Abstractions for Software Testing

DISSERTATION FOR THE AWARD OF THE DOCTORAL DIPLOMA
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

2022

Thodoris Sotiropoulos
Department of Management Science and Technology
Athens University of Economics and Business

Department of Management Science and Technology
Athens University of Economics and Business
Email: theosotr@aueb.gr

Supervised by Prof Diomidis Spinellis

Dedicated to my friends
for their unconditional support.

Contents

List of Figures

List of Tables

Acknowledgements

First, I am grateful for three people: my advisor, Dimitris, and Stefanos, who significantly helped throughout this PhD journey, each of them with their own way. I would not have been able to complete these studies without their continuous help and support.

I would like to thank my advisor, Prof Diomidis Spinellis, for providing me with a platform for conducting research. Prof Spinellis trusted me since the early days of my PhD studies, and allowed me to follow my own research agenda, even though he was reluctant in certain cases. Having the freedom to follow your own research path is of high importance for three main reasons. First, it helps you become an independent researcher. Second, it is a highly rewarding learning process, as you learn from your own failure. Third, doing what you love is the key of success. As a good advisor and manager, Prof Spinellis boosted my productivity, as he tackled any problems that prevented me from being effective in my job. Through his help and encouragement, I was able to focus solely on research and things that matter to me. I still remember his idea about detecting dependency bugs in Puppet programs, which was the first project of my PhD. His words were: *"configuration management systems are understudied in academia, so it's totally worth building something for them."* This advice and project were a great boost for my PhD. Notably, this was the PhD project I enjoyed the most. Finally, I would like to thank Prof Spinellis for all his support and guidance throughout all the years I know him: from the second year of my undergraduate studies till now.

Now, I would like to thank Dimitris Mitropoulos. I decided to pursue a PhD thanks to Dimitris, because he was the one who introduced research to me. Specifically, I was an undergraduate student at the Department of Management Science and Technology, when I participated in a research project led by Dimitris. This is the moment when I realized that I enjoy solving research problems. Through Dimitris' guidance, I learned how to conduct research, and effectively describe my work in the paper. In particular, my writing skills were substantially improved via endless suggestions made by Dimitris. Dimitris is now my mentor since my undergraduate studies. I have been always seeking his advice and help for various matters, including my undergraduate and postgraduate studies, job opportunities, research, or other personal issues. I really appreciate his support for what I have achieved so far.

Last but not least, I would like to specially thank my friend Stefanos Chaliasos. I closely collaborated with Stefanos in four out of five papers of my PhD. Stefanos helped me overcome many technical challenges associated with the implementation of the techniques presented in this thesis. Stefanos is the only guy I trust when it comes to engineering work that involves tons of technical difficulties. As time passed, Stefanos took a more active role in the papers,

especially in our award-winning work on compiler testing. I am not sure I would have been able to finish my PhD studies in a timely manner without Stefanos. Finally, Stefanos and I had much fun in the lab, and he was always there to discuss any personal affairs.

I also want to thank Dr Panos Louridas as he is one of those who first believed in me. Even though I was still an undergraduate student, he helped me join the Greek Research and Technology Network (GRNET) as a software engineer. This was the first professional experience I ever had, and my time there significantly improved my technical skills. I would also like to thank him for his invaluable guidance on our paper about search engine similarity analysis.

I would like thank the members of the lunch club: Vaggelis Atlidakis, Charalambos Mitropoulos, and Konstantina Dritsa. We spent great time together having fun and relaxing. Special thanks to Vaggelis for tolerating my endless research concerns. I also want to thank the older members of the Lab: Konstantinos Kravvaritis, Stefanos Georgiou, and Antonis Gkortzis. All of them wholeheartedly welcomed me when I joined the Lab. I still remember all of the fun discussions I had with Konstantinos and Stefanos. I also want to thank Konstantinos for assisting me during the tutorial and lab sessions of an undergraduate course called "Programming II". Lastly, I would like to thank Giorgos Drosos, a brilliant student who contributed to our study of compiler typing bugs. Giorgos is always eager to help and learn new things. I am sure that he will succeed in the future.

This thesis would not have been possible without the funding from: (1) European Union's Horizon 2020 research and innovation programs under grant agreement No. No. 732223, and (2) European Union's Horizon 2020 research and innovation programs under grant agreement No. 825328.

Abstract

Developers and practitioners spend considerable amount of their time in testing their software and fixing software bugs. To do so more effectively, they automate the process of finding deep software bugs that are challenging to uncover via manually-written test cases by integrating automated bug-finding tools in the software development process. A challenge of automated bug detection is the identification of *subtle* and *latent* defects in software that involves complex functionality. This is because such bugs are easy to remain unnoticed as the software under test does not complain with warnings or other runtime failures (e.g., crashes) during its execution. Worse, subtle defects often confuse users who do not blame the buggy software for the unexpected behavior, because they believe that the error is from their side (e.g., wrong input is given). Another shortcoming of many existing bug-finding tools is their limited applicability. Indeed, many of them are tailored to specific piece of software. This lack of applicability is mainly attributed to fundamental issues related to the design of the underlying methods.

