer`. Stubbing can be very useful for inducing behavior that may be hard to produce locally within the test, such as error- or corner-cases. Mocking can also be used for *verifying* object protocols [17]. For example, consider a method called `subscribeToNewsletter`, which should call another method `sendWelcomeEmail` on an object of type `EmailService`. Developers can mock the `EmailService` object within the test for `subscribeToNewsletter`, to verify that the method `sendWelcomeEmail` is indeed invoked on `EmailService` exactly once, with a parameter of type `UserID`. This interaction is therefore verifiable without side-effects, i.e., without an actual email being sent.

In short, the three key concepts of testing with mocks are *Mocking*, *Stubbing*, and *Verifying*. Real objects can be replaced with fake implementations called mocks. These mocks can be stubbed to define tailor-made behaviors, i.e., produce a certain output for a given input. Moreover, the interactions made with the mocks can be verified, such as the number of times they were triggered with a given input, and in a specific sequence.

```
1 public class ReservationCentre {
2   ...
3   // Target method
4   public void purchaseTickets(int quantity, PaymentService
        paymentService) {
5     ...
6     double amount = basePrice * quantity;
7     ...
8     // Mockable method call #1
9     if (paymentService.checkActiveConnections() > 0) {
10      ...
11      // Mockable method call #2
12      boolean isPaymentSuccessful =
            paymentService.processPayment(amount);
13    ...
14    }
15    ...
16    return ...;
17  }
18  ...
19 }
```

Listing 1: Target method `purchaseTickets` has mockable method calls on the `PaymentService` object

```
1 @Test
2 public void testTicketPurchasing() {
3   ReservationCentre resCentre = new ReservationCentre();

5   // Mock external types
6   PaymentService mockPayService = mock(PaymentService.class);

8   // Stub behavior
9   when(mockPayService.checkActiveConnections()).thenReturn(1);
10  when(mockPayService.processPayment(42.24)).thenReturn(true);

12  resCentre.purchaseTickets(2, mockPayService);

14  // Verify invocations on mocks
15  verify(mockPayService, times(1)).checkActiveConnections();
16  verify(mockPayService, times(1)).processPayment(anyDouble());
17 }
```

Listing 2: A test for the `purchaseTickets` method which mocks `PaymentService`

2.2 The Practice of Testing with Mocks

Mocks can be implemented in several ways. For example, developers may manually write classes that are intended as replacements for real implementations [18]. However, a more common way of defining and using mocks is through the use of mocking libraries, which are available for most programming languages. Mockito[2] is one of the most popular mocking frameworks for Java, both in the industry and in software engineering research [19], [20]. It can be integrated with testing frameworks such as JUnit and TestNG, allowing developers to write tests that use mocks.

Let us consider the example of the method `purchaseTickets` presented in Listing 1. This method handles the purchase of tickets, including the interaction with the payment gateway, `PaymentService`. It is defined in the `ReservationCentre` class, and takes two parameters. The first parameter is an integer value for the `quantity` of tickets, and the second parameter is the object `paymentService` of the external type `PaymentService`. Two methods are called on the `paymentService` object: `checkActiveConnections` on line 9, and `processPayment` on line 12 which accepts a parameter of type `double`. We illustrate the use of mocks, stubs, and verification through the unit test for `purchaseTickets` presented in Listing 2. The intention of this test, `testTicketPurchasing`, is to verify the behavior of `purchaseTickets`, while mocking its interactions with `PaymentService`. First, the receiving object `resCentre` of type `ReservationCentre` is set up (line 3). Next, `PaymentService` is mocked, through the `mockPayService` object (line 6). Lines 9 and 10 stub the two methods called on this mock: `checkActiveConnections` is stubbed to return a value of `1`, and `processPayment` is stubbed to return `true` when invoked with the `double` value `42.24`. Finally, on line 12, `purchaseTickets` is called with the `quantity` of 2, and the mocked parameter `mockPayService`. The statements on lines 15 and 16 verify that this invocation of `purchaseTickets` calls `checkActiveConnections` exactly once, and `processPayment` exactly once, with a double value (specified using `anyDouble()`). Thus, this test verifies the behavior of the target method, `purchaseTickets`, isolating it from the interactions with a real `PaymentService` object. Moreover, method calls on `PaymentService` are stubbed so that `purchaseTickets` gets executed as it normally would, without the side-effect of an actual payment being made. This allows for more focus on the method under test, `purchaseTickets`.

2.3 The Challenges of Mocking

Despite the benefits of mocking that we highlight, it is not trivial to incorporate mocks in practice. Deciding what to mock, and how the mocks should behave, is hard [11]. For example, developers would first identify that `PaymentService` may be mocked within the test for `purchaseTickets` in Listing 2. Next, they must also manually define concrete values for the parameters and returned values for stubbing the calls made on this mock, in order trigger a specific path through `purchaseTickets`. Additionally, they would have to determine which interactions made on this mock are verifiable. It is also challenging to decide between conventional object-based testing and mock-based testing. Because of these challenges, developers are hesitant to incorporate mocks within their testing practice, as highlighted by a study by Spadini *et al.* [19], who found that mocks are most likely to be introduced at the time a test class is first written. This suggests the potential opportunity and benefits of automated mock generation throughout the development lifecycle. The results from our developer study in RQ5 address this aspect.

To address the challenges of manually implementing mocks, several studies propose methodologies for their automated generation, such as through search-based algorithms [21] or symbolic execution [22]. However, none of these studies use data from production executions to do so. In this work, we propose to monitor an application in production, with the goal of generating tests with mocks. These tests use mocks to 1) isolate a target method from external units, and 2) verify distinct aspects of the behavior with oracles specific to mocks.

3 Mock Generation with RICK

We introduce RICK, a novel approach for automatically generating tests with mock objects, using data collected during the execution of an application. In production, RICK

2. https://site.mockito.org/

collects realistic data for recreating the program states for the method under test, as well as the parameters and values returned by methods called on external objects. In subsection 3.1, we present an overview of the RICK pipeline. This is followed in subsection 3.2 by a discussion of the kinds of oracles produced by RICK. Next, subsection 3.3 motivates the design decisions of RICK and highlights its key features. We discuss in subsection 3.4 how RICK can be useful in the software development lifecycle. Finally, subsection 3.5 presents technical details of its implementation.

3.1 Overview

RICK operates in three phases. We illustrate them in Figure 1. In the first phase (1), RICK identifies test generation targets within an application. These targets are called methods under test, and they have mockable method calls. We define them as follows:

Methods under Test (MUTs): The target methods for test generation by RICK are called methods under test (MUTs). A method is considered as being an MUT if it invokes methods on objects of other types. The identification of MUTs forms the basis of the test generation effort, since the intention of each test generated by RICK is to verify the behavior of one MUT after isolating it from such external interactions.

Mockable method calls: We define a mockable method call as a method call nested within an MUT, that is made on a field or a parameter object whose type is different from the declaring type of the MUT. RICK will mock objects of types that are declared within the project, and not types from the standard library or dependencies. When RICK generates a test for the MUT, a mockable method call becomes a *mock method call*, i.e., the external object is replaced with a mock object, and the mockable method call occurs on this mock object.

Figure 1 illustrates the identification of an MUT (highlighted in yellow), together with its mockable method calls (shown here in green circles). In subsection 3.3 we detail how a nested method call qualifies as being mockable. As an example, consider the class `ClassUnderTest` presented in Listing 3. RICK identifies the method `methodUnderTestOne` (line 5) as an MUT. Moreover, the nested call to `mockableMethodOne` on line 8 within `methodUnderTestOne` is identified as a mockable method call because it is made on the field `extField` of `ClassUnderTest` (line 2), which is of an external type. Similarly, the method `methodUnderTestTwo` is also considered as an MUT by RICK, because it has two mockable methods called within it. The first is `mockableMethodTwo` called 42 times inside a loop (line 20), and the second one is `mockableMethodThree` (line 22). Both of these methods are called on `extParam`, which is an external parameter of `methodUnderTestTwo`.

The second phase (2) of RICK occurs when the application is deployed and running. During this phase, RICK monitors the invocation of the MUTs identified in the previous phase, and collects data corresponding to these invocations. By construction, the monitoring data reflects real interactions by end-users. Moreover, for inadequately-tested applications, it may represent usage scenarios that are not well tested by developer-written tests [13], [15].

RICK collects data for an invocation in production with the end goal of recreating the same invocation within a generated test. This data includes the parameters and returned values for each MUT and its corresponding mockable method calls, as well as the object on which the MUT is invoked, which we refer to as the *receiving object*. Figure 1 depicts the second phase, where the monitor defined within RICK is attached to both an MUT as well as its mockable method calls, in order to collect data about their invocations. subsection 3.3 presents more details on this monitoring.

Finally, in the third phase (3), RICK uses the data collected in the second phase as inputs to generate tests with mocks, as illustrated in Figure 1. These tests are designed to recreate the invocation of the MUTs, and verify their behavior as was observed in production, while simulating the interactions of the MUTs with external objects using mocks and stubs. They can serve as regression tests, and can also potentially lead to faster fault localization because they isolate the invocation of the MUT from the mockable method calls. For example, Listing 4 presents one test generated for the MUT `methodUnderTestOne`. This test verifies the observed behavior of `methodUnderTestOne` through the assertion on line 17, while mocking the `ExternalTypeOne` object (line 8). Within the test, the mockable method `mockableMethodOne` becomes the mock method, and is stubbed on line 11 using its parameter and returned value captured from production. We present more information on how RICK processes production data to generate tests in subsection 3.3. The generated tests are the final output of the RICK pipeline. Each generated test falls under one of three categories, determined by the kind of oracle within it. We discuss these categories in subsection 3.2.

3.2 Mock-based Oracles

The oracle in a unit test specifies a behavior that is expected as a consequence of running the MUT with a specific test input [4]. In the context of the tests generated by RICK, the oracles in the generated test verify the behavior of the MUT, while isolating it from method calls to external objects made within the MUT, i.e., mockable method calls. This facilitates the decoupling of the MUT from its environment, and allows the focus of the testing to be on the MUT itself. Moreover, the behavior being verified in the generated tests, both for the MUT and the mockable method calls, is sourced from production. This means that through these generated tests, developers can verify how the system behaves for actual users.

There is no systematized knowledge on oracles for tests with mocks. To overcome this, we now define three categories of oracles, all implemented by RICK.

Output Oracle, OO: The first category of tests generated by RICK have an output oracle. This oracle verifies that the behavior of the MUT is the same as the one observed in production, despite the introduction of mock objects. Even though this oracle relies on regular assertions, we still consider it as a mock-based oracle, as it assesses the behavior of the MUT in the presence of mock objects. A failure in a test with an output oracle indicates a regression in the MUT, which may be caused by its interaction with the mockable method call. Listing 4 presents an example

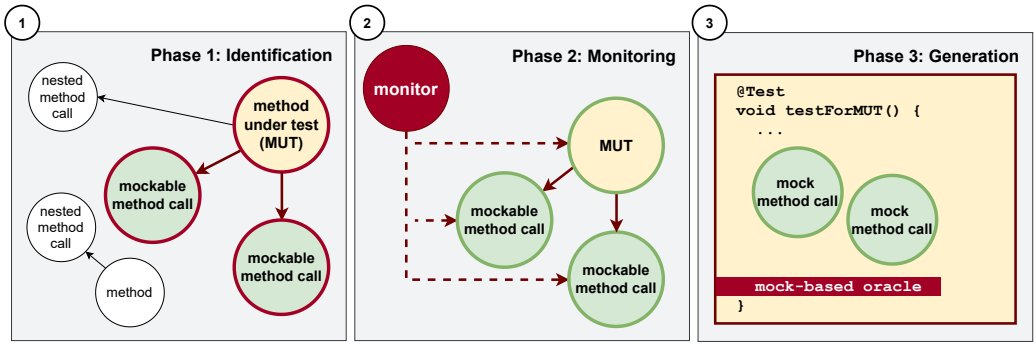


Fig. 1: The RICK test generation pipeline: offline, identify methods under test and mockable method calls; in production, monitor state and arguments for methods under test and mockable method calls; offline again, generate tests with mocks for the methods under test

```
1 class ClassUnderTest {
2  ExternalTypeOne extField;
3  ...

5  public int methodUnderTestOne(int param) {
6   ...
7   // mockable method call on field
8   int x = param + extField.mockableMethodOne(booleanVal);
9   // nested, non-mockable method call
10  int y = x + methodFour();
11  ...
12  return ...;
13 }

15 public int methodUnderTestTwo(double param,
16                      ExternalTypeTwo extParam) {
17  ...
18  // mockable method calls on parameter
19  for (int i = 0; i < 42; i++) {
20   listOfInts.add(extParam.mockableMethodTwo(floatVals[i]));
21  }
22  int z = extParam.mockableMethodThree(intVal);
23  ...
24  return ...;
25 }

27 private void methodThree() { ... }

29 public int methodFour() {
30  // nested, non-mockable method call
31  methodThree();
32  ...
33  return ...;
34 }
35 }
```

Listing 3: The class, `ClassUnderTest`, has four methods. RICK identifies two of these methods, `methodUnderTestOne` and `methodUnderTestTwo`, as MUTs as they have mockable method calls.

of a generated test with an output oracle. The MUT is `methodUnderTestOne`, and the mockable method call is `mockableMethodOne`. This test corresponds to one invocation of `methodUnderTestOne` observed by RICK in production. The test recreates the receiving object, `productionObj`, as it was observed in production, by deserializing it (line 5). Next, it mocks the field `extField` and injects it into `productionObj` (line 8). This is followed in line 11 by stubbing `mockableMethodOne`, to return a value of 27 when invoked with the parameter `true`, in accordance with the

```
1 @Test
2 public void testMethodUnderTestOne_OO() {
3 // Arrange
4  // Create test fixture from serialized production data
5  ClassUnderTest productionObj = deserialize(new File(
        "receiving1.xml"));

7  // Inject the mock
8  ExternalTypeOne mockExternalTypeOne =
       injectMockField_extField_InClassUnderTest();

10 // Stub the behavior
11 when(mockExternalTypeOne.mockableMethodOne(true))
       .thenReturn(27);

13 // Act
14  int actual = productionObj.methodUnderTestOne(17);

16 // Assert
17  assertEquals(42, actual);
18 }
```

Listing 4: A RICK test with an Output Oracle, **OO**, for the MUT `methodUnderTestOne`

observed production behavior of `mockableMethodOne`. On line 14, `methodUnderTestOne` is invoked on `productionObj` with the production parameter 17. Finally, the output oracle is the assertion on line 17, which verifies that the output from this invocation of `methodUnderTestOne`, with the stubbed call to `mockableMethodOne`, is 42, which is the value observed for this invocation in production.