To tackle the aforementioned issues, this thesis investigates ways for improving the effectiveness and applicability of automated software testing by introducing different forms of abstractions in testing workflow. The aim of these abstractions is to provide a common platform for reasoning and identification of (subtle) bugs in software systems and programs that exhibit dissimilar interfaces, implementations, or semantics. The thesis demonstrates this concept by applying abstractions in the context of three important problems: the detection of (1) compiler typing bugs, (2) bugs in data-centric software, and (3) dependency bugs in file system resources. This is achieved through the design and development of three bug-finding tools. In particular:

First, we introduce Hephaestus, a testing framework for validating static typing procedures in compilers. Our core component is a program generator suitably crafted for producing programs that are likely to trigger compiler typing bugs. One of our main contributions is that our program generator gives rise to transformation-based compiler testing for finding typing bugs. We present two novel approaches (*type erasure mutation* and *type overwriting mutation*) that apply targeted transformations to an input program to reveal type inference and soundness compiler bugs respectively. Both approaches are guided by an intra-procedural type inference analysis used to capture type information flow. The design of Hephaestus is guided by the observations and findings of what is to the best of our knowledge, the first empirical study for understanding and characterizing compiler typing bugs. To do so, we manually study 320 typing bugs (along with their fixes and test cases) that are randomly sampled from four mainstream JVM languages, namely Java, Scala, Kotlin, and Groovy. We evaluate each bug in terms of several aspects, including their symptom, root cause, bug fix's size, and the characteristics

of the bug-revealing test cases.

Second, we introduce CYNTHIA, the first approach for systematically testing Object-Relational Mapping (ORM) systems. Our approach leverages differential testing to establish a test oracle for ORM-specific bugs. Specifically, we first generate random relational database schemas, set up the respective databases, and then, we query these databases using the APIs of the ORM systems under test. To tackle the challenge that ORMs lack a common input language, we generate queries written in an abstract query language. These abstract queries are translated into concrete, executable ORM queries, which are ultimately used to differentially test the correctness of target implementations. The effectiveness of our method heavily relies on the data inserted to the underlying databases. Therefore, we employ a solver-based approach for producing targeted database records with respect to the constraints of the generated queries.

Third, we introduce FSMOVE, a generally-applicable model for reasoning about file system-intensive executions. FSMOVE takes into account the specification (as declared in a high-level program) and the actual behavior (low-level file system operation) of operations. We then formally define different types of dependency bugs that lead to mishandling of file system resources in the context of two everyday development tasks: automated builds and automated system configuration. These dependency bugs are defined in terms of the conditions under which a file system operation violates the specification of a build or a configuration management operation. Our testing approach, which relies on the proposed model, analyzes a build or a configuration management execution, translates it into FSMOVE, and uncovers dependency bugs by checking for corresponding violations.

The work presented in this thesis substantially improved the reliability of well-established and critical software used by millions of users and applications. Overall, our bug-finding techniques and tools led to the disclosure and fix of more than 400 bugs found in systems, such as the Java and the Groovy compilers, the Django web framework, or dozens of popular configuration management libraries used for managing critical infrastructures (e.g., the Apache server). This thesis exhibits practical impact on the software industry, benefits real-world developers, and opens new research opportunities associated with the application of programming language concepts to automated software testing and software reliability.

Chapter 1

Introduction

Software is by far one of the most elaborate human artifacts ever made. In today's world, the majority of software consists of million of lines of code, each of them affects diverse entities, including the runtime, the operating system, (transitive) software dependencies, and external services, such as database or web services. All these components must interact well with each other to provide millions of users with the intended functionality.

Due to this inherent software complexity, software defects, simply known as *software bugs*, come inevitably into play. A bug makes software not behave as expected leading to unexpected outputs and functional errors, misuses of system resources (e.g., memory [Durumeric et al., 2014], file system [Kellogg et al., 2021]), performance degradation, and other abnormal executions, such as runtime failures and crashes. Depending on its nature, a software bug can exhibit severe consequences on the users and applications that rely on a buggy software. Some of the most pernicious results of software bugs include disclosure of sensitive data [Durumeric et al., 2014], money loss [Liu et al., 2018], or even risk to human lives [Leveson and Turner, 1993].

To boost software reliability and quality, practitioners and researchers have spent much time on developing bug detection and prevention techniques. Applying automated software testing [Visser et al., 2016b; Ke Mao, 2018] or dynamic [Serebryany et al., 2012; Chabbi and Ramanathan, 2022] and static [Ayewah et al., 2008] program analysis techniques before shipping software into production is now part of almost every software development lifecycle. For example, Sapienz [Mao et al., 2016], an automated testing tool for Android applications has integrated into Meta's testing pipeline [Ke Mao, 2018]. Sapienz employs a meta-heuristic algorithm to automatically generate inputs to explore program areas that are often neglected by manually-written test cases during unit testing. Google acquired the `GraphicsFuzz` technology [Donaldson et al., 2017; Donaldson and Lascu, 2016], which is based on randomized testing. Using `GraphicsFuzz`, Google improved the effectiveness of the Vulkan Conformance Test Suite [Khronos Group, 2022] by adding test cases that uncover new bugs or exercise unexplored code in graphics driver implementations. Chabbi and Ramanathan [2022] explain how Uber deploys the dynamic race detector supported by Golang to uncover thousands of data races in Uber's huge codebase written in Golang.

Unfortunately, the majority of bug-finding methods involve some major challenges when it comes to the detection of bugs other than crashes and runtime failures. Automated software

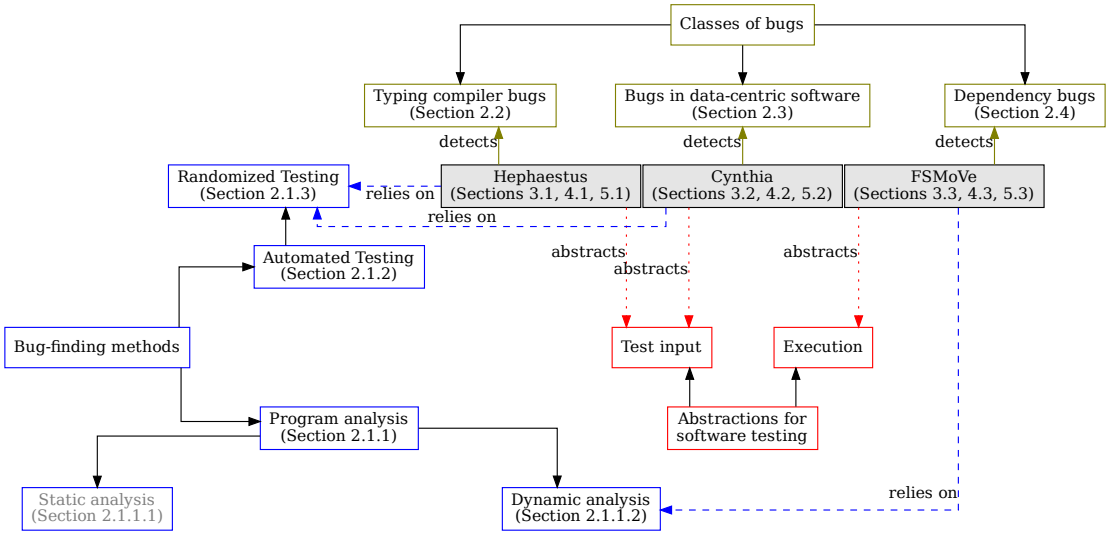


Figure 1.1: Thesis map: The bug-finding tools proposed in this thesis along with (1) the underlying bug-finding method they rely on, (2) the classes of bugs they detect, and (3) the entities they abstract.

testing relies on a *test oracle* to distinguish a buggy behavior from the expected one. Nevertheless, a test oracle is not always available, especially when dealing with a software under test that *subtly* fails, or exhibits a latent bug without raising any error. At the same time, many existing bug-finding techniques are tailored to identifying a particular class of bugs found only in a certain piece of software. Therefore, it is ineffective to apply the proposed methods to other similar software systems. The issues of test oracle specification and applicability make many automated software testing methods face adoption issues, and prevent them from being useful in practice.

In this thesis, we tackle the aforementioned issues by designing bug-finding methods guided by two types of abstractions, namely, test input abstraction, and execution abstraction. The purpose of these abstractions is to facilitate the reasoning required for performing bug identification, while preserving the applicability of our testing efforts to software systems with dissimilar interfaces, implementations, or semantics. To demonstrate how abstractions can improve the effectiveness of software testing, we develop Hephaestus, cynthia, and fsmove, three testing tools for identifying three important classes of bugs: *compiler typing bugs*, *bug in data-centric software*, and *dependency bugs in file system resources* respectively. Our tools are underpinned by well-established methods, primarily *dynamic program analysis* and *randomized testing*, which have been broadly used in the literature for automated bug detection (see Section 2.1). A diagrammatic synopsis of the methods described in this thesis is shown in Figure 1.1.

We argue that having a sufficient understanding about the essence of bugs we aim to detect, including bug manifestations, bug causes, bug importance, or bug triggers, is of paramount importance when designing and developing effective bug-finding tools. To this end, this thesis also presents a method for (1) collecting existing bugs found in complex software, (2) extracting

essential information from the collected bugs through manual bug analysis, (3) leveraging the extracted knowledge for the design of future automated testing techniques.

We now provide an introduction of the three problems tackled in the thesis.

1.1 Finding Compiler Typing Bugs

Compiler reliability has a tremendous impact on the entire software ecosystem. To this end, *compiler testing* has substantially thrived since the beginning of the last decade [Chen et al., 2020], when Csmith [Yang et al., 2011], the most well-known program generator for C programs, first appeared. Csmith has paved the way for advanced program generation [Livinskii et al., 2020; Lidbury et al., 2015a; Even-Mendoza et al., 2020], transformation-based compiler testing [Le et al., 2014; Sun et al., 2016b; Le et al., 2015a; Lidbury et al., 2015a; Donaldson et al., 2017], test-case reduction [Regehr et al., 2012; Pflanzer et al., 2016], and test-case prioritization [Chen et al., 2017, 2016]. Although the initial focus was on C/C++ compilers, researchers have also invested much effort on testing other compilers [Lidbury et al., 2015a; Dewey et al., 2015; Donaldson et al., 2017], runtime systems [Chen et al., 2016, 2019], and even dynamic programming languages [Wang et al., 2019; Holler et al., 2012; Park et al., 2020]. The result from this research is far beyond prominent: (1) discovery and fixing of thousands of (critical) bugs in industrial-strength compilers, such as GCC and LLVM, (2) enhancements on the compiler testing pipelines [LLVM Project, 2021], and (3) prevention of important compiler crashes and miscompilations (i.e., generation of incorrect machine instructions).

State-of-the-art research endeavors primarily focus on finding bugs in optimizing compilers. Indeed, optimizations is a source of problems that justifiably keeps researchers preoccupied with verifying and testing the implementation of optimizations [Lopes et al., 2021; Lee et al., 2018; Lopes et al., 2015; Leroy, 2006; Le et al., 2014; Yang et al., 2011; Zhang et al., 2017]. However, optimization issues is not the only challenge when working with compilers: it is surprising that this growing body of work currently neglects other compiler components, most notably the front-end. The compiler front-end is responsible for performing (1) the source code's lexical analysis and parsing, and (2) a set of semantic analyses that verifies whether the input code is error-free and respects the semantics of the language. Unfortunately, the ongoing language evolution and the difficulty of harmonizing new language features with type systems [Mackay et al., 2020; Grigore, 2017] render the implementation of the corresponding typing algorithms notoriously challenging. Indeed, in statically-typed languages with (1) rich and expressive type systems that rely on complex type theories (e.g., higher-kinded types [Moors et al., 2008], parametric polymorphism, or path-dependent types [Amin et al., 2016]), and (2) modern features (e.g., type inference, mix of object-oriented with functional programming), the implementation of front-ends (and especially the task of typing programs) has become particularly complex exhibiting a high-density of bugs. For example, at the time of writing, the type checker of the Scala 2 compiler (*typer*) is the component that suffers from the most bugs (see scala/bug).