Parameter Oracle, PO: The second category of generated tests have an oracle to verify that the mockable method calls occur with specific parameter(s), the same as production, within the invocation of the MUT. A test with a parameter oracle may fail due to regressions in the MUT which cause a mockable method call to be made with unexpected parameters. An example of this oracle is presented in Listing 5. This test recreates the receiving object `productionObj`, for the MUT `methodUnderTestTwo` (line 5). Next, it prepares a mock object for `ExternalTypeTwo` called `mockExternalTypeTwo` (line 8), and stubs the 42 invocations of the mockable method call, `mockableMethodTwo`. For brevity, we include only two of these stubs on lines 11 and 12. The single invocation of `mockableMethodThree`

```
1 @Test
2 public void testMethodUnderTestTwo_PO() {
3 // Arrange
4  // Create test fixture from serialized production data
5  ClassUnderTest productionObj = deserialize(new File(
         "receiving2.xml"));

7  // Create the mock
8  ExternalTypeTwo mockExternalTypeTwo = mock(
         ExternalTypeTwo.class);

10 // Stub the behavior
11 when(mockExternalTypeTwo.mockableMethodTwo(4.2F))
         .thenReturn(89);
12 when(mockExternalTypeTwo.mockableMethodTwo(9.8F))
         .thenReturn(92);
13 ...
14 when(mockExternalTypeTwo.mockableMethodThree(15))
         .thenReturn(48);

16 // Act
17 productionObj.methodUnderTestTwo(6.2, mockExternalTypeTwo);

19 // Assert
20 verify(mockExternalTypeTwo, atLeastOnce())
         .mockableMethodTwo(4.2F);
21 verify(mockExternalTypeTwo, atLeastOnce())
         .mockableMethodTwo(9.8F);
22 ...
23 verify(mockExternalTypeTwo, atLeastOnce())
         .mockableMethodThree(15);
24 }
```

Listing 5: A RICK test with a Parameter Oracle, **PO**, for the MUT `methodUnderTestTwo`

```
1 @Test
2 public void testMethodUnderTestTwo_CO() {
3 // Arrange
4  // Create test fixture from serialized production data
5  ClassUnderTest productionObj = deserialize(new File(
         "receiving2.xml"));

7  // Create the mock
8  ExternalTypeTwo mockExternalTypeTwo = mock(
         ExternalTypeTwo.class);

10 // Stub the behavior
11 when(mockExternalTypeTwo.mockableMethodTwo(4.2F))
         .thenReturn(89);
12 when(mockExternalTypeTwo.mockableMethodTwo(9.8F))
         .thenReturn(92);
13 ...
14 when(mockExternalTypeTwo.mockableMethodThree(15))
         .thenReturn(48);

16 // Act
17 productionObj.methodUnderTestTwo(6.2, mockExternalTypeTwo);

19 // Assert
20 InOrder orderVerifier = inOrder(mockExternalTypeTwo);
21 orderVerifier.verify(mockExternalTypeTwo, times(42))
         .mockableMethodTwo(anyFloat());
22 orderVerifier.verify(mockExternalTypeTwo, times(1))
         .mockableMethodTwo(anyInt());

24 }
```

Listing 6: A RICK test with a Call Oracle, **CO**, for the MUT `methodUnderTestTwo`

(line 14) is also stubbed, with the parameter and returned value observed in production. This is followed by a call to `methodUnderTestTwo` on `productionObj` (line 17), passing it `mockExternalTypeTwo` as parameter. Finally, the statements on lines 20 to 23 are unique to this category of tests, and serve as the parameter oracle. The statements on lines 21 and 22 verify that `mockableMethodTwo` is called at least once on `mockExternalTypeTwo` with the concrete production parameter `4.2F`, as well as with `9.8F`. We omit the verification of the other 40 invocations of `mockExternalTypeTwo` from this code snippet. Next, the parameter oracle verifies on line 23 that `mockableMethodThree` is called at least once with the parameter `15`, within this invocation of `methodUnderTestTwo`.

Call Oracle, CO: Oracles in the generated tests for the third category verify the sequence and frequency of mockable method calls within the invocation of the MUT. Any deviation from the expected sequence and frequency of mockable method calls within an MUT will cause a test with a call oracle to fail. This can be helpful for developers when localizing a regression related to object protocols within the MUT. An example of this oracle is presented in Listing 6. This test first recreates the receiving object, `productionObj`, for the MUT `methodUnderTestTwo` (line 5), stubs the mockable method calls to `mockableMethodTwo` and `mockableMethodThree` (lines 11 to 14), and invokes `methodUnderTestTwo` with the mocked parameter (line 17). Next, the call oracle in this test verifies the number of times the mockable method calls occur within this invocation of `methodUnderTestTwo`, as well as the order in which these calls occur. This is achieved with the order verifier defined on line 20. The statements on lines 21 and 22 verify that `mockableMethodTwo` is invoked exactly 42 times with a float parameter, and that these invocations are followed by one call to `mockableMethodThree` with an integer parameter, as was observed in production.

3.3 Key Phases

As outlined in subsection 3.1, RICK operates in three phases. We now discuss these three phases in more detail.

3.3.1 *Identification of Test Generation Targets*

It is not possible to generate test cases with mocks for all methods with nested method calls. For example, a static method invoked within another method is typically not mocked [3]. Also, it is not feasible to replace an object created within the body of a method, and subsequently mock the interactions made with it. Therefore, RICK includes a set of rules to determine the methods that can be valid targets for the generation of test cases and mocks. It is also possible for developers to provide an initial set of methods of interest, which RICK can consider.

3.3.1.1 Identifying MUTs: First, RICK finds methods that are part of the API of the application, i.e., methods that are public, non-abstract, non-deprecated, and non-empty [23], [24]. These criteria have also been used previously to generate differential unit tests for open-source Java projects [15]. Of these methods, RICK selects as MUTs the ones that invoke other methods on objects of external types.

3.3.1.2 Identifying Mockable Method Calls: Second, RICK identifies the nested method calls within each of the selected MUTs which could be mocked. An MUT may have several nested method calls, not all of which are suitable for mocking. For it to be mocked, a nested method call must be invoked on a parameter or a field, such that a mock can be injected to substitute it in the generated test. Next, the declaring type of this parameter or field should

be different from the type of the MUT, in order to represent an interaction of the MUT with an external type, per the theory of mocking external resources. RICK stubs methods that return a primitive or `String` value. Mocks are never returned from stubbed methods. Nested method calls that meet all these criteria, are marked as a *mockable method calls*.

We illustrate the rules for target selection with the help of the excerpt of the class `ClassUnderTest` in Listing 3. This excerpt includes a field as well as four methods defined in `ClassUnderTest`. The first method, `methodUnderTestOne` (lines 5 to 13), accepts an integer parameter, and returns an integer value. The body of `methodUnderTestOne` includes a call to the method `mockableMethodOne(boolean)`, on the field `extField` (line 8). This field is declared as being of type `ExternalTypeOne` in `ClassUnderTest` (line 2). There is another call on line 10 to a method defined in `ClassUnderTest` called `methodFour`. The second method, `methodUnderTestTwo` (lines 15 to 25), returns an integer value, and accepts two parameters. The first parameter is a double, and the second parameter called `extParam` is of type `ExternalTypeTwo`. Within the loop on lines 19 to 21, `methodUnderTestTwo` calls the method `mockableMethodTwo(float)` on the parameter `extParam` (line 20). There is another call on `extParam` to `mockableMethodThree(int)` (line 22). The third method defined in `ClassUnderTest` is a private method called `methodThree` (line 27). It does not call any other method. Finally, the fourth method in this excerpt is `methodFour` (lines 29 to 34), which has a call to `methodThree` (line 31).

As a consequence of the aforementioned criteria, RICK identifies `methodUnderTestOne` and `methodUnderTestTwo` as MUTs. Moreover, the nested method calls, `mockableMethodOne`, and `mockableMethodTwo` and `mockableMethodThree`, in these MUTs respectively, are recognized as mockable method calls by RICK. However, the call to `methodFour` within `methodUnderTestOne` is not mockable within `methodUnderTestOne`, and `methodThree` and `methodFour` are not MUTs since they do not fulfill these criteria.

3.3.2 Monitoring Test Generation Targets

Once it finds a set of MUTs and their corresponding mockable method calls, RICK instruments them in order to monitor their execution as the application runs in production. The goal of this instrumentation is to collect data as the application executes. RICK collects data about each invocation of an MUT: the receiving object, which is the object on which it is invoked, the parameters passed to the invocation, as well as the object returned from the invocation. At the same time, RICK collects data about the mockable method calls within this MUT. This includes the parameters and the returned value for each mockable method call. The data collected from this monitoring is serialized and saved to disk. For example, the sequence diagram in Figure 2 illustrates the monitoring of the MUT `methodUnderTestTwo` in `ClassUnderTest` presented in Listing 3. For one invocation of `methodUnderTestTwo`, RICK collects its receiving object, parameters, and returned value, as well as the parameters and returned values corresponding to the invocations of mockable method calls to `mockableMethodTwo` and `mockableMethodThree` within `methodUnderTestTwo`.

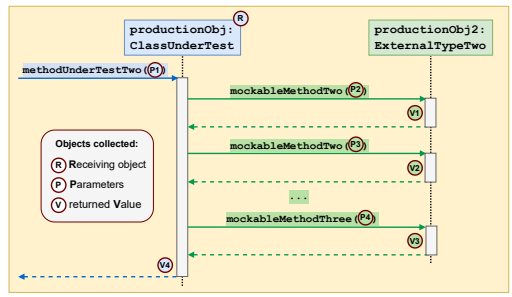


Fig. 2: Monitoring method invocations in production: RICK observes the invocation of the target method, `methodUnderTestTwo`, and captures the receiving object and parameters for this invocation, as well as the object returned from it. RICK also observes the method calls to `mockableMethodTwo` and `mockableMethodThree`, collecting their parameters and returned values.

We systematically consider special cases. First, a mockable method may be invoked from different MUTs. Also, an MUT may itself be a mockable method for another MUT. Moreover, an MUT may be invoked without its corresponding mockable method call(s), if the latter is invoked within a branch, for example. It is therefore important to ensure that the data collected for a mockable method call is associated with a specific invocation of an MUT. RICK implements this association by assigning a unique identifier to each MUT invocation, and the same identifier to each mockable method call within it. This information is required for the generation of all of the three kinds of oracles, i.e., the output oracle, the parameter oracle, as well as the call oracle. Second, one invocation of an MUT may have multiple mockable method calls, which may or may not have the same signature. Furthermore, these invocations occur in a specific order within the MUT. In order to account for this, RICK collects the timestamps for each mockable method call. This is done to synthesize statements corresponding to the call oracle, which verify the sequence and frequency of mockable method calls in the generated tests.

3.3.3 Generation of Tests with Mocks

Once RICK has collected data about invocations of MUTs and corresponding mockable method calls, the final phase can begin, triggered by the developer. RICK connects an MUT invocation with mockable method calls by utilizing the unique identifiers assigned to each invocation observed in production. It then generates code to produce the three categories of tests for each invocation, as detailed in subsection 3.2. The final output from the test generation phase is a set of test classes, which include tests from the three categories, for each invocation of an MUT that was observed by RICK in production.

RICK generates tests by bringing together all the data it has observed, collected, and linked to the respective invocation of an MUT and its mockable method calls. Within each test generated by RICK for an MUT:

- The serialized receiving object and parameter(s) for the MUT are deserialized to recreate their production state. For example, the receiving object of the respective MUT is recreated from its serialized XML state on line 5 of Listing 4, Listing 5, and Listing 6.
- External objects, on which mockable method calls occur, are substituted with mock objects. For example, a mock object substitutes the external field `extField` on line 8 of Listing 4. A mock object is prepared for the parameter `ExternalTypeTwo` on line 8 of Listing 5 and Listing 6.
- Mockable method calls become mock method calls: they are stubbed, with production parameter(s) and returned value. The mock method call within `methodUnderTestOne` is stubbed on line 11 of Listing 4. The mock method calls within `methodUnderTestTwo` are stubbed from lines 11 to 14 in Listing 5 and Listing 6.
- The generated test case calls the method under test, once. This makes the test intention very clear: the behavior of the MUT is the one which will be assessed by the oracle. Note that multiple methods may be called on the mock objects, all stubbed. The number of mock objects and stubbed methods is what creates a large testing space.
- The oracle verifies a unique aspect about the invocation of the MUT and its interactions with the external object(s): the **OO** on line 17 of Listing 4 verifies the output of `methodUnderTestOne`, the **PO** on lines 20 to 23 of Listing 5 verify that the mock method calls occur with the same parameters as they did in production, and the **CO** on lines 21 and 22 of Listing 6 verify that the mock method calls happen in the same sequence and the same frequency as they did in production.

3.4 RICK in the Software Development Lifecycle

There are two main use cases for RICK in the software development pipeline. First, for a project that has few automated tests, or uses only manual testing [25], [26], RICK can be used to bootstrap the creation of a test suite. The setup would be as follows: human QA testers are employed to evaluate and manually test the system as a whole. Meanwhile, RICK would capture the realistic interactions that the testers trigger, and would generate automated unit test cases, which can be frequently run when the developers evolve the application.

Second, for projects that already contain automated tests, RICK can contribute with unit tests that reflect realistic behavior, as observed in production. Field executions can be a rich source of data, and are likely to include usage scenarios not envisioned by developers [13]. Leveraging the monitoring and automated test generation capabilities of RICK would allow these behaviors to be incorporated into the test suite. New tests based on production observations have been shown to complement developer-written tests and improve the effectiveness of the test suite [15], [27].

A key phase in both use cases is the curation of a set of essential methods of interest, which will be the targets for test generation with RICK. Though RICK ships with good default filters for identifying target methods, developers can use their domain expertise to define the most valuable target methods within their project. Ideal candidates for test generation include methods that have recently been added or modified, methods at the public interface of the application, or methods that do not meet a specified test adequacy criterion.

The test cases generated by RICK fully depend on the production usages that were monitored. To that extent, RICK is not good at generating tests for corner cases that rarely occur. On the other hand, RICK is excellent at generating tests for mission-critical functionalities that reflect typical usage scenarios for a target application.

Finally, the tests generated by RICK can be fully integrated in a code review process. We envision that the lead test engineer handle the test generation and open a pull-request to add the new tests. Then, fellow developers would review the tests generated for each target method before merging them into the test suite for the project. This process can be repeated multiple times, one target method at a time, throughout the lifecycle of the project.