Typing bugs degrade the reliability of the compiled programs and developers' productivity. Specifically, such bugs can (1) lead to the rejection of well-formed programs making developers waste time on debugging their correct programs, (2) violate the safety provided by type

systems [Milner, 1978] and can potentially cause security issues at runtime, or (3) invalidate subsequent compiler phases, such as optimizations.

Despite their importance, compiler typing bugs are little-known, and the community has not given much emphasis to understanding and testing static typing procedures. For this reason, before designing and proposing an approach for finding bugs in compilers' typing algorithms, this thesis takes a step back in order to get a better understanding of the nature and characteristics of compiler typing bugs. To do so, we conduct the *first* quantitative and qualitative study of the characteristics of compiler typing bugs based on a proposed *bug collection and analysis framework* [Chaliasos, Sotiropoulos, et al., 2021]. Specifically, through this bug analysis, we aim to understand their manifestations, their nature, and obtain insights into how these bugs are introduced, triggered, and fixed.

Based on the findings and observations of our bug study, we design an approach for detecting compiler typing bugs [Chaliasos, Sotiropoulos, et al., 2022]. The core component of our approach is a program generator designed to produce *well-formed* programs written in an *abstract representation (IR)*, an object-oriented language supporting parametric polymorphism, functional programming constructs, and type inference. One of the design goals of our approach is its ability to test multiple compilers. For this reason, we use our IR to *abstract away* differences of target languages. Our generator takes as input a configuration that can either disable certain features (e.g. bounded polymorphism), enable them, or affect their probability distribution. Language-aware translators then convert a program written in the IR into a corresponding source file, which is ultimately passed as an input to the compiler under test.

We have also designed two transformation-based approaches namely: a *type erasure mutation* and a *type overwriting mutation* for detecting type inference and soundness bugs respectively. The type erasure mutation is a semantics-preserving transformation that removes type-related information from an input program. The type overwriting mutation replaces a type t with another type t' in a way that this replacement invalidates the program's correctness. Hence, unlike a program produced by our generator or through the type erasure mutation, we expect the compiler to reject the output of type overwriting mutation. Both mutations are empowered by *type graph*, an abstraction we define upon our IR for modeling program type information. Using type graph and its properties, we model the problems of removing program type information, and injecting type errors in an input program, as graph reachability problems.

Our tool implementation called HEPHAESTUSis currently able to test compilers for three different languages: Java (`javac`), Kotlin (`kotlinc`), and Groovy (`groovyc`). All selected languages are statically-typed, object-oriented languages, feature advanced type systems, and support parametric polymorphism via the Java generics framework [Bracha et al., 1998]. Java is consistently on the list of the top five most popular languages [Github Inc., 2021; TIOBE Software BV, 2021]. Kotlin has become the de-facto language for Android development [Mateus and Martinez, 2020]: already over 80% of the top-1000 Android applications use Kotlin [Google developers, 2021]. Finally, Groovy is a popular hybrid language that supports both dynamic and static typing.

Over a period of nine months, we have found 170 bugs in all the examined compilers, of which 104 bugs were subsequently fixed by developers. Thanks to type erasure and type over-

writing mutations, we have uncovered 56 inference and 23 soundness bugs, which we were unable to detect by using our program generator by itself. Our results further indicate that our mutations are able to increase coverage of compiler code. For example, type erasure mutation covers up to 5,431 more branches and invokes up to 217 more functions, when compared to our generator.

1.2 Finding Bugs in Data-Centric Software

Data-intensive applications are becoming more and more pervasive in today's world. Almost every modern application is now backed by a data processing or data management system, such as relational and non-relational database engines, systems for managing data pipelines, object-relational mapping systems, stream processing libraries and frameworks, and many more. This demand for data-centric software is aligned with a tremendous supply: we witness an ample number of data-oriented solutions and systems. For example, at the time of writing, there are almost 400 different database engines (including both relational and non-relational databases) according to DB-Engines [2022].

It is more than evident that the reliability of data processing and data management systems is of paramount importance. A subtle implementation bug in a data-centric system can affect millions of applications and users that rely on the buggy system. In this thesis, we focus on examining the reliability of object-relational mapping systems, a kind of software that is in a widespread use in modern web and cloud applications. *Object-Relational Mapping (ORM)* is an established programming technique [Torres et al., 2017; Chen et al., 2014; Roebuck, 2012] that has emerged as a solution to the *Object-Relational Impedance Mismatch* problem [Bauer et al., 2015; Maier, 1990]. ORM fills the gap between relational databases and object-oriented programs by providing an object-oriented interface on top of relational databases that gives rise to an important property: *persistence*. Through this interface, the objects of a program can be easily saved and retrieved from the secondary storage without requiring boilerplate code for mapping application data to database records. ORM not only boosts developer productivity and reduces maintenance costs [Bauer et al., 2015; Eltsufin, 2019], but also promotes portability because it abstracts away differences of Database Management Systems (DBMS) [Bauer et al., 2015; Eltsufin, 2019].

Currently, there is a plethora of ORM implementations: through a simple Github search, one runs into more than 50 ORM frameworks, written for almost every language. Indicative examples include Django and SQLAlchemy for Python, Hibernate for Java, ActiveRecord for Ruby, and Sequelize for JavaScript. These systems are used by millions of applications [Chen et al., 2016] and are adopted by many popular organizations, such as Dropbox, Gitlab, and OpenStack [Bayer, 2020; Django Software Foundation, 2020c; Yang et al., 2018].

Despite their wide industrial adoption, the automated testing of ORM systems is an overlooked problem. Current testing efforts mainly use manually-written test suites, which, as we demonstrate, are often insufficient for ensuring the correctness of ORM implementations. Yet, ORM implementations are complex [Bauer et al., 2015] (typically consist of thousands lines of code) and, unfortunately, involve a high density of bugs. For example, the ORM implementa-

tion in the Django web framework is the component that suffers from the most bugs [Django Software Foundation, 2020a]: 23% of the reported bugs in Django are related to the ORM component, and they are significantly more than the reported bugs associated with the secondly affected component (8%). These ORM bugs may have potential reliability ramifications on the applications that rely on them. For example, ORM bugs lead to incorrect interactions with the underlying database that cause frustrating crashes [Viswa, 2020], wrong store and retrieval of data [Bank, 2016], and even security vulnerabilities [sql, 2019; dja, 2020].

To detect bugs in ORM implementations, we propose a differential testing approach [Sotiropoulos et al., 2021]. At a high-level, our approach exercises ORM systems by constructing equivalent queries written in the target ORM implementations, and then compares query results for mismatches. We begin by generating a random database schema used to set up databases across multiple DBMSs. We test the functionality of ORMs by querying the databases using each ORM's API. However, since ORM systems do not share a common input format, we design an *abstract* query language which is close to ORM APIs. This allows us to build expressive queries that exercise diverse functionality combinations across ORM implementations. Finally, we use ORM-specific *translators* to convert abstract queries into concrete ones, which are executable in the corresponding ORM implementations.

Our differential testing approach is *data-oriented*: beyond queries, it is the data inserted to the underlying databases that affect the effectiveness of the testing efforts. We employ a solver-based approach for generating *targeted* database records with respect to the constraints of the generated abstract queries. This improves the effectiveness of differential testing because it minimizes the number of queries where ORMs return empty results. Our approach goes beyond the existing body of work in compiler and programming language testing [Yang et al., 2011; Chen et al., 2016, 2019; Lidbury et al., 2015b; Sun et al., 2016a], and addresses several challenges specific to ORM systems, such as lack of a common input, data generation, database schema generation, or DBMS setup.

We use our implementation called CYNTHIA to test five popular ORM systems on four widely-used database engines, and find 28 unique bugs. The vast majority of these bugs are confirmed (25 / 28), more than half were fixed (20 / 28), and three were marked as release blockers by the corresponding developers.

1.3 Finding Dependency Bugs in File System Resources

In addition to processing data that reside in the memory, many modern applications and systems deal with intensive operations on file system resources, such as files and directories. These operations are done as part of various, complicated, but routine tasks: from logging and storing (intermediate) outputs into a file system, to compiling or building software and configuring entire computer systems (e.g., installing packages and libraries into a machine).

Reading and writing from/to a disk are expensive operations. Therefore, to increase throughput, applications and systems typically treat file system-intensive tasks as *partially ordered*. This means that applications are free to schedule these tasks in any order to support parallel execution. However, partially-ordered tasks come with some caveats, especially when

dealing with dependent tasks, i.e., tasks that touch the same file system resources. These dependency relations impose implicit ordering constraints that should be satisfied to perform a smooth and reliable execution.

Unfortunately, the dependencies between tasks that process common file system resources are not always known to the developers. This degrades the reliability of applications, as missing dependency information can lead to runtime failures (e.g., when trying to read a file that is not present in the file system), and even worse, other, less noticeable, incorrect output results. For example, missing dependency information can result in a file system that is not in the desired state (e.g., when writing incorrect contents to a file due to erroneous ordering of operations).

In this thesis, we study and prevent *missing dependency information* related to file system resources, in the context of two popular and everyday development tasks: *automated system configuration* and *automated builds* [Sotiropoulos et al., 2020b,a]. We propose an *effective* and *efficient dynamic* method for detecting dependency bugs in configuration management and build scripts. Our method is based on a model (FSMove) that treats an execution stemming from an *arbitrary* configuration management and build system as a sequence of tasks, where each task receives a set of input files, performs a number of file system operations, and finally produces a number of output files. FSMove takes into account (1) the specification (as declared in build and configuration management scripts) and (2) the definition (as observed during an execution through file accesses) of each task. By combining the two elements, we formally define four different types of dependency bugs associated with the handling of file system resources that arise when a file access violates the specification of the given configuration management and build script. Our testing approach operates as follows. First, it monitors the execution of a build or configuration management script, and models this execution in FSMove. Our method then verifies the correctness of the execution by ensuring that there is no file access that leads to any dependency bug in file system resources. Note that to uncover bugs, our method only requires monitoring a single execution.

We demonstrate the applicability of our approach on scripts written in popular build automation and configuration management systems, namely, Make, Gradle and Puppet. Make is one of the most well-established build tools [McIntosh et al., 2015], Gradle is a modern Java Virtual Machine (JVM)-based system that has become the de-facto build tool for Android and Kotlin programs [Karanpuria and Roy, 2018; Pelgrims, 2015; Derr et al., 2017], while Puppet [Loope, 2011] is one of the most popular system configuration tools used to manage infrastructures [Shambaugh et al., 2016; Rahman et al., 2019].