3.5 Implementation

RICK is implemented in Java. MUTs and their corresponding mockable method calls are identified through static analysis with Spoon [28]. Once identified, they are instrumented and monitored in production with Glowroot, an open-source Application Production Monitoring agent[3]. Glowroot is a well-documented, industry-grade tool. It has a low overhead and is stable. This makes it the best fit for monitoring production executions for mock generation. Data collection from production is handled through serialization by XStream[4]. RICK relies on the code generation capabilities of Spoon[5] to produce JUnit tests[6]. These tests define and use Mockito[7] mocks, specifically the `mockito-inline` flavour, which allows mocking final classes. By default, RICK generates three separate tests that contain the parameter oracle the call oracle, and the output oracle. If the MUT does not return a primitive or a `String`, RICK generates only two tests with the parameter and the call oracles. RICK uses some capabilities provided by the PANKTI framework [15].

4 Experimental Methodology

This section introduces our experimental methodology. We describe the open-source projects we use as case studies to evaluate mock generation with RICK. Then, we describe the production conditions that we use to collect data for test generation. Next, we present our research questions and define the protocol that we use to answer them.

4.1 Case Studies

As detailed in section 3, RICK uses data collected from an application in production, in order to generate tests with mock-based oracles. For this evaluation, we therefore target applications that we can build, deploy, and for which we

3. https://glowroot.org/
4. https://x-stream.github.io/
5. https://spoon.gforge.inria.fr/
6. https://junit.org/
7. https://site.mockito.org/

TABLE 1: Case studies for the evaluation of RICK

METRIC	GRAPHHOPPER	GEPHI	PDFBOX
VERSION	5.3	0.9.6	2.0.24
SHA	af5ac0b	ea3b28f	8876e8e
STARS	3.7K	4.9K	1.7K
COMMITS	5.8K	6.5K	8.7K
LOC	89K	35K	165K
METHODS	4,104	2,117	9,102
CANDIDATE_MUTS	356	115	319

can define a production-grade usage scenario. We manually search for three notable, open-source Java projects that satisfy these criteria. We also make sure that the case studies represent different categories of software: a library with a command-line interface, a desktop application, and a backend server application.

Table 1 summarizes the details of the projects we use as case studies to evaluate the capabilities of RICK. For each case study, we provide the exact version as well as the SHA of the commit used for our experiments. This information will facilitate further replication. We also provide the number of lines of code, the number of commits, and the number of methods as indicators of the scale of the project, as well as the number of stars in the project repository, as an indicator of its visibility. The last row in Table 1 indicates the number of candidate MUTs for each case study, which is the set of MUTs with mockable method calls identified by RICK.

Our first case study is the web-based routing application based on OpenStreetMap called GRAPHHOPPER[8]. It allows users to find the route between locations on a map, considering diverse means of transport and other routing information such as elevation. We use version 5.3 of GRAPHHOPPER, with 89K lines of code (LOC), 5,844 commits, and over 4K methods. The project's repository on GitHub has 3.7K stars.

The second case study is GEPHI, an application for working with graph data[9]. With 4.9K stargazers on GitHub, GEPHI is very popular, and has been adopted by both the industry and by researchers [29]. It allows users to import graph files, manipulate them, and export them in different file formats. We use version 0.9.6 of GEPHI, which includes 6,548 commits and 126K LOC. For our evaluation, we exclude the GUI modules, as the generation of GUI tests has its own challenges [30] that are outside the scope of RICK. The 8 modules of GEPHI we consider are implemented in 35K LOC and contain 2,117 methods in total.

The last case study is PDFBOX, a PDF manipulation command-line tool developed and maintained by the Apache Software Foundation[10] [31]. It can extract text and images from PDF documents, convert between text files and PDF documents, encrypt and decrypt, and split and merge

8. https://www.graphhopper.com/open-source/
9. https://gephi.org/
10. https://pdfbox.apache.org/

TABLE 2: Characteristics of the workloads for the case studies in production: The four metrics are defined in subsection 4.2.

METRIC	GRAPHHOPPER	GEPHI	PDFBOX	TOTAL
MUT_INVOKED	72	68	72	212
MOCKABLE_INVOKED	81	63	55	199
MUT_INVOCATIONS	73,025	21,548	7,429,800	7,524,525
MOCKABLE_INVOCATIONS	246,822	202,630	5,144,790	5,594,242

PDF documents. As highlighted in Table 1, we use version 2.0.24 of PDFBOX, which has 165K LOC, over 9K methods, 8,797 commits, and has been starred by 1.7K GitHub users.

To generate tests with RICK, the first step consists of identifying candidate MUTs, which RICK will instrument so their invocations can be monitored as the project executes in production. According to the criteria introduced in subsection 3.3, RICK identifies and instruments a total of 790 CANDIDATE_MUTs across the three applications: 356 in GRAPHHOPPER, 115 in GEPHI, and 319 in PDFBOX. These methods have interactions with objects of external types, where these objects are either the parameters of the MUT, or a field defined within the declaring type of the MUT.

4.2 Production Usage

Once the candidate MUTs for an application are identified, the instrumented application is deployed and run under a certain workload. As RICK aims at consolidating test suites for mission-critical functionalities that everybody relies on, we design workloads that exercise common features. We manually analyze the case studies' codebase and refer to their documentation, in order to design workloads that capture commonly used features and operations.

Table 2 summarizes the key characteristics of the workloads. Rows 2 and 3 capture the scope for test generation that we consider for our experiments. The number of candidate MUTs actually invoked in production is indicated by MUT_INVOKED, while MOCKABLE_INVOKED represents the number of distinct mockable methods invoked within these invoked MUTs. We also report the number of times the MUTs and their mockable methods are invoked in the rows MUT_INVOCATIONS and MOCKABLE_INVOCATIONS, respectively. The number of invocations of MUTs and mockable methods demonstrate the relevance and comprehensiveness of the production scenarios we design. They represent the extent to which we exercise the three applications in production, and reflect actual usage of their functionalities. In total, RICK focuses on 212 target methods, that invoke 199 mockable method calls. Our experiments trigger more than 7.5 million invocations of these 212 methods.

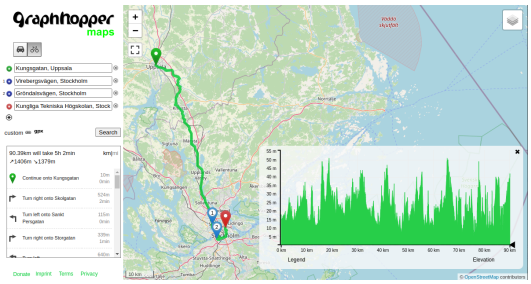


Fig. 3: Snapshot of GRAPHHOPPER in production. We query for the route between 4 locations in Sweden, as RICK monitors target method invocations.

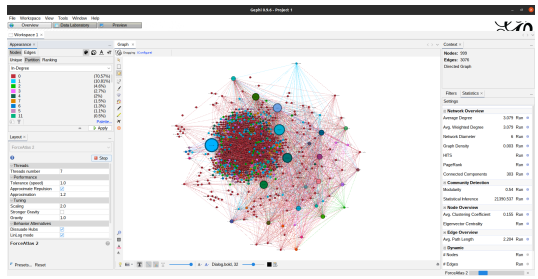


Fig. 4: We use GEPHI to import a data file with details on 999 Java artifacts on Maven Central. We interact with the features of GEPHI to manipulate the resulting graph. RICK monitors method invocations corresponding to these interactions in production.

GRAPHHOPPER: To experiment with GRAPHHOPPER, we deploy its server and use its website to search for the car and bike route between four points in Sweden, specifically, from the residence of each author to their common workplace in Stockholm[11]. Figure 3 is a snapshot of this experiment. Recall from subsection 4.1 that RICK monitors the invocation of 356 CANDIDATE_MUTs as GRAPHHOPPER executes. As presented in Table 2, 72 of the candidate MUTs are invoked (MUT_INVOKED), which become test generation targets for RICK. Within these MUTs, 81 distinct mockable methods are also called. With this production scenario, the 72 MUTs are invoked a total of $73,025$ times, while the 81 mockable methods are invoked $246,822$ times within the MUTs.

GEPHI: As production usage for GEPHI, we deploy the application and import a graph data file. This file has details about the top 999 Java artifacts published on Maven Central, as well as the dependencies between them. We retrieve this data file from previous work [32], [33]. We use GEPHI to produce a graph from this data, and to manipulate its layout, as illustrated in Figure 4. Finally, we export the resulting graph in PDF, PNG, and SVG formats, before exiting the application. These interactions with GEPHI lead to the invocation of 68 of the 115 candidate MUTs, and 63 distinct mockable methods called by these MUTs. Moreover, these MUTs are invoked $21,548$ times, while there are $202,630$ mockable method calls.

PDFBOX: We use the command-line utilities provided by PDFBOX to perform 10 typical PDF manipulation operations on 5 PDF documents. These documents are sourced from [34], and the operations performed on them include text and image extraction, conversion into a text file, and vice-versa, etc. This methodology has been adopted from previous work [15]. Of the 319 candidate MUTs, this workload leads to the invocation of 72 MUTs and 55 different mockable methods. The MUTs are called over 7 million times, and the mockable methods are called over 5 million times. The magnitude of these invocation counts is due to the processing of a myriad of media content from the real-world PDF documents we select.

11. https://bit.ly/3LG2zSQ

4.3 Research Questions

For each application described in subsection 4.1 and exercised in production per the workload specified in subsection 4.2, RICK generates tests with mocks using the captured production data. Through these experiments, we aim to answer the following research questions.

- ***RQ1 [methods under test]***: To what extent can RICK generate tests for MUTs invoked in production?
- ***RQ2 [production-based mocks]***: How rich is the production context reflected in the tests, mocks, and oracles generated by RICK?
- ***RQ3 [mimicking production]***: To what extent can the execution of generated tests and mocks mimic realistic production behavior?
- ***RQ4 [effectiveness]***: How effective are the generated tests at detecting regressions?
- ***RQ5 [quality]***: What is the opinion of developers about the tests generated by RICK?

Each of the research questions presented above highlights a unique aspect of the capabilities of RICK, with respect to the automated generation of tests with mocks, using data sourced from production executions.

4.4 Protocols for Evaluation

The MUTs instrumented by RICK are invoked thousands of times (Table 2). For experimental purposes, and to allow for a thorough, qualitative analysis of the results, we collect data about the first invocation of the MUT in production and use it to generate tests with RICK. We analyze these generated tests according to the following protocols in order to answer the research questions presented in subsection 4.3.

Protocol for RQ1: This first research question aims to characterize the target MUTs for which RICK transforms observations made in production into concrete test cases. We describe the MUTs by reporting their number of lines of code (LOC) as well as the number of parameters. We also report the number of tests generated by RICK for these target MUTs. This includes the tests with the three kind of mock-based oracles, **OO** for MUTs that return primitive values, **PO**, and **CO**, for one invocation of the MUT observed in production.

Protocol for RQ2: With RQ2, we analyse the ability of RICK to capture rich production contexts and turn them into test inputs and oracles that verify distinct aspects of their behavior. We answer RQ2 by dissecting the data captured by RICK as the three applications execute, as well as the tests generated by RICK using this data. First, to characterize the receiving object and parameters captured for the MUTs from production, we discuss the size of the serialized production state on disk. Second, we analyse the three kinds of oracles in the generated tests, specifically the assertion statement in the OO tests, and the number of verification statements in PO and CO tests. Furthermore, we report the number of external objects (fields and/or parameters) mocked within the tests, the stubs produced by RICK based on production observations, as well as the mock method calls.

Protocol for RQ3: With RQ3 we highlight the feasibility and complexity of automatically generating tests and mocks that successfully execute in order to mimic actual production behavior, in an isolated manner. In order to answer RQ3, we execute the generated tests, and we analyse the outcome. There are three possible outcomes of the execution of a generated test. First, a test is successfully executed if the oracles pass, implying that the test mimics the behavior of both the MUT and the mock method calls(s) observed by RICK in production. Second, a generated oracle may fail, meaning that the test and its mocks do not replicate the production observations. For example, objects recreated within the generated test through deserialization may not be identical to those observed in production [15]. Third, during the execution of the test, a runtime exception may happen before the oracles are evaluated, rendering them useless. We report on those cases as well.

Protocol for RQ4: The goal of RQ4 is to determine how effectively the tests with mock-based oracles generated by RICK can detect regressions within the MUTs. In order to do so, we inject realistic bugs within each MUT that has at least one passing RICK test [35]. We rely on the LittleDarwin mutation testing tool [36] to generate a set of first-order mutants for these MUTs. LittleDarwin is ideal for our analysis as it generates mutated source files on disk, which allows for automated, configurable, and reproducible experiments. The 14 mutation operators provided by LittleDarwin include the standard arithmetic, relational, and nullifying mutants, as well as one extreme mutation operator that replaces the whole body of an MUT with a default return statement [37]. Next, we substitute each MUT with a version that contains a mutant reachable by the test input [38], and run each test generated for the MUT by RICK. A test failure indicates that a mutant was covered, detected, and killed by the corresponding **OO**, **PO**, or **CO** within the test. We also analyze whether mock-based oracles differ in their ability to find faults [39], [40].

Protocol for RQ5: RQ5 is a qualitative assessment of the tests generated by RICK. It serves as a proxy for the readiness of the RICK tests to be integrated into the test suite of projects. To assess the quality of the generated tests, we conduct a developer survey, presenting a set of 6 tests generated by RICK for 2 MUTs in GRAPHHOPPER, to 5 software testers from the industry. We carefully select the tests to present to the survey participants in order to have a representation of the three kinds of oracles as well as diverse mocking contexts i.e., external field or parameter objects. Developer surveys have previously been conducted to assess mocking practices [3], [12]. The key novelty of our survey consists in assessing mocks that have been automatically generated.

We conduct each survey online for one hour, and follow this systematic structure: introduce mocking, the RICK test generation pipeline, and the GRAPHHOPPER case study; next, we give the participant access to a fork of GRAPHHOPPER on GitHub, with the RICK tests added, inviting the participant to clone this repository, or browse through it online; finally, we ask them questions about the generated tests. We select GRAPHHOPPER for the survey because its workload, fetching a route on a map, is intuitive and does not add to the complexity of the interview. We select the MUTs for our discussions based on the following criteria: the MUTs have at least 10 lines of code, and the tests generated for them have at least one stub. From these, we select two MUTs in GRAPHHOPPER for which RICK has generated all three mock-based oracles, and for which the tests contain a mocked field and a mocked parameter.

The goal of this survey is to gauge the opinion of developers about the quality of the 6 tests with respect to three criteria: mocking effectiveness, structure, and understandability. Our replication package includes all the details about this survey.

5 EXPERIMENTAL RESULTS

This section presents the results from our evaluation of RICK with GRAPHHOPPER, GEPHI, and PDFBOX. In subsection 5.1, subsection 5.2 and subsection 5.3, we answer RQ1, RQ2 and RQ3 based on the metrics summarized in Table 3, Table 4, and Table 5. The results for RQ4 are presented in subsection 5.4. In subsection 5.5 we answer RQ5 based on the surveys conducted with testers from the industry.