We evaluate the effectiveness of FSMove by detecting issues in 357 out of the 966 examined Make, Gradle, and Puppet projects. Notably, 300 issues found in 72 open-source projects were confirmed and fixed by upstream developers. This indicates that our tool can identify issues that matter to the community. Furthermore, our approach is more effective, and 74× faster than the state-of-the-art, on average, when analyzing Make projects.

1.4 Contributions

This thesis makes the following contributions:

- We introduce a method for collecting and assessing previously-reported bugs. Our method is realized by examining 320 compiler typing bugs taken from popular JVM compilers. Through our in-depth analysis on diverse aspects, including bug symptoms, root causes, bug fixes, and test case characteristics, we enumerate the implications of our findings, and discuss potential future directions. Finally, we provide a corresponding reference dataset consisting of the examined bugs (Section 2.2).

- We introduce a framework for finding compiler typing bugs. This framework contains a program generator carefully designed to find typing bugs in compilers of different languages by generating inputs written in an abstract representation. On top of our program generator, we design two novel transformation-based testing techniques, namely *type erasure mutation* and *type overwriting mutation* used for finding type inference and soundness issues. Both methods rely on a model and an intra-procedural type inference analysis for capturing type information flow (Section 3.1).

- We introduce an automatic, data-oriented differential testing approach for finding bugs in data-centric software. This approach is based on an abstract language for synthesizing queries and their associated data. The generated queries are guaranteed to fetch non-empty results from the storage by modeling data generation as a constraint problem. The query results are then compared against a test oracle for identifying functional errors (Section 3.2).

- We introduce FSMOVE, a model for specifying and verifying arbitrary executions against the presence of dependency bugs associated with file system resources. Based on FSMOVE, we design a dynamic method that is able to uncover four different types of dependency issues that lead to mishandling of file system resources, such as files and directories (Section 3.3).

- We develop and provide three open-source, bug-finding tools namely, HEPHAESTUS, CYNTHIA, and FSMOVE, each of them implements the testing techniques presented in this thesis. HEPHAESTUS is the first framework for validating the implementation of various type checkers: `javac`, `kotlinc` and `groovyc`. CYNTHIA is the realization of the data-oriented differential testing and is used to test many popular ORM system implementations. The FSMOVE tool currently finds dependency bugs in build and configuration management scripts written in Make, Gradle, and Puppet (Chapter 4).

- We present an thorough evaluation of our bug-finding tools in terms of many aspects, including bug-finding capability, code coverage improvement, runtime performance. Overall, our tools led to the *disclosure and fix* of more than *400 bugs* found in critical and widely-used software. We also provide a qualitative analysis of the discovered bugs and their characteristics. This analysis is helpful for the developers and practitioners to identify which are the main pitfalls when developing their software, and where to focus their development efforts (Chapter 5).

Availability: All the bug-finding tools of this thesis are available as open-source software under the GNU General Public License v3.0.

- HEPHAESTUS is available at https://github.com/hephaestus-compiler-project/hephaestus.
- CYNTHIA is available at https://github.com/theosotr/cynthia.
- FSMOVE is available at https://github.com/AUEB-BALab/fsmove.
- The dataset that comes from the study of typing bugs in JVM compilers is available at https://zenodo.org/record/5411667.

1.5 Thesis Outline

Figure 1.1 shows a map of this thesis and how its components relate to each other. The rest of the thesis is organized as follows:

- **Chapter 2 (Background):** This chapter introduces the key elements of the fundamental automated bug detection techniques that have been used in the literature. It also discusses the basic concepts behind the three classes of bugs addressed in this thesis, i.e., their importance, motivating examples, and the main challenges associated with their discovery. This chapter also presents a framework for collecting and analyzing previously-reported bugs. This framework was used for getting a better understanding about the nature and characteristics of typing bugs found in four popular JVM languages (Section 2.2). Section 2.2 studies many facets of typing bugs (e.g., symptoms, root causes), and enumerates a set of potential implications for compiler testing.
- **Chapter 3 (Methods):** This chapter introduces three methods for finding (1) compiler typing bugs, (2) bugs in data-centric software, and (3) dependency bugs in file system resources. All these methods are underpinned by abstractions whose purpose is to facilitate reasoning for bug detection, and promote generalizability.
- **Chapter 4 (Implementation):** This chapter discusses the most important technical details behind the implementation of the three bug-finding tools developed in this thesis, i.e., HEPHAESTUS, CYNTHIA, and FSMOVE.
- **Chapter 5 (Evaluation):** This chapter evaluates HEPHAESTUS, CYNTHIA, and FSMOVE, in terms of many facets, such as bug-finding capability, code coverage improvement, characteristics and importance of the discovered bugs, runtime performance.
- **Chapter 6 (Related Work):** This chapter covers related work and explains how our techniques differ from prior work.
- **Chapter 7 (Conclusion and Future Work):** This chapter concludes this thesis by discussing the key takeaway, the implications on the software industry, and the future work that stem from our work.

1.6 Publications

The contents of this thesis are based on the following publications:

- *Thodoris Sotiropoulos*, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical Fault Detection in Puppet Programs. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, https://doi.org/10.1145/3377811.3380384.