5.1 Results for RQ1 [Methods Under Test]

As presented in the first four columns of Table 3, Table 4, and Table 5, RICK generates tests for 23 MUTs in GRAPHHOPPER, 57 in GEPHI, and 48 in PDFBOX. In total, RICK generates tests for 128 MUTs which have at least one mockable method call. The median number of LOCs in these 128 target methods is 18, while the largest method is MUT#26 in PDFBOX with 328 lines of code. The median number of parameters for the MUTs is 1, while several MUTs (such as MUT#38 and MUT#39 in GEPHI) take as many as 6 parameters. In general, RICK handles a wide variety of MUTs in the case studies, with successful identification and instrumentation of these methods, detailed monitoring in production, as well as the generation of tests that compile and run.

These results validate that mock generation from production can indeed be fully automated, and is robust with respect to the complexity of real world methods. RICK handles the diversity of methods, data types, and interactions observed in real software and production usage scenarios.

In Table 2, we see that the workloads trigger the execution of 72 MUTs in GRAPHHOPPER, 68 in GEPHI, and 72 in PDFBOX. A subset of them are actually used as targets for

TABLE 3: Experimental results for GRAPHHOPPER

RQ1: METHOD UNDER TEST				RQ2: PRODUCTION-BASED MOCKS							RQ3: MIMICKING PRODUCTION		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_OBJ_SIZE	#MOCK_OBJECTS	#MOCK_METHODS	#STUBS	#OO_STMNTS	#PO_STMNTS	#CO_STMNTS	#SUCCESSFULLY_MIMIC	#INCOMPLETELY_MIMIC	#UNHANDLED_MUT_BEHAVIOR
MUT # 1	7	1	2	41 B	1	1	0	0	1	1	0	0	2
MUT # 2	19	1	3	47 B	1	3	6	1	6	9	0	3	0
MUT # 3	8	1	2	53 B	1	2	0	0	2	3	2	0	0
MUT # 4	5	1	2	284 B	1	1	1	0	1	1	2	0	0
MUT # 5	15	1	3	477 B	1	1	0	1	1	1	3	0	0
MUT # 6	13	0	3	982 B	1	3	2	1	3	3	3	0	0
MUT # 7	7	1	2	1.1 KB	1	1	1	1	1	1	2	0	0
MUT # 8	15	2	3	1.5 KB	1	2	4	1	4	2	3	0	0
MUT # 9	25	0	3	2.7 KB	2	5	9	1	10	8	0	3	0
MUT # 10	15	0	3	2.7 KB	2	3	11	1	7	6	0	3	0
MUT # 11	26	0	3	3 KB	2	2	11	1	11	2	0	3	0
MUT # 12	5	1	3	3.1 KB	1	1	1	1	1	1	3	0	0
MUT # 13	5	1	3	3.1 KB	1	1	1	1	1	1	3	0	0
MUT # 14	25	0	3	56.7 KB	2	4	5	1	5	5	0	3	0
MUT # 15	5	1	3	156.8 KB	1	1	1	1	1	1	3	0	0
MUT # 16	58	1	3	156.8 KB	1	1	1	1	1	1	3	0	0
MUT # 17	18	1	3	156.9 KB	1	1	1	1	1	1	3	0	0
MUT # 18	5	1	3	383 KB	1	1	1	1	1	1	3	0	0
MUT # 19	74	1	2	590 KB	2	2	0	0	2	2	2	0	0
MUT # 20	8	0	2	729.2 KB	1	2	20	0	20	20	0	2	0
MUT # 21	57	0	3	2.2 MB	3	3	2	1	3	3	0	0	3
MUT # 22	37	2	3	9.8 MB	1	3	3	1	4	4	2	1	0
MUT # 23	4	0	2	9.8 MB	2	1	0	0	1	1	0	2	0
TOTAL: 23	**MEDIAN: 15**	**MEDIAN: 1**	**62**	**MEDIAN: 3.1 KB**	**31**	**45**	**81**	**16**	**88**	**78**	**37**	**20**	**5**

test generation: 23, 57, and 48 MUTs in the respective case studies. This happens due to two reasons.

First, when statically identifying targets for test generation, RICK finds MUTs with mockable method calls. However, as highlighted in subsubsection 3.3.2, MUTs may be invoked, without their corresponding mockable methods being called, because of the flow of control through the program. For example, a mockable method call may happen only within a certain path through the MUT, which is not observed in production. In this case, a test with mocks is not generated. Second, the receiving objects for some MUTs are sometimes too large and complex to be captured through serialization, which is the case for some MUTs invoked within GRAPHHOPPER. We have observed serialized object snapshots beyond tens of megabytes, which reaches the scalability limits of state of the art serialization techniques.

The difference between the number of invoked MUTs and the number of MUTs for which RICK generates tests is most significant for GRAPHHOPPER. We notice that many receiving objects for GRAPHHOPPER do not get successfully serialized owing to their large size. For example, the receiving object for an invoked MUT was as large as 454 MB, before even being serialized as XML. To counter this, we allocated more heap space, increasing it up to 9 GB, while deploying the server, yet were unsuccessful in serializing it. Our research on using serialization for automated mocking clearly touches the frontier of serialization for handling arbitrarily large and complex data from production.

The total number of tests generated by RICK for the 128 target MUTs is 294. Recall from subsection 4.2 that we only monitor and generate tests for a single invocation – the first one – of each of the target MUTs. Note also that, as signified by the column #TESTS in Table 3, Table 4, and Table 5, RICK generates either 2 or 3 tests for each target MUT. The number of tests generated for an MUT depends on its return type. For MUTs that return a non-void, primitive value, such as MUT#2 in GRAPHHOPPER, RICK generates an **OO** test to assert on the output of the MUT invocation, in addition to a **PO** and **CO** test. Across the three case studies, RICK generates **OO** tests for 38 MUTs, and **PO** and **CO** tests for all 128 MUTs.

Answer to RQ1

RICK captures the production behavior for a set of 128 out of the 212 MUTs invoked in production. RICK transforms the data collected from these production invocations into 294 concrete tests with different oracles. The key result of RQ1 is that RICK handles a large variety of real world methods in an end to end manner, from monitoring in production to the generation of tests with mocks.

5.2 Results for RQ2 [Production-based Mocks]

RICK generates mocks from real data observed in production. While RQ1 has demonstrated feasibility, RQ2 explores how production data is reflected in the generated test cases.

Columns 5 through 11 in Table 3, Table 4, and Table 5 present the results for RQ2. Each row highlights the data that RICK collects for one MUT and its mock method call(s) from production. The column CAPTURED_OBJ_SIZE presents the size on disk (in B / KB / MB) of the serialized object states for the receiving object and, if present, the parameter objects of the MUT. We also present the number of external objects mocked within the test (#MOCK_OBJECTS), the number of methods called on these mock objects (#MOCK_METHODS), the number of stubs (#STUBS) defined for these mock methods, and the number of statements corresponding to the **OO**, **PO**, and **CO** oracles in the gener-

TABLE 4: Experimental results for GEPHI

RQ1: METHOD UNDER TEST				RQ2: PRODUCTION-BASED MOCKS							RQ3: MIMICKING PRODUCTION		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_OBJ_SIZE	#MOCK_OBJECTS	#MOCK_METHODS	#STUBS	#OO_STMNTS	#PO_STMNTS	#CO_STMNTS	#SUCCESSFULLY_MIMIC	#INCOMPLETELY_MIMIC	#UNHANDLED_MUT_BEHAVIOR
MUT # 1	16	1	2	48 B	1	2	2	0	2	2	0	0	2
MUT # 2	14	1	2	48 B	1	1	1	0	1	1	0	0	2
MUT # 3	9	3	2	56 B	1	1	1	0	1	1	2	0	0
MUT # 4	6	1	3	56 B	1	1	1	1	1	1	0	0	3
MUT # 5	16	1	3	56 B	1	1	1	1	1	1	3	0	0
MUT # 6	31	3	3	56 B	1	1	0	1	1	1	3	0	0
MUT # 7	12	2	2	118 B	1	4	2	0	4	4	2	0	0
MUT # 8	12	2	2	134 B	1	2	0	0	2	2	1	1	0
MUT # 9	11	2	3	143 B	1	2	2	1	2	2	3	0	0
MUT # 10	8	1	2	186 B	1	1	0	0	1	1	0	0	2
MUT # 11	18	3	2	206 B	1	4	4	0	4	20	1	1	0
MUT # 12	13	1	2	237 B	1	2	1	0	2	2	0	0	2
MUT # 13	14	1	2	238 B	1	1	1	0	1	1	0	0	2
MUT # 14	12	1	2	255 B	1	1	1	0	1	1	0	0	2
MUT # 15	17	1	2	291 B	1	1	1	0	1	1	0	0	2
MUT # 16	24	1	2	299 B	1	1	1	0	1	1	0	0	2
MUT # 17	43	3	3	307 B	1	1	1	1	1	1	0	0	3
MUT # 18	18	1	2	340 B	1	1	1	0	1	1	0	0	2
MUT # 19	7	1	2	344 B	1	1	1	0	1	1	2	0	0
MUT # 20	29	1	2	425 B	1	1	1	0	1	1	0	0	2
MUT # 21	11	1	3	472 B	1	2	2	1	2	2	3	0	0
MUT # 22	40	3	2	1.7 KB	3	3	1	0	3	3	0	2	0
MUT # 23	50	2	2	2.2 KB	1	2	1	0	2	2	0	2	0
MUT # 24	23	3	2	4.4 KB	1	1	1	0	1	1	2	0	0
MUT # 25	13	1	2	58.2 KB	3	3	3	0	3	3	0	2	0
MUT # 26	4	1	3	58.2 KB	1	1	1	1	1	1	3	0	0
MUT # 27	23	3	2	59.2 KB	1	1	1	0	1	1	2	0	0
MUT # 28	53	1	3	60.1 KB	1	4	10	1	4	27	0	3	0
MUT # 29	21	2	2	79 KB	1	1	1	0	1	1	2	0	0
MUT # 30	85	4	2	100 KB	1	1	1	0	1	1	0	0	2
MUT # 31	85	1	2	112.5 KB	4	4	5	0	4	6	0	0	2
MUT # 32	25	1	2	112.5 KB	1	1	1	0	1	1	2	0	0
MUT # 33	10	1	3	149.5 KB	1	1	1	1	1	1	2	1	0
MUT # 34	90	0	3	177.4 KB	1	1	1	1	1	1	0	3	0
MUT # 35	15	3	2	196.3 KB	1	3	0	0	3	3	2	0	0
MUT # 36	28	2	2	276 KB	1	1	1	0	1	1	0	0	2
MUT # 37	40	2	2	300 KB	1	1	1	0	1	1	0	0	2
MUT # 38	18	6	2	370 KB	1	1	1	0	1	1	0	0	2
MUT # 39	16	6	3	600 KB	1	1	1	1	1	1	0	0	3
MUT # 40	5	2	2	3.6 MB	1	1	1	0	1	1	0	0	2
MUT # 41	62	6	2	3.6 MB	2	8	8	0	8	8	0	0	2
MUT # 42	15	6	2	3.8 MB	1	1	1	0	1	1	0	0	2
MUT # 43	26	0	2	3.9 MB	1	1	6	0	1	1	0	2	0
MUT # 44	185	0	2	3.9 MB	1	1	6	0	1	1	0	0	2
MUT # 45	11	0	2	3.9 MB	1	1	0	0	1	1	2	0	0
MUT # 46	123	0	2	3.9 MB	2	3	1	0	3	3	0	2	0
MUT # 47	59	5	2	4 MB	1	1	3	0	1	1	0	0	2
MUT # 48	45	6	2	4.1 MB	2	2	3	0	2	2	0	0	2
MUT # 49	20	0	2	4.2 MB	1	1	0	0	1	1	2	0	0
MUT # 50	18	2	3	4.5 MB	1	1	0	1	1	1	0	0	3
MUT # 51	126	0	2	10.8 MB	1	1	6	0	1	1	0	2	0
MUT # 52	24	0	2	10.8 MB	1	1	1	0	1	1	0	2	0
MUT # 53	24	0	2	10.8 MB	1	1	1	0	1	1	0	2	0
MUT # 54	15	2	2	11.2 MB	1	1	1	0	1	1	2	0	0
MUT # 55	11	2	2	11.6 MB	1	2	4	0	2	2	1	1	0
MUT # 56	7	3	2	11.6 MB	1	1	1	0	1	1	2	0	0
MUT # 57	32	3	2	11.6 MB	1	2	3	0	3	3	0	0	2
TOTAL: 57	**MEDIAN: 18**	**MEDIAN: 1**	**126**	**MEDIAN: 60.1 KB**	**67**	**93**	**103**	**12**	**94**	**135**	**44**	**26**	**56**

TABLE 5: Experimental results for PDFBox

RQ1: Method Under Test				RQ2: Production-based Mocks							RQ3: Mimicking Production		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_OBJ_SIZE	MOCK_OBJECTS	MOCK_METHODS	#STUBS	#OO_STMNTS	#PO_STMNTS	#CO_STMNTS	#SUCCESSFULLY_MIMIC	#INCOMPLETELY_MIMIC	#UNHANDLED_MUT_BEHAVIOR
MUT # 1	84	5	2	37 B	1	1	0	0	1	1	2	0	0
MUT # 2	6	1	2	352 B	1	2	2	0	2	4	0	0	2
MUT # 3	11	1	2	3.3 KB	1	1	0	0	1	1	0	0	2
MUT # 4	5	1	2	4.3 KB	1	1	0	0	1	1	2	0	0
MUT # 5	5	1	2	4.7 KB	1	1	0	0	1	1	2	0	0
MUT # 6	101	3	2	6.5 KB	1	1	1	0	1	1	0	0	2
MUT # 7	28	1	2	11.7 KB	1	1	0	0	5	1	2	0	0
MUT # 8	34	1	2	267 KB	1	1	1	0	3	2	2	0	0
MUT # 9	42	2	3	1.6 MB	1	1	1	1	1	1	3	0	0
MUT # 10	33	1	2	2 MB	1	1	0	0	1	1	2	0	0
MUT # 11	16	1	2	2.2 MB	1	1	1	0	1	1	0	0	2
MUT # 12	10	0	3	2.3 MB	1	2	2	1	2	2	3	0	0
MUT # 13	19	3	3	2.3 MB	1	2	2	1	2	2	3	0	0
MUT # 14	19	0	2	2.4 MB	1	1	1	0	1	1	2	0	0
MUT # 15	18	0	3	3 MB	1	1	1	1	1	1	3	0	0
MUT # 16	18	2	2	3 MB	1	1	1	0	1	1	2	0	0
MUT # 17	8	0	2	3.2 MB	1	1	1	0	1	1	2	0	0
MUT # 18	9	1	3	3.6 MB	1	1	1	1	1	1	3	0	0
MUT # 19	8	0	3	3.8 MB	1	1	1	1	1	1	3	0	0
MUT # 20	18	1	2	3.8 MB	1	2	2	0	2	2	2	0	0
MUT # 21	5	1	2	4.4 MB	1	1	1	0	1	1	2	0	0
MUT # 22	19	0	3	4.6 MB	1	2	2	1	2	2	3	0	0
MUT # 23	39	1	2	4.6 MB	1	1	0	0	1	1	2	0	0
MUT # 24	85	1	3	5.3 MB	3	1	1	1	1	1	0	3	0
MUT # 25	40	1	2	5.3 MB	1	1	1	0	1	1	0	0	2
MUT # 26	328	2	2	5.4 MB	1	1	1	0	1	1	0	0	2
MUT # 27	11	1	3	5.6 MB	1	2	2	1	2	2	3	0	0
MUT # 28	29	2	3	5.8 MB	1	2	3	1	2	4	2	1	0
MUT # 29	79	4	2	6.9 MB	1	1	0	0	1	1	2	0	0
MUT # 30	19	1	2	6.4 MB	1	1	0	0	1	1	2	0	0
MUT # 31	31	1	2	6.6 MB	1	1	0	0	1	1	2	0	0
MUT # 32	62	4	2	6.8 MB	1	1	1	0	1	1	2	0	0
MUT # 33	23	1	2	7.2 MB	1	1	0	0	1	1	2	0	0
MUT # 34	27	2	2	7.4 MB	1	1	0	0	1	1	0	0	2
MUT # 35	40	1	2	7.6 MB	1	1	1	0	1	1	0	0	2
MUT # 36	20	1	2	8.3 MB	1	1	1	0	1	1	0	2	0
MUT # 37	12	1	2	9.5 MB	1	1	0	0	1	1	2	0	0
MUT # 38	12	1	2	9.5 MB	1	1	0	0	1	1	2	0	0
MUT # 39	11	1	2	9.7 MB	1	1	0	0	1	1	2	0	0
MUT # 40	23	1	2	10.1 MB	1	1	0	0	1	0	2	0	0
MUT # 41	11	0	2	11.1 MB	1	1	0	0	1	1	1	1	0
MUT # 42	5	1	2	21.4 MB	1	1	0	0	1	1	2	0	0
MUT # 43	8	4	2	30.8 MB	3	5	4	0	7	13	0	2	0
MUT # 44	14	0	2	31 MB	1	2	0	0	2	2	0	0	2
MUT # 45	17	2	2	32 MB	1	2	0	0	3	3	0	2	0
MUT # 46	36	2	2	32 MB	1	1	0	0	1	1	0	0	2
MUT # 47	12	0	2	34 MB	2	5	0	0	5	5	2	0	0
MUT # 48	117	1	2	39 MB	1	2	2	0	2	3	0	0	2
TOTAL: 48	**MEDIAN: 19**	**MEDIAN: 1**	**106**	**MEDIAN: 5.3 MB**	**53**	**66**	**38**	**10**	**75**	**80**	**73**	**11**	**22**

ated tests (signified through #OO_STMNTS, #PO_STMNTS, and #CO_STMNTS, respectively).

We first discuss the serialized production state, defined as a receiving object on which the MUT is invoked, as well as the parameters with which this invocation is made. Recall that the receiving object is serialized. For example, consider MUT#32 in PDFBox (Table 5). RICK captures the production state of the receiving object on which MUT#32 gets invoked, and the 4 parameters for this invocation. The size of these captured objects amounts to a total of 6.8 MB. These objects serve as a snapshot for reproducing the production state within the 2 tests generated by RICK for MUT#32. For our experiments with the three applications, the maximum size of the captured objects is 39 MB for MUT#48 in PDFBox. The median size of these captured objects is 486 bytes of serialized data. Table 5 characterizes the realistic nature of the objects that RICK captures over 128 real-world MUTs, as they are invoked during the end-to-end execution of the applications.

The main feature of RICK is to monitor and collect data about mockable method calls that occur within the MUT, with the receiving objects and passed parameters. Specifically, as detailed in subsection 3.3, RICK mocks the parameters and fields of external types, on which mockable method calls occur in production. For example, there are 2 mock objects within the 2 tests generated for MUT#19 in GraphHopper (Table 3) for two fields in the declaring type of MUT#19. Moreover, 2 mock method calls are made within

```
1 @Test
2 @DisplayName("moveNode with parameter oracle,
3 mocking Node.x(), Node.y(), Node.setX(float), Node.setY(float)")
4 public void testMoveNode_PO() throws Exception {
5   // Arrange
6   StepDisplacement receivingObject =
          deserialize("receiving.xml");
7   Object[] paramObjects = deserialize("params.xml");
8   ForceVector paramObject2 = (ForceVector) paramObjects[1];
9   Node mockNode = Mockito.mock(Node.class);
10  when(mockNode.x()).thenReturn(-423.78378F);
11  when(mockNode.y()).thenReturn(107.523186F);

13  // Act
14  receivingObject.moveNode(mockNode, paramObject2);

16  // Assert
17  verify(mockNode, atLeastOnce()).x();
18  verify(mockNode, atLeastOnce()).y();
19  verify(mockNode, atLeastOnce()).setX(-403.92587F);
20  verify(mockNode, atLeastOnce()).setY(105.14341F);
21 }
```

Listing 7: The **PO** test generated by RICK for MUT#7 in GEPHI, which is a method called `moveNode`. The test mocks a parameter on which four mocked methods are invoked. The behavior of two mocked methods is stubbed within the test.

the tests, one on each of the 2 mock objects. In comparison, there is 1 mock object within each of the 3 tests generated for MUT#21 in GEPHI (Table 4). This mock object replaces the parameter of MUT#21, on which 2 mockable method calls are observed by RICK in production. In total, RICK uses 151 mock objects as parameter or field across the generated tests. 204 mock methods are invoked on these mock objects, reflecting the production interactions with external objects within the MUTs. These generated mock objects recreate actual interactions of the MUT with its environment. The data serialized from production provide developers with realistic test data.

Within the generated tests, the behavior of the mock objects is defined through method stubs. A stub provides the canned response that should be returned from a non-void method called on a mock object, given a set of parameters, i.e., it defines the behavior of a mock method within the test for an MUT. RICK sources the parameters and the primitive returned value from production observations, and represents them through stubs within the generated tests. For example, RICK generates 2 tests for MUT#7 in GEPHI (Table 4). This MUT is the method `moveNode(Node,ForceVector)`. Within each of the generated tests, 1 of the parameters of `moveNode` is mocked, and 4 methods are invoked on this mock object. We present the **PO** test generated for `moveNode` in Listing 7. On line 9 the `Node` object is mocked. Within the invocation of `moveNode` in production, RICK recorded the invocation of the methods `x()` and `y()` on the `Node` object, including their returned values. Consequently, RICK expresses their behavior through the 2 stubs in the generated test (lines 10 and 11) using this production data. The invocations of the two other mockable methods `setX(float)` and `setY(float)` are not stubbed because they are void methods. In total, RICK generates a total of 222 stubs to mimic the production behavior of mockable method calls that occur within MUTs. These generated stubs guide the behavior of the MUT within the generated test, per the observations made for it in production.

We observe from the column #OO_STMNTS in Table 3, Table 4, and Table 5, that the number of **OO** statements for any MUT is either 0 or 1. This is because the oracle in **OO** tests is expressed as a single assertion statement for the output of an MUT that returns a primitive value. However, the number of **PO** and **CO** statements within each test varies depending on the observations made for the corresponding MUT in production. For instance, three tests are generated for MUT#8 in GRAPHHOPPER (Table 3). In each of the three tests, there is 1 mock object, and 2 different mock methods are invoked on this mock object. The behavior of the 2 mock methods is defined through the 4 generated stubs. The **OO** test has an assertion to verify the output of MUT#8. The **PO** test has 4 verification statements to verify the parameters with which the mock methods are called. The **CO** test has 2 verification statements which correspond to the observations made by RICK about the sequence and frequency of these mock method calls within the invocation of MUT#8 in production.

In total, across the 294 tests generated for the 128 MUTs, RICK generates 38 assertion statements (one in each **OO**), 257 statements that verify the parameters with which mock methods are called, and 293 statements to verify the sequence and frequency of mock method calls. Furthermore, RICK uses the parameters passed to, and the value returned from the mockable method calls observed in production, to generate a total of 222 stubs across the 294 tests.

Answer to RQ2

RICK captures a wide range of production data for test generation. We have demonstrated with static insights that it can handle well the two dimensions of mock-based testing: capturing the production state in mock objects, and stubbing real-world methods. The analysis of the generated tests shows that RICK can generate various types of oracles that verify different aspects of the MUT interacting with its environment.

5.3 Results for RQ3 [Mimicking Production]

The results for RQ3 are presented in the last three columns in Table 3, Table 4, and Table 5. All the tests generated by RICK for the 128 MUTs are self-contained and compile correctly. We run each generated test ten times and verify that it is not flaky. Next, for each test, we report the status of its execution. In each row, the column #SUCCESSFULLY_MIMIC highlights the number of tests that completely recreate the observed production context with mocks and oracle(s) that pass, while #INCOMPLETELY_MIMIC signifies the number of tests for which at least one oracle fails. The last column, #UNHANDLED_MUT_BEHAVIOR, represents those test executions where we observe a runtime exception thrown by the MUT. We now discuss the implications of each of these scenarios, and why they occur.

An **OO** test successfully mimics the production context if the MUT returns the same output as it did in production, when invoked with mocks replacing the external objects, and stubs for the behavior of the mock method calls. If the test does not completely mimic the production behavior of the MUT, the invocation of the MUT returns a different

output, which is unequal to the one returned by the MUT in production. Consequently, the assertion statement within the **OO** test fails.

For a **PO** test to be successful, all the verification statements must pass, indicating that mock methods are called by the MUT within the generated test with the same arguments as the ones observed in production. In comparison, a failure in any of the verification statements implies that a mock method call occurs with different parameters than the ones observed for its invocation in production. This implies that the test does not faithfully recreate the observed interactions of the MUT and the mock method.

A passing **CO** test verifies that the mock method calls occur in the same order and the same number of times within the MUT, as they did in production. On the contrary, if the test does not completely mimic the order and/or frequency of mock method calls, a verification statement will fail.

The proportion of tests that successfully mimic production behavior differs across case studies (column #SUCCESSFULLY_MIMIC). Of the tests generated for GraphHopper, 59.7% are successful. In Gephi, the successful tests are 35% of the total generated tests, while in PDFBox 68.9% tests are successful. Overall, the 154 successful tests account for 52.4% of all the tests generated. This is arguably a high ratio given the multi-stage pipeline of RICK, where each stage can fail in some conditions. In 52.4% of cases, all stages of RICK succeed: All the captured objects are correctly serialized and deserialized before the MUT is invoked, recreating an appropriate and realistic execution state. Also, the mock methods are successfully stubbed: they mimic production behavior without impacting the behavior of the MUT.

In the column #INCOMPLETELY_MIMIC, we observe that 57 generated tests, which account for 19.4% of the total, have a failing oracle, implying that they do not completely mimic production behavior. This includes 32.2% of the GraphHopper tests, 20.6% Gephi tests, and 10.4% of the tests generated for PDFBox. These failures can occur due to the following reasons.

Unfaithful recreation of production state: The captured objects may be inaccurately deserialized within the generated test, implying that production states are not completely recreated. Deserialization of complex objects captured from production is a known problem and a key challenge for recreating real production conditions in generated tests [15], [27]. A test may also fail because a production resource, such as a file, is not available during test execution, resulting in an exception. Moreover, since mocks are skeletal objects that substitute a concrete object within the test, they can induce a change in the path taken through the MUT, which renders the oracle unsuccessful. For example, the tests generated for MUT#2 in GraphHopper (Table 3) fail due to failing **OO**, **PO**, and **CO** oracles. The tests mock an object of type `com.graphhopper.util.PointList`, and MUT#2 tries to invoke a loop over the size of this mock list of points. Since the loop is not exercised, a different path is traversed through the MUT than the one observed in production.

Type-based stubbing: We observe that some tests fail because of the granularity of stubbing. Glowroot, the current infrastructure for monitoring within RICK, identifies a target type based on its fully qualified name. Consequently, RICK stubs mockable methods called on the type of an object, but not based on specific instances of the object. A failure can occur if an MUT calls the same mockable method on multiple parameters or fields of the same type. For instance, we present an excerpt of MUT#24 in PDFBox in Listing 8. This method `codeToGID(int)` (line 6), has three calls to the same mockable method, `getGlyphId(int)` (lines 9, 11, and 13). These calls are made on three different fields of type `CmapSubtable` called `cmapWinUnicode`, `cmapWinSymbol`, and `cmapMacRoman` defined in the `PDTrueTypeFont` class (lines 2 to 4). RICK records the mockable method call in production, and mocks the three fields. However, the information on which of these mock fields actually calls the mock method is not available. One solution would be to do object-based stubbing, but we are not aware of any work on this and consider this sophistication as future work.

```
1 class PDTrueTypeFont {
2   CmapSubtable cmapWinUnicode;
3   CmapSubtable cmapWinSymbol;
4   CmapSubtable cmapMacRoman;

6   public int codeToGID(int code) {
7     ...
8     if (...) {
9       gid = cmapWinUnicode.getGlyphId(...);
10    } else if (...) {
11      gid = cmapMacRoman.getGlyphId(...);
12    } else {
13      gid = cmapWinSymbol.getGlyphId(...);
14    }
15    ...
16    return gid;
17  }
18 }
```

Listing 8: The same mockable method, `getGlyphId`, is called on three different fields of the same type within MUT#24 of PDFBox, `codeToGID`

We now discuss the cases of #UNHANDLED_MUT_BEHAVIOR. The execution of 83 of the 294 generated tests (28.2%) causes exceptions to be thrown by the MUT. For example, the MUT may have multiple non-mockable interactions with the mock object, before the mock method is invoked on it. Any of these other interactions can behave unexpectedly, resulting in exceptions to be raised before the oracle is even evaluated within the test. Examining the test execution logs, we see such unhandled behaviors as exceptions. We observe these cases in all three applications: 8.1% in GraphHopper, 44.5% in Gephi, 20.7% in PDFBox. For example, a null pointer exception is thrown from MUT#1 in GraphHopper, when it calls other methods on the mock object before the mockable method is called. Across the 83 unhandled cases, 74 arise from null pointer exceptions, of which 54 are in Gephi, 18 in PDFBox, and 2 in GraphHopper. We find the other two unhandled cases for Gephi in the tests generated for MUT#31, `addEdge(EdgeDraft)`, where the parameter `EdgeDraft` is one of the 4 mock objects. This MUT calls another method, which has been designed by the developers of Gephi to throw a `ClassCastException` if `EdgeDraft` is not an instance of `ElementDraftImpl`, which fully explains the failure to execute with a mock. In GraphHopper, the three tests generated for `load` (MUT#21) mock the field `EncodingManager` within the type `GraphHopper`. This MUT

invokes method `checkProfilesConsistency`, which is designed to return an `IllegalArgumentException` if the `EncodingManager` does not have an encoder for the vehicle set in the profile. In PDFBox, the four tests generated for `prepareForDecryption` (MUT#6) and `getColorSpace` (MUT#34) throw an `IOException` because methods called by these MUTs find an unexpected value when accessing a field within the mock object. Across all these cases, the unhandled behavior occurs within a non-mockable method which is called by the MUT, and is thus indirectly called by the generated test. Our results demonstrate that automatic mocking is full of caveats and handling all corner cases is an important direction for future work on automated mock generation.