- *Thodoris Sotiropoulos*, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. A Model for Detecting Faults in Build Specifications. Proc. ACM Program. Lang. 4, OOPSLA, Article 144 (nov 2020), 30 pages. https://doi.org/10.1145/3428212

- *Thodoris Sotiropoulos*, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In Proceedings of the 43rd International Conference on Software Engineering (Madrid, Spain) (ICSE '21). IEEE Press, 1535–1547. https://doi.org/10.1109/ICSE43902.2021.00137. *Distinguished Artifact Award.*

- Stefanos Chaliasos*, *Thodoris Sotiropoulos**, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. Proc. ACM Program. Lang. 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

- Stefanos Chaliasos*, *Thodoris Sotiropoulos**, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Compiler Typing Bugs. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design Implementation (San Diego, CA, USA) (PLDI 2022). Association for Computing Machinery. https://doi.org/10.1145/3519939.3523427. *Distinguished Paper Award, Best Artifact Award.*

*These authors contributed equally.

Chapter 2

Background

In this chapter, we describe and classify the fundamental automated bug detection methods that have been used in the literature (Section 2.1). Then, we discuss every class of bugs that this thesis aims to discover, that is, compiler typing bugs (Section 2.2), bugs in data-centric software (Section 2.3), and dependency bugs in file system resources (Section 2.4). Every bug category is described as follows. We first give a brief introduction of the classes of software where every bug category occurs. Next, we present some motivating bug examples, and discuss the key technical challenges associated with bug discovery. This chapter concludes by highlighting the need for abstractions for more effective software testing (Section 2.5).

2.1 Automated Bug Detection

In this section, we focus on providing a background on the basics of *automated bug detection*. We begin with presenting how program analysis is used for bug identification (Section 2.1.1), and continue with a high-level overview of automated software testing (Section 2.1.2). Finally, we end up with the fundamentals behind randomized testing (Section 2.1.3), a form of software testing that occupies significant room in this thesis.

2.1.1 Program Analysis

Program analysis is the process of reasoning about the behavior of a program, and computing the presence or absence of some program properties, such as correctness. Program analysis has been traditionally used for bug-finding purposes, and is split into two main categories: *static analysis* and *dynamic analysis*. Note that this thesis mainly focuses on discovering *functional errors* using dynamic analysis and automated testing techniques.

2.1.1.1 Static Analysis

Static analysis computes summaries about program properties without actually running the program. Examples of such properties include the list of values or addresses that a particular variable might hold or point to [Milanova et al., 2002], the list of methods invoked by a

certain caller method [Salis et al., 2021], and more. A static bug-finding tool can then leverage these summaries to detect various kinds of errors. For example, knowing the program locations where a certain variable is read and written, a static analyzer can identify uses of uninitialized variables. Numerous bug-finding tools relying on static analysis have been developed for detecting a plethora of issues, including but not limited to memory safety issues [Calcagno and Distefano, 2011; Sui et al., 2012, 2014], concurrency bugs [Naik et al., 2006; Gabet and Yoshida, 2020], security vulnerabilities [Livshits and Lam, 2005; Wassermann and Su, 2008], type errors in dynamic languages [Jensen et al., 2009; Bae et al., 2014; Sotiropoulos and Livshits, 2019]. Nevertheless, static analysis is unable to determine functional correctness [Emanuelsson and Nilsson, 2008], i.e., whether a particular procedure produces the expected result.

One of the first applications of static analysis is the domain of compiler optimizations. In particular, static analysis underpins many traditional program transformations implemented in modern compilers [Cooper and Torczon, 2012, Chapter 9], such as constant folding, dead code elimination.

Static analysis typically *over-approximates* the behavior of a program by introducing abstractions that capture all possible program behaviors. For example, consider a static analysis that computes the sign of integer variables at every program location, i.e., whether a variable holds a positive or a negative number. Such a static analysis does not work on the unbounded domain of integers, but on an *abstract* domain where an integer can hold three possible values: *negative*, *zero*, *positive*. Values are typically abstracted through lattices [Kam and Ullman, 1977].

Over-approximation introduces false positives, as static analysis captures behaviors and properties that are not actually present in the original program. A large number of false positives hinders the adoption of static analysis tools [Johnson et al., 2013]. To reduce false positives, static analysis often comes with various kinds of *sensitivity*. For example, *context-sensitivity* differentiates method invocations based on some context, e.g., the parameters (parameter-sensitivity), or the receiver (object-sensitivity) [Milanova et al., 2002]. *Flow-sensitivity* takes into account the control-flow of the program, and thus respecting the order in which operations take place. Unfortunately, improving analysis precision introduces some overhead that can limit analysis scalability. Therefore, it is important to strike a balance between precision and performance, and use the right abstractions and forms of sensitivity [Smaragdakis et al., 2011].

2.1.1.2 Dynamic Analysis

Dynamic analysis computes properties and detects program errors, while executing a program. This is achieved by a sort of *instrumentation* on the input program. The purpose of the instrumentation is to collect facts about program execution, such as memory accesses, code coverage information, caller-callee relationships, by inserting special instructions to dedicated program locations. Instrumentation inevitably slows down execution, but the amount of overhead depends on the implementation and purpose of instrumentation. There are two types of instrumentation: *static instrumentation* and *dynamic instrumentation*.

Static instrumentation: Static instrumentation instruments the code before the program runs, e.g., during the compilation process. An example of a dynamic analysis tool that performs

static instrumentation is compiler sanitizers, such as the address sanitizer [Serebryany et al., 2012] for C/C++ programs. For example, the address sanitizer maps a memory location l to a region that resides in a *shadow memory*. Locations in the shadow memory indicate whether an access to the original location l is safe or not. The address sanitizer instruments every memory access, resolves it to a shadow address, and performs a check about its validity. There are different types of compiler sanitizers that are capable of identifying different types of errors, including memory safety issues (buffer overflows, memory leaks), undefined behaviors (arithmetic overflows), or concurrency issues (data races). Compiler sanitizers are now part in many modern compilers, e.g., GCC, LLVM, and the compiler of Golang.