Answer to RQ3

RICK succeeds in generating 294 tests, of which 154 (52.4%) fully mimic production observations with fully passing oracles. For 19.4% of the test cases, at least one oracle statement fails, showing that the oracles can indeed differentiate between successfully mimicked and incompletely mimicked contexts. At runtime, the majority of the tests generated by RICK completely mimic production behavior in the sense that the state asserted by the oracle is equal to the one observed in production. The cases where the generated tests fail at runtime reveal promising research directions for sophisticated production monitoring tools, such as effective deserialization and efficient resource snapshotting.

5.4 Results for RQ4 [Effectiveness]

As described in subsection 4.4, we want to determine the effectiveness of RICK tests at determining regressions [35]. We generate a set of first-order mutants for each MUT with LittleDarwin [36]. We focus the generation of mutants for the 68 MUTs across the three projects that have at least one passing RICK test. Furthermore, we consider the mutants that are covered by the test input in the generated tests, i.e., mutants that lie on the path of the MUTs exercised by the tests. This is because a mutant that lies on an uncovered path will be undetectable by design [38]. In total, we consider 449 mutants: 69 mutants for the 14 GRAPHHOPPER MUTs that have at least one passing test, 107 mutants for the 21 MUTs in GEPHI, and 273 mutants for the 33 PDFBOX MUTs. Our replication package[12] contains the automated mutation analysis pipeline, as well as the generated mutants and test execution logs. We also include detailed reports on the set of mutants detected by each mock-based oracle of each MUT.

Our findings from the execution of the generated tests against the mutants are summarized in Figure 5. The Venn diagrams represent the distribution of the 210 mutants killed by **OO**, **PO**, and **CO** for the three case studies. We note from the Venn diagrams that 19 mutants in GRAPHHOPPER, 8 mutants in GEPHI, and 20 in PDFBOX are killed by all three mock-based oracles. Meanwhile, for all three projects, the three mock-based oracles differ in their ability to detect mutants. For example, per Figure 5a, 2 and 4 mutants in GRAPHHOPPER are detected only by **OO** and **PO**, respectively. Likewise, in Figure 5c, 2 mutants in PDFBOX result in extra invocations of a mocked method that are only detected by **CO**. Moreover, in Figure 5b, 2 mutants in GEPHI are killed by **OO** and **PO**, but not **CO**. Across the three projects, **OO** kills 16 mutants, **PO** kills 18 mutants, and **CO** kills 2 mutants that are undetected by other oracles. The lower number of mutants killed only by **CO** can be attributed to the fact that LittleDarwin does not include a mutation operator that directly removes method calls. Also, mocked methods may be invoked in the expected order and frequency, but with different, mutated parameters. This will not be detected by **CO** but will be detected by **PO**. The set of mutants killed by **OO** is always smaller than for the other oracles. This is because, as highlighted in subsection 5.1, we generate **OO** tests only for MUTs that return primitive values. Yet, when present, **OO** kills mutants in all three projects, i.e., 24 mutants in GRAPHHOPPER, 17 in GEPHI, and 30 mutants in PDFBOX.

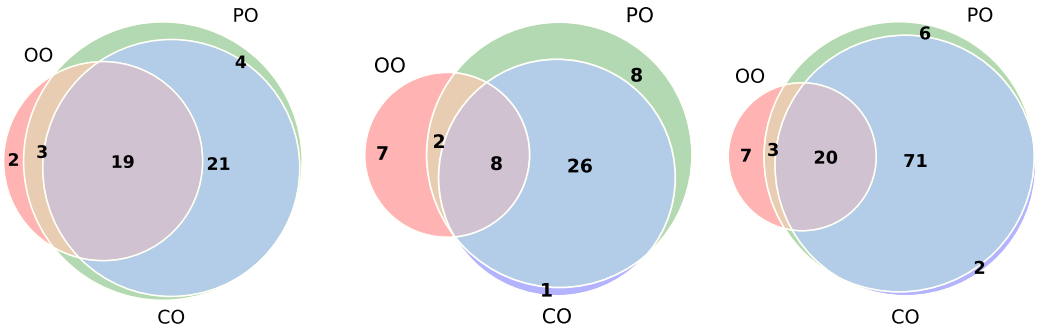
Overall, all three types of oracles are effective at detecting regressions. We have also observed that the generated tests kill at least one mutant for each MUT. These observations are evidence that the RICK tests, with inputs sourced from production, indeed specify the behavior of the MUTs. Moreover, **OO**, **PO**, and **CO** can detect different bugs. The RICK tests with mock-based oracles can therefore complement each other, even given the same test input. This aligns with the findings of Staats *et al.* [39] and Gay *et al.* [40] that multiple oracles specified for a test input may perform differently with respect to their fault-finding ability.

We illustrate this phenomenon using the example of MUT#8 in GRAPHHOPPER, which is the `loadExisting` method presented in Listing 9. Listing 11 presents the common *Arrange* and *Act* phases of the three tests generated by RICK for `loadExisting`. Listing 12, Listing 13, and Listing 14 contain the *Assert* phase of the **OO**, **PO**, and **CO** test, respectively. The 5 mutants produced by LittleDarwin for `loadExisting`, are shown in Listing 10. The generated **OO** test detects 2 of these mutants (#2 and #3), as the assertion (Listing 12) fails on an output that differs from the expected `boolean` value. The **CO** (Listing 14) kills 3 mutants (#1, #2, and #3), as the invocations to the methods `loadExisting` and `getHeader` are expected on the mocked `DataAccess` object, but do not occur due to the mutation. The verification statements in the **PO** (Listing 13) kill all 5 mutants. This is because the expected mock method invocations within the MUT either do not occur entirely, or occur with unexpected

```
1 public class LineIntIndex {
2  ...
3  public boolean loadExisting() {
4    ...
5    if (!dataAccess.loadExisting())
6     return false;
7    ...
8    GHUtility.checkDAVersion(..., dataAccess.getHeader(0));
9    checksum = dataAccess.getHeader(1 * 4);
10   minResolutionInMeter = dataAccess.getHeader(2 * 4);
11   ...
12   return true;
13  }
14 }
```

Listing 9: The original MUT#8 in GRAPHHOPPER, `loadExisting`

12. https://github.com/ASSERT-KTH/rick-experiments



(a) GRAPHHOPPER (b) GEPHI (c) PDFBOX

Fig. 5: The RICK tests kill 49 mutants in GRAPHHOPPER, 52 mutants in GEPHI, and 109 mutants in PDFBOX. The three types of mock-based oracles complement each other to detect regressions in all three projects.

```
// Mutant #1: Extreme mutation - Lines 4 to 12
public boolean loadExisting() {
- if (initialized)
- ...
- return true;
+ return true;
}

// Mutant #2: Extreme mutation - Lines 4 to 12
public boolean loadExisting() {
- if (initialized)
- ...
- return true;
+ return false;
}

// Mutant #3: Line 5
- if (!dataAccess.loadExisting())
+ if (dataAccess.loadExisting())

// Mutant #4: Line 9
- checksum = dataAccess.getHeader(1 * 4);
+ checksum = dataAccess.getHeader(1 / 4);

// Mutant #5: Line 10
- minResolutionInMeter = dataAccess.getHeader(2 * 4);
+ minResolutionInMeter = dataAccess.getHeader(2 / 4);
```

Listing 10: Five first-order mutants are introduced in `loadExisting`. The line numbers correspond to the line numbers in Listing 9.

parameters, causing the **PO** test to fail.

We now discuss cases where mutants are not killed by RICK tests. First, a mutant will be undetected if it results in a behavior that is equivalent to that of the original MUT. This is a well-known limitation of mutation analysis [41]. Second, a mutant will be undetected if the production input cannot infect the program's state, or if a mock-based oracle does not capture its side-effects. The challenges of producing test cases that can effectively infect, propagate and observe the effects of a mutant are well known in the literature [42], [43]. An interesting prospect for future work is to trigger the detection of alive covered mutants through stubbing. For example, we observe this for mutant #1 in `loadExisting` (Listing 10), which is undetected by the **OO** in Listing 12. The mutation causes the MUT to directly return `true`, which

```
1 @Test
2 @DisplayName("Test for loadExisting, mocking
3 DataAccess.loadExisting(), DataAccess.getHeader(int)")
4 public void testLoadExisting() {
5   // Arrange
6   LineIntIndex receivingObject = deserialize(
        "receiving.xml");
7   DataAccess mockDataAccess =
        insertMockField_DataAccess_InLineIntIndex(
        receivingObject);
8   when(mockDataAccess.loadExisting()).thenReturn(true);
9   when(mockDataAccess.getHeader(0)).thenReturn(5);
10  when(mockDataAccess.getHeader(4)).thenReturn(1813699);
11  when(mockDataAccess.getHeader(8)).thenReturn(300);

13  // Act
14  boolean actual = receivingObject.loadExisting();

16  // Assert
17  ...
```

Listing 11: The *Arrange* and *Act* phases of the RICK tests for `loadExisting`

```
17  assertEquals(true, actual);
18 }
```

Listing 12: The generated **OO** for `loadExisting`

```
17  verify(mockDataAccess, atLeastOnce()).loadExisting();
18  verify(mockDataAccess, atLeastOnce()).getHeader(0);
19  verify(mockDataAccess, atLeastOnce()).getHeader(4);
20  verify(mockDataAccess, atLeastOnce()).getHeader(8);
21 }
```

Listing 13: The generated **PO** for `loadExisting`

```
17  InOrder orderVerifier = inOrder(mockDataAccess);
18  orderVerifier.verify(mockDataAccess, times(1)).loadExisting();
19  orderVerifier.verify(mockDataAccess, times(3))
        .getHeader(anyInt());
20 }
```

Listing 14: The generated **CO** for `loadExisting`

is indeed the expected value specified by the **OO**. However, this mutant is killed by **PO** and **CO**, as the mock method calls they specify no longer occur within the MUT. Similarly, the **CO** in Listing 14 does not kill mutants #4 and #5 in Listing 10. The method calls specified by the **CO** still occur with the same frequency and in the expected order, but with different parameters. Parameter verification is not the focus of the **CO**, but the **PO** in Listing 13 detects the mutants and fails.

Answer to RQ4

All three types of oracles are effective at detecting regressions introduced within MUTs. Moreover, 16, 18, and 2 mutants are only killed by **OO**, **PO**, and **CO**, respectively. The RICK tests with mock-based oracles complement each other for regression testing, making them a useful addition to the test suite.

5.5 Results for RQ5 [Quality]

Per the protocol described in subsection 4.4, we have interviewed 5 developers between June and July, 2022, with the goal of assessing their opinion on the tests generated by RICK. Table 6 presents the details of the participants of the survey. The five developers work in different sectors of the IT industry, and have between 6 and 30 years of experience with software development. Notably, participant P4 is a core contributor to GRAPHHOPPER, with the highest number of commits to its GitHub repository in the last 5 years. From Table 6, we see that all participants write tests sometimes or everyday, and most of them also define and use mocks. The developers also work with diverse programming environments. P1, P2, and P4 work with Java testing and mocking frameworks, specifically JUnit and Mockito. P3 is a Python developer, while P5, who works in a game development company, works mostly with C, C#, and .NET framework.

We begin each meeting by introducing the concepts and terminology of mock-based testing, the RICK pipeline, and our experiments with GRAPHHOPPER. We demonstrate the features of the tests generated by RICK by selecting a total of 6 generated tests, one for each of the three mock-based oracles for 2 MUTs defined in GRAPHHOPPER. We select the first two MUTs from Table 3 that meet the selection criteria mentioned in subsection 4.4. The two selected MUTs, MUT#16 and MUT#6, have 58 and 13 LOC, respectively. The tests generated for MUT#16 have two stubs, and a method call on an external parameter mock object. The tests for MUT#6 have one stub and three method calls on a mocked field. Excluding comments, the number of lines of code across the six generated tests is 39 (median 6 lines of code for each test). We introduce the two MUTs, MUT#6 and MUT#16, to the participant, also presenting their source code. We invite them to clone a fork of GRAPHHOPPER which includes the generated tests. During the meeting, we browse through the generated tests with them via screen sharing. Finally, we ask the participant three sets of questions about the generated tests while documenting their responses. These questions relate to *mocking effectiveness*, i.e., how mocks are used within the tests, as well as the *structure* and *understandability* of the generated tests.

Mocking effectiveness: The first set of questions relates to how the mocks are used in the generated tests. Per their answers, all five participants agree that the generated tests for the 2 MUTs represent realistic behavior of GRAPHHOPPER in production, which would be useful for developers. P2 observed that this can *"save the time spent on deciding the combination of inputs and finding corner cases, especially for methods with branches."* P5 added that, according to them, collecting data from production is *"where* RICK *shines most, since it abstracts away [for developers] the tricky exercise of deciding test inputs and internal states."* Furthermore, we had detailed discussions with the participants about the verification statements in **PO** and **CO** tests. P1, P3, and P5 noted that they contribute differently to the verification of the behavior of the MUT. P5 remarked that while some verification statements may be redundant, they *"can be manually customized by developers"* when generated tests are presented to them. However, P3, who works primarily with Python, a dynamically-typed language, commented that for them *"the verification of the frequency of the mock method calls in the CO tests is useful, but not the* `anyInt()` *or* `anyString()` *wildcards to match argument types."* Additionally, P3 and P4 also discussed the stability of these tests with respect to code refactoring, with P4 saying that *"the tests might break in case a developer refactors legacy code. But because they are so detailed, a regression can also be figured out at a very low level."* P5 highlighted an interesting aspect about the human element in software development by sharing that *"while* RICK *fits exactly an actual problem in the industry, a potential disadvantage is that it can spoil developers who may become incentivized to design without testability in mind. The tests can be automatically produced later when the application is production-ready."*

Structure: Next, we assess the opinion of developers about the structure of the generated tests. All 5 developers appreciated the "Arrange-Act-Assert" pattern [44] that is systematically followed in the generated tests. They note that this pattern makes the structure of the tests clear. P2, who has experimented with other test generation tools, noted that clear structure and intention is important to improve the adoption of automated tools. All the developers mentioned that they typically use the pattern when writing tests, including P5 who added that the structure was *"spot on."* Moreover, P1, P2, and P4 were appreciative of the description generated by RICK for each test, using the `@DisplayName` JUnit annotation, with P2 noting that *"it is rare and useful."*

Understandability: Finally, we question each participant about the understandability of the generated tests. P1 and P2 mentioned that the comments demarcating each phase in the generated tests contribute to their intuitiveness and make their intention clearer, with P5 exclaiming that they make the tests *"super easy to visualize."* Additionally, P2, P4, and P5 noted that the comments are useful, especially since they help understand what is happening within tests generated automatically. However, P4 observed that while the comments *"do not hurt"*, they would not add them while writing the test manually. P3 was also of the opinion that the comments may be removed without impacting the understandability of the tests.

The perception of developers of automatically generated tests and mocks, using data collected from production, is

TABLE 6: Profiles of the developers who participated in the survey to assess the quality of tests generated by RICK

PARTICIPANT	EXPERIENCE (YEARS)	SECTOR	WRITES TESTS	USES MOCKS	PROGRAMMING ENVIRONMENT
P1	6	Consultancy	*"everyday, testing is [my] life"*	sometimes	`JUnit + Mockito`
P2	30	Consultancy	often	sometimes	`JUnit + Mockito`
P3	7	Telecom	TDD practitioner	sometimes	`Python (pytest)`
P4	10	Product (GRAPHHOPPER)	*"all the time"*	sometimes	`JUnit + Mockito`
P5	14	Game development	sometimes	rarely, *"want to mock more"*	`C, C#, .NET`

valuable qualitative feedback about the relevance of RICK. The 5 experienced developers and testers confirm that the data collected in production is realistic and useful to generate tests and mocks. They also appreciate the systematic structure of the test cases, as well as the explicit intention documented in comments.

This qualitative study also suggests the need for further work in the area of automated mock generation. For example, P3 expressed interest in analyzing the influence of the architecture of the system under test on the stability of the mocks. P4 observed that it would be useful to evaluate the overall testability of an application in terms of how many MUTs have mockable method calls. We also discussed with P5 about adding more context and business-awareness to test names and comments to further help developers troubleshoot test executions. We identify these as excellent directions for further work, with much potential for impact on the industry.

Furthermore, from our discussions with the developers, we find that they all agree that mocking is advantageous. However, developers often rely on metrics such as coverage as a proxy for the strength of their test suite. This leads to a methodological mismatch where mocks are desirable, yet do not contribute directly to the strength of the tests, i.e., do not have an impact on test coverage. We note that an implicit prerequisite for developers to be more open to the benefits of using mocks is to consider test quality beyond coverage, to embrace the value of mocking, as well as the effort required to include mocks in their pipeline. Mocking is not trivial and comes with the challenges highlighted in subsection 2.3. RICK can help with realistic mocks, directly available in readable tests, that can capture regressions.

Answer to RQ5

Five experienced developers confirm that the concept of data collection in production is relevant for the generation of tests with mocks. They all appreciate the systematic "Arrange-Act-Assert" template for the test which contributes to the overall good understandability of the tests generated by RICK.

6 DISCUSSION

We now discuss the performance of RICK, the limitations of our approach for mock generation, and the threats to the validity of our findings.

6.1 Performance

Runtime data capturing is a key process within RICK, which requires some additional computation and memory resources. We adapt the methodology used in previous work [15] to measure the performance implications of RICK during the execution of our three case studies. We exercise each application with the workload described in subsection 4.2 in three distinct ways. First, we determine the baseline performance of the application by running it without any agent attached. Next, we run it with the default monitoring agent, i.e., Glowroot, attached (recall that Glowroot is standard monitoring technology used in the industry). Finally, we attach RICK to the application as a Glowroot plugin, which means the complete monitoring plus data collection machinery for the MUT and mockable method invocations. The experiments are performed on a machine running Ubuntu 22.04, with an 8-core Intel i5 processor and 16GB memory. We use the Linux `top` command, filtered on the application name, to obtain its CPU and memory usage.

For GRAPHHOPPER, the average CPU consumption for the baseline execution is 35.4% while the memory usage is 824.7 MB. Attaching Glowroot as a monitoring agent to GRAPHHOPPER increases the CPU and memory consumption to 66% and 983.4 MB. Attaching the complete monitoring and data capturing abilities of RICK results in the CPU and memory usage of 109.2% and 1570.2 MB. This means that the execution with RICK and all MUTs and mockable methods monitored, consumes thrice the CPU compared to baseline, and twice the memory. Next, normal execution of GEPHI consumes 117.6% CPU and 856.5 MB memory on average. Monitoring with Glowroot increases these usages to 142.2% and 1157.8 MB, respectively. Attaching RICK with GEPHI results in 249.1% CPU and 1601.9 MB memory usage. The resource consumption during execution of GEPHI with RICK is about twice the baseline amount. Finally, averaging across 10 executions of PDFBOX, we find that its CPU consumption is 92%, while its memory usage is 63.4 MB. Attaching Glowroot increases CPU and memory consumption to 335.4% and 190.3 MB, respectively. Attaching RICK does not contribute to additional CPU consumption, and leads to an increased memory usage at 428.2 MB (about 6.7 times the baseline). Overall, we note that monitoring contributes to additional resource consumption for all cases, with respect to the baseline. This increases more with RICK as a consequence of dynamic instrumentation and serialization.

Monitoring is an essential component of modern ob-

servability, for ensuring smooth operation and diagnosis [45]. RICK leverages monitoring to generate unit tests that reflect production behaviors and detect regressions. However, monitoring comes at a cost, impacting the scalability of RICK, as well as similar approaches for observation-based test generation [15], [27]. We have taken measures to mitigate this issue, designing RICK to be configurable. First, developers configure the target methods to monitor and capture data for, and only those are instrumented. Additionally, they also specify the number of invocations of these methods that must be monitored in production. Being a research prototype, it is clear that the monitoring agent has ample room for performance optimization when productized. Furthermore, as we highlight in subsection 3.4, RICK is only meant to be periodically employed, for every testing campaign when the development team focuses on improving unit tests with production data.

6.2 Subsequent Test Failures

As we demonstrate with RQ4, the potential future failures of a test generated by RICK are meant to be indicative of regressions in the method under test, compared to the current behavior. However, there are two cases when a RICK test would fail in the absence of a regression.

First, for mocked methods that perform non-deterministic actions, such as network calls, the stubs and mock-based oracles generated by RICK reflect the responses observed in the field, by design. For instance, calls that result in valid responses (2XX) or 4XX errors, such as a 404 for a non-existent resource, will lead to the generation of useful tests with RICK. However, RICK will generate a test that may overfit a situation with a 5XX error. Such a response might not be reproduced within the generated test, causing it to fail.

Second, the failure of a RICK-generated test may be the result of a refactoring [46], and not a regression. For example, for a method under test m, the call oracle will fail if the order of the mock method calls within m is changed, while the semantics of m is preserved. Also, the parameter oracle may not hold if the arguments received by at least one mock method call within m deviate from their expected values. This can indicate a behavioral change within m, but also a refactoring of the stubbed method call. The parameter and call oracles are sensitive to refactoring changes. However, a failing output oracle implies that the output from m is unequal to the expected output, signifying a behavioral change in the public API.

Consider lines 1 to 9 of Listing 15, which present the method `getWidthFromFont` of PDFBOX (MUT#27 in Table 5). This MUT calls two mock methods on the field `ttf` of type `TrueTypeFont`, `getAdvanceWidth` (line 4) and `getUnitsPerEm` (line 5). The mock-based oracles generated by RICK for `getWidthFromFont` are presented on line 15 (**OO**), lines 22-23 (**PO**), and lines 30-32 (**CO**). Listing 16 shows a refactored version of `getWidthFromFont`, with two semantically-preserving changes: 1) the mock method calls are reordered, i.e., `getUnitsPerEm` occurs first (line 4), and 2) `getAdvanceWidth` is renamed to `calculateAdvanceWidth` (line 5). These changes cause a failure of the **CO** and **PO**, but the **OO** still holds.

```
1 public float getWidthFromFont(int code) {
2   int gid = codeToGID(code);
3   float width = ttf.getAdvanceWidth(gid);
4   float unitsPerEM = ttf.getUnitsPerEm();
5   if (unitsPerEM != 1000) {
6     width *= 1000f / unitsPerEM;
7   }
8   return width;
9 }
10 ...............................................................................
11 @Test
12 public void testGetWidthFromFont_OO() {
13   ...
14   // Assert
15   assertEquals(750.0, actual, 0.0);
16 }
17 ...............................................................................
18 @Test
19 public void testGetWidthFromFont_PO() {
20   ...
21   // Assert
22   verify(mockTTF, atLeastOnce()).getAdvanceWidth(0);
23   verify(mockTTF, atLeastOnce()).getUnitsPerEm();
24 }
25 ...............................................................................
26 @Test
27 public void testGetWidthFromFont_CO() {
28   ...
29   // Assert
30   InOrder ordVerifier = inOrder(mockTTF);
31   ordVerifier.verify(mockTTF,
         times(1)).getAdvanceWidth(anyInt());
32   ordVerifier.verify(mockTTF, times(1)).getUnitsPerEm();
33 }
```

Listing 15: RICK generates tests with the three mock-based oracles for MUT#27 of PDFBOX, `getWidthFromFont`. Refactoring `getWidthFromFont` can impact the validity of these oracles.

```
1 // Refactored getWidthFromFont
2 public float getWidthFromFont(int code) {
3   int gid = codeToGID(code);
4   float unitsPerEM = ttf.getUnitsPerEm(); // reordered call
5   float width = ttf.calculateAdvanceWidth(gid); // renamed
        method
6   if (unitsPerEM != 1000) {
7     width *= 1000f / unitsPerEM;
8   }
9   return width;
10 }
```

Listing 16: MUT#27 of PDFBOX, `getWidthFromFont`, after refactoring. Relative to Listing 15, its mockable method calls `getAdvanceWidth` and `getUnitsPerEm` are reordered, and the former is renamed to `calculateAdvanceWidth`.

Despite their varying degree of sensitivity, all three mock-based oracles alert the developer of behavioral changes that are introduced in their code, which is also the goal of testing. The impact of code refactoring on mock-based oracles is an important direction for future work.

6.3 Threats & Limitations

Serialization The internal validity of RICK is impacted by technical limitations. For example, instrumentation of some methods may fail [47]. One of the biggest technical challenge is the serialization and deserialization of large and complex objects. This may result in incomplete or unfaithful program states within the generated tests, which do not reflect the ones observed in production.

Application Vs Library A source of threat to the external validity of our findings arises from the software projects we consider for the evaluation of RICK. We make sure that the

three projects are 1) complex, and 2) from diverse domains. Indeed, we consider a library, a desktop application, and a backend application. We do not guarantee that our findings hold for applications and libraries in other languages such as TypeScript, or nim. Still, we believe that RICK would not let them down.

Program Evolution Tests and programs co-evolve within software projects [48]–[50]. The tests generated by RICK are no different, they may be impacted by changes in the application code. A refactoring in the source code, such as a modification in the order of method invocations, or a change in the parameters of an existing method, may require changes in the generated tests. However, if these modifications are not semantically relevant, this is tedious, low value work for developers. This limitation is shared by all regression test generation techniques. In this case, developers can always regenerate new tests when significant changes are made to the methods under tests.

7 Related Work

This section presents the literature on mock objects, as well as their automated generation. We also discuss studies about the use of information collected from production for the generation of tests.

7.1 Studies on Mocking

Since mocks were first proposed [1], they have been widely studied [2], [51]. Their use has been analyzed for major platforms, such as Java [19], Python [52], C [53], Scala [54], Android [55], [56], PHP, and JavaScript [57]. These studies highlight the prevalence and practices of defining and using mocks. Some empirical studies analyze more specific aspects about the usage of mocks, such as their definition through developer-written mock classes [18], or mocking frameworks [20], or their use in the replacement of calls to the file system [58]. Xiao *et al.* [31] find that mocking is practiced in 66% of the 264 projects of the Apache Software Foundation. Spadini *et al.* discuss the criteria developers consider when deciding what to mock [3], as well as how these mocks evolve [19]. MockSniffer by Zhu *et al.* [59] uses machine learning models to recommend mocks. Mockingbird by Lockwood *et al.* [60] uses mocks to isolate the code under dynamic analysis from its dependencies. The use of mocks for Test Driven Development [61], modeling [62], and as an educational tool for object-oriented design [63], [64] has also been investigated. These works have been inspirational for us in many respects. Moreover, RICK is designed to generate tests that use Mockito, which is the most popular mocking framework [20], [31], [57], and is itself a subject of study [65]–[68]. However, none of these related works touch upon mocking in the context of production monitoring, specifically generating mocks and mock-based oracles. These are the two key contributions of our work.

7.2 Mock Generation

Several studies propose approaches to automatically generate mocks, albeit not from production executions. For example, search-based test generation can be extended to include mock objects [21], to mock calls to the file system [6] and the network [7]. Symbolic execution may also be used to generate mocks [8], [22], [69], to mock the file system [70], or a database for use in tests [71]. Honfi and Micskei [72] generate mocks to replace external interactions with calls to a parameterized sandbox. This sandbox receives inputs from the white-box test generator, Pex [23]. Moles by Halleux and Tillmann [73] also works with Pex to isolate the unit under test from interactions with dependencies by delegating them to alternative implementations. Salva and Blot [74] propose a model-based approach for mock generation. Bhagya *et al.* [75] use machine learning models to mock HTTP services using network traffic data. GenUTest [76] generates JUnit tests and mock aspects by capturing the interactions that occur between objects during the execution of medium-sized Java programs. StubCoder [5] by Zhu *et al.* uses an evolutionary algorithm to generate new stubs and repair incorrect stubs within existing JUnit tests. On the other hand, ARUS [77] utilizes information from the execution of the test suite to detect and remove unnecessary stubbing. Abdi and Demeyer [78] leverage mocking within their proposed test transplantation technique that ports client tests into library test suites. ARTISAN by Gambi *et al.* [79] instruments end-to-end GUI tests of Android applications, in order to carve unit tests that mock classes of the Android framework.

To the best of our knowledge, RICK is the only tool that generates mock-based oracles to verify the behavior of the system under test, per the the production executions, with real usages and real data.

The definition and behavior of mocks can also be extracted from other artifacts. For instance, the design contract of the type being mocked can be used to define the behavior of a mock [80]. Samimi *et al.* [81] propose declarative mocking, an approach that uses constraint solving with executable specifications of the mock method calls. Solms and Marshall [82] extract the behavior of mock objects from interfaces that specify their contract. Wang *et al.* [83] propose an approach to refactor out developer-written subclasses that are used for the purpose of mocking, and replace them with Mockito mocks. Mocks have also been generated in the context of cloud computing [84], such as for the emulation of infrastructure by MockFog [16]. Jacinto *et al.* [85] propose a mock-testing mode for drag-and-drop application development platforms. Contrary to these approaches, RICK monitors applications in production in order to generate mocks. Consequently, the generated tests reflect the behavior of an application with respect to actual user interactions.

The executions of system tests in the existing test suite can also be leveraged to generate mocks. This approach has been used by Saff *et al.* [9] through system test executions, and Fazzini *et al.* [55] to generate mocks for mobile applications. Bragg *et al.* [86] use the test suite of Sketch programs to generate mocks in order to modularize program synthesis. However, system tests are artifacts written by developers, and can therefore suffer from biases that developers have about how the system should behave. In contrast, production executions, where RICK sources its test inputs, are free from these assumptions, and reflect how the system actually behaves under real workloads.

7.3 Capture and Replay

Many studies propose techniques to capture a sequence of events that occur within an executing system, with the goal of replaying it [87]. The premise of these techniques is to replicate the state of the system as it was at a certain point in time. Capture and replay has been successfully applied for the reproduction of crashes [88] and failures [89] that occur in the field. The captured sequence of events leading up to a crash or failure allows for more efficient debugging when replayed offline by developers [90], [91], as well as the evaluation of candidate patches for bugs [92]. Capture and replay can also be used to exercise the same sequence of interactions with an application GUI as was done by end-users [93]. This can be used to analyze the performance of interactive applications [94]. All these existing techniques do not generate an explicit oracle. They instead rely on an implicit oracle, such as the reproduction of a failure. Saff *et al.* [9] carve focused unit tests from system tests. Their technique cannot be applied to production environments without major challenges. In particular, a key challenge that we address with RICK is the serialization of production objects with reasonable overhead. This challenge is also noted by Meta, who propose TestGen for generating observation-based tests for Instagram [27]. This aspect, together with our specific oracles, are fundamentally novel compared to the technique of Saff *et al.* [9]. In addition, their evaluation considers one program, while our evaluation considers three real-world programs exercised with representative field workloads.

RICK is fundamentally different from capture and replay techniques since it generates full-fledged test cases, which include an explicit oracle in an assertion. This essential difference allows us to assess the effectiveness of the generated tests with mutation analysis, which none of the capture and replay techniques do.

7.4 Production-based Oracles

Monitoring an executing application with the goal of generating tests is an effective means of bridging the gap between the developers' understanding of their system, and how it is actually exercised by users [13]. To this end, several studies propose tools that capture runtime information. Thummalapenta *et al.* [95] use execution traces for the generation of parameterized unit tests. Wang and Orso [14] capture the sequence of method executions in the field, and apply symbolic execution to generate tests for untested behavior. Jaygarl *et al.* [96] capture objects from program executions, which can then be used as inputs by other tools for the generation of method sequences. Tiwari *et al.* [15] generate tests for inadequately tested methods using production object states. PRODJ by Wachter *et al.* [47] focus on the readability of the unit tests generated from production data, incorporating the objects captured at runtime as plain Java code. Incoming production requests have also been utilized to produce tests for databases [97] and web applications [26]. RICK leverages this methodology with the novel and specific goal of generating tests with mock-based oracles that verify the interactions between a method and objects of external types, as they occur in production.

8 Conclusion

In this paper, we present RICK, a novel approach for generating tests with mocks, using data captured from the observation of applications executing in production. RICK instruments a set of methods under test, monitors their invocations in production, and captures data about the methods and the mockable method calls. Finally, RICK generates tests using the captured data. The mock-based oracles within the generated tests verify distinct aspects of the interactions of the method under test with the external object, such as the output of the method (**OO**), the parameters with which invocations are made on the external object (**PO**), and the sequence of these invocations (**CO**). Our evaluation with three open-source applications demonstrates that RICK never gives up: It monitors and transforms observed production behavior into concrete tests (RQ1). The data collected from production is expressed within these generated tests as complex receiving objects and parameters for the methods, as well stubs and mock-based oracles (RQ2). When executed, 52.4% of the generated tests successfully mimic the observed production behavior. This means that they recreate the execution context for the method under test, the stubbed behavior is appropriate, and the oracle verifies that the method under test behaves the same way as it did in production (RQ3). The three mock-based oracles can detect regressions within the methods under test, and **OO**, **PO**, and **CO** can complement each other in finding bugs (RQ4). Furthermore, our qualitative survey with professional software developers reveals that the data and oracle extracted from production by RICK are relevant, and that the systematic structure of RICK tests is understandable (RQ5).

Overall, we are the first to demonstrate the feasibility of creating tests with mocks directly from production, in other terms to capture production behavior in isolated tests. Since the generated tests reflect the actual behavior of an application in terms of concrete inputs and oracles, they are valuable for developers to augment manually crafted inputs with ones that are relevant in production.

Our findings open up several opportunities for more research. It would be useful to handle more kinds of interactions of a method under test with its environment, such as all method calls made on an external object within the method under test, in order to achieve further isolation within the generated tests. Future work should also consider different choices of mockable method calls, to support mocking types within dependencies. Additionally, the impact of code refactoring on automatically generated mock-based oracles warrants a detailed analysis.

Acknowledgements

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, as well as by the Chains project funded by the Swedish Foundation for Strategic Research (SSF).

References

[1] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.

[2] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[3] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 402–412.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[5] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

[6] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 79–90.

[7] ——, "Generating tcp/udp network data for automated unit test generation," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 155–165.

[8] M. Islam and C. Csallner, "Dsc+ mock: A test case+ mock class generator in support of coding against interfaces," in *Proceedings of the Eighth International Workshop on Dynamic Analysis*, 2010, pp. 26–31.

[9] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 114–123.

[10] M. Fowler, "Mocks aren't stubs," https://martinfowler.com/articles/mocksArentStubs.html, date accessed September 1, 2024.

[11] M. Christakis, P. Emmisberger, P. Godefroid, and P. Müller, "A general framework for dynamic stub injection," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 586–596.

[12] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[13] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?" in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 321–332.

[14] Q. Wang and A. Orso, "Improving testing by mimicking user behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 488–498.

[15] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Transactions on Reliability*, pp. 1–17, 2021.

[16] J. Hasenburg, M. Grambow, and D. Bermbach, "Mockfog 2.0: Automated execution of fog application experiments in the cloud," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.

[17] N. E. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 2–26.

[18] G. Pereira and A. Hora, "Assessing mock classes: An empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 453–463.

[19] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1461–1498, 2019.

[20] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[21] A. Arcuri, G. Fraser, and R. Just, "Private API access and functional mocking in automated unit test generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2017, pp. 126–137.

[22] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 365–368.

[23] N. Tillmann and J. d. Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[24] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 193–202.

[25] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1281–1291.

[26] L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry, "Harvesting production GraphQL queries to detect schema faults," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 365–376.

[27] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based unit test generation at meta," in *Proceedings of ESEC/FSE*, 2024.

[28] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[29] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, no. 1, 2009, pp. 361–362.

[30] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.

[31] L. Xiao, K. Li, E. Lim, X. Wang, C. Wei, T. Yu, and X. Wang, "An empirical study on the usage of mocking frameworks in apache software foundation," *Available at SSRN 4100265*.

[32] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR'19. IEEE Press, 2019, p. 344–348.

[33] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR'19. IEEE Press, 2019, p. 333–343.

[34] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[36] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*. Springer, 2017, pp. 148–163.

[37] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *Empirical Software Engineering*, vol. 24, pp. 1195–1225, 2019.

[38] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2021.

[39] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 870–880.

[40] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Automated oracle data selection support," *IEEE Trans. Software Eng.*, vol. 41, no. 11, pp. 1119–1137, 2015.

[41] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[42] H. Du, V. K. Palepu, and J. A. Jones, "Ripples of a mutation—an empirical study of propagation effects in mutation testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[43] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "Suggestions on test suite improvements with automatic infection and propagation analysis," 2019. [Online]. Available: https://arxiv.org/abs/1909.04770

[44] C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically Tagging the "AAA" Pattern in Unit Test Cases Using Machine Learning Models," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–3.

[45] L. Maguire, "Automation doesn't work the way we think it does," *IEEE Software*, vol. 41, no. 01, pp. 138–141, 2024.

[46] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[47] J. Wachter, D. Tiwari, M. Monperrus, and B. Baudry, "Serializing java objects in plain code," *arXiv preprint arXiv:2405.11294*, 2024.

[48] S. Shimmi and M. Rahimi, "Leveraging code-test co-evolution patterns for automated test case recommendation," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 65–76.

[49] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[50] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 206–216.

[51] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, pp. 236–246.

[52] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 207–218.

[53] S. Mudduluru, "Investigation of test-driven development based on mock objects for non-oo languages," Master's thesis, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2012.

[54] K. Laufer, J. O'Sullivan, and G. K. Thiruvathukal, "Tests as maintainable assets via auto-generated spies: A case study involving the scala collections library's iterator trait," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, 2019, pp. 17–21.

[55] M. Fazzini, A. Gorla, and A. Orso, "A framework for automated test mocking of mobile apps," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1204–1208.

[56] M. Fazzini, C. Choi, J. M. Copia, G. Lee, Y. Kakehi, A. Gorla, and A. Orso, "Use of test doubles in android testing: An in-depth investigation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2266–2278.

[57] R. De Almeida, R. M. Da Silva, L. S. Serrano, H. De Souza Campos Junior, and V. De Oliveira Neves, "Mock objects in software testing: An analysis of usage in open-source projects," in *Proceedings of the XXII Brazilian Symposium on Software Quality*, 2023, pp. 72–79.

[58] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *2009 ICSE Workshop on Automation of Software Test*. IEEE, 2009, pp. 149–153.

[59] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, and C. Zhou, "MockSniffer: Characterizing and recommending mocking decisions for unit tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 436–447.

[60] D. Lockwood, B. Holland, and S. Kothari, "Mockingbird: a framework for enabling targeted dynamic analysis of java programs," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 39–42.

[61] T. Kim, C. Park, and C. Wu, "Mock object models for test driven development," in *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*. IEEE, 2006, pp. 221–228.

[62] J. Stoel, T. van der Storm, and J. Vinju, "Modeling with mocking," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 59–70.

[63] J. Nandigam, V. N. Gudivada, A. Hamou-Lhadj, and Y. Tao, "Interface-based object-oriented design with mock objects," in *2009 Sixth International Conference on Information Technology: New Generations*. IEEE, 2009, pp. 713–718.

[64] J. Nandigam, Y. Tao, V. N. Gudivada, and A. Hamou-Lhadj, "Using mock object frameworks to teach object-oriented design principles," *The Journal of Computing Sciences in Colleges*, vol. 26, no. 1, pp. 40–48, 2010.

[65] G. Gay, "Challenges in using search-based test generation to identify real faults in mockito," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 231–237.

[66] A. J. Turner, D. R. White, and J. H. Drake, "Multi-objective regression test suite minimisation for mockito," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 244–249.

[67] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 260–266.

[68] D. J. Kim, N. Tsantalis, T.-H. P. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 62–73.

[69] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, "AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation," in *Practical Software Testing : Tool Automation and Human Factors*, ser. Dagstuhl Seminar Proceedings (DagSemProc), vol. 10111, 2010, pp. 1–4.

[70] S. Kong, N. Tillmann, and J. de Halleux, "Automated testing of environment-dependent programs - a case study of modeling the file system for pex," in *2009 Sixth International Conference on Information Technology: New Generations*, 2009, pp. 758–762.

[71] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 289–292.

[72] D. Honfi and Z. Micskei, "Automated isolation for white-box test generation," *Information and Software Technology*, vol. 125, p. 106319, 2020.

[73] J. d. Halleux and N. Tillmann, "Moles: tool-assisted environment isolation with closures," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2010, pp. 253–270.

[74] S. Salva and E. Blot, "Using model learning for the generation of mock components," in *IFIP International Conference on Testing Software and Systems*. Springer, 2020, pp. 3–19.

[75] T. Bhagya, J. Dietrich, and H. Guesgen, "Generating mock skeletons for lightweight web-service testing," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 181–188.

[76] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: a unit test and mock aspect generation tool," *International journal on software tools for technology transfer*, vol. 11, no. 4, pp. 273–290, 2009.

[77] M. Li and M. Fazzini, "Automatically removing unnecessary stubbings from test suites," *arXiv preprint arXiv:2407.20924*, 2024.

[78] M. Abdi and S. Demeyer, "Test transplantation through dynamic test slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 35–39.

[79] A. Gambi, H. Gouni, D. Berreiter, V. Tymofyeyev, and M. Fazzini, "Action-based test carving for android apps," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 107–116.

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

[81] H. Samimi, R. Hicks, A. Fogel, and T. Millstein, "Declarative mocking," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 246–256.

[82] F. Solms and L. Marshall, "Contract-based mocking for services-oriented development," in *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, 2016, pp. 1–8.

[83] X. Wang, L. Xiao, T. Yu, A. Woepse, and S. Wong, "An automatic refactoring framework for replacing test-production inheritance by mocking mechanism," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 540–552.

[84] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. De Halleux, "Environmental modeling for automated cloud application testing," *IEEE software*, vol. 29, no. 2, pp. 30–35, 2011.

[85] A. Jacinto, M. Lourenço, and C. Ferreira, "Test mocks for low-code applications built with outsystems," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–5.

[86] N. F. Bragg, J. S. Foster, C. Roux, and A. Solar-Lezama, "Program sketching by automatically generating mocks from tests," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 808–831.

[87] S. Joshi and A. Orso, "Scarpe: A technique and tool for selective capture and replay of program executions," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 234–243.

[88] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej, "Monitoring user interactions for supporting failure reproduction," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 73–82.

[89] J. Bell, N. Sarda, and G. Kaiser, "Chronicler: Lightweight recording to reproduce field failures," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 362–371.

[90] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.

[91] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 2013, pp. 473–484.

[92] A. Saieva and G. Kaiser, "Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting," *Journal of Systems and Software*, vol. 191, p. 111381, 2022.

[93] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jrapture: A capture/replay tool for observation-based testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000, pp. 158–167.

[94] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated gui performance testing," *Software Quality Journal*, vol. 19, no. 4, pp. 801–839, 2011.

[95] S. Thummalapenta, J. d. Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *International Conference on Tests and Proofs*. Springer, 2010, pp. 77–93.

[96] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "Ocat: object capture-based automated testing," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 159–170.

[97] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee, "Snowtrail: Testing with production queries on a cloud database," in *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.