An advantage of static instrumentation is that it is typically lightweight and imposes little overhead on program execution. Its disadvantage is that it requires (1) the program's source code, (2) re-compiling the program to insert the instrumentation. Another disadvantage of static instrumentation is that the instrumented code can be incompatible with any dynamic libraries (i.e., uninstrumented code) linked with the original program.

Dynamic instrumentation: Dynamic instrumentation augments program behavior, as the program under test is running. A clear advantage of this over static instrumentation is that the instrumentation does not require re-compilation. Therefore, dynamic instrumentation can be also applied to proprietary codebases. Two popular and widely-used dynamic instrumentation frameworks are Valgrind [Nethercote and Seward, 2007] and DynamoRIO [Bruening et al., 2003]. For example, Valgrind translates machine instructions into an intermediate language (IR), and accepts dynamic analysis tools to modify these instructions or add new ones. Then, Valgrind compiles the instrumented IR code back to machine code, which is in turn executed by Valgrind. Many dynamic bug-finding tools have relied on Valgrind and DynamoRIO for (1) uncovering buffer overflows and memory leaks, (2) profiling, or (3) building dynamic call graphs.

A drawback of dynamic instrumentation is slow execution times, especially when compared to static instrumentation. The slowdown can reach $10\times$–$20\times$ for certain benchmarks [Serebryany et al., 2012].

The bug-finding capability of dynamic analysis tools heavily depends on the quality of program inputs. This means that if the quality of a program input is poor and does not exercise a deep program execution path, a dynamic analysis tool is unable to produce any bug report. Dynamic bug-finding tools significantly benefit from methods for automated test input generation [Andreasen et al., 2017], such as those employed in randomized testing (see Section 2.1.3).

2.1.2 Automated Software Testing

Automated software testing is the de-facto method for uncovering bugs in (complex) software systems. It comes with several forms that involve different granularities and goals, such as unit testing, regression testing, integration testing, randomized testing, model-based testing [Visser et al., 2016b; El-Far and Whittaker, 2002; Tretmans, 2008; LLVM Project, 2021]. At a high-level, though, all software testing methods share the same workflow, which is shown in Figure 2.1. Here, we have a *system under test (SUT)* that we test by passing a set of existing or automatically

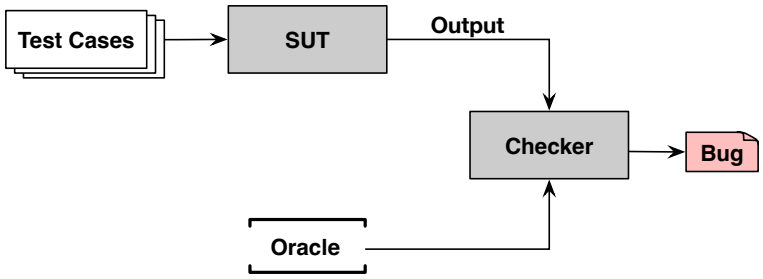


Figure 2.1: Workflow of traditional software testing.

generated inputs, known as *test cases*. Based on the given test cases, the behavior of SUT is checked by comparing its outputs against an expected behavior described by a *test oracle*.

The effectiveness of automated of software testing relies on two important aspects: *existence of diverse test cases*, and *test oracle specification*. When testing a program with a specific test case, we typically reason about a single execution path. Ideally, to verify correctness and absence of bugs, we need to provide SUT with a corpus of test cases that covers all its possible executions. Unfortunately, the vast majority of software comes with an enormous input domain that makes exhaustive enumeration of all possible inputs impractical. Thankfully, there are many automated input generation techniques whose purpose is to prune search space and yield test inputs that are expected to uncover deep bugs and trigger interesting program behaviors [Mao et al., 2016; Zhang et al., 2017; Aschermann et al., 2019; Padhye et al., 2019; Lemieux et al., 2018].

However, even in the presence of a highly effective suite of test cases, software testing becomes unrealistic when there is no test oracle. Indeed, defining a test oracle for determining whether a certain program behavior is buggy is not always possible. This inability of specifying an oracle for testing software is known as the *test oracle problem* [Weyuker, 1982]. The test oracle problem emerges for plenty of reasons. Some indicative examples include: the system under test (1) produces output that is too complex to be verified for correctness, (2) lacks of clear semantics, (3) is not deterministic, or (4) involves floating-point computations, and thus, output contains some noise due to round-off errors. To better illustrate the test oracle problem, imagine that we want to test a compiler. The output of the compiler is a program written in a low-level language. Based on the given source program, there is an ample number of low-level programs that are equivalent of the given source. The question that arises at this point is: how can we verify that the actual output of the compiler matches the semantics of the original source program? This questions does not have an apparent answer, and as we will see later, compiler testing addresses this issue through the introduction of a *pseudo-oracle* [Weyuker, 1982].

In Section 2.1.3, we provide more details about a form of automated testing that this thesis

heavily relies on, that is, randomized testing.

2.1.3 Randomized Testing

Randomized testing (or fuzzing) is a sort of an automated testing method, where a SUT is tested by generating a large set of test inputs through a randomized procedure. The goal of randomized testing is to produce diverse test cases so that different program behaviors are exercised, and thus uncovering bugs. Depending on the underlying strategy, this test input generation is completely random, or more guided (see Section 2.1.3.2). Notably, when a test oracle is not present, randomized testing seeks for program crashes or hangs.

Due to its simplicity (at least at conceptual level) and scalability, randomized testing has been employed with a tremendous success in discovering bugs and security vulnerabilities in software from a plethora of domains, including but not limited to: compilers and programming language systems [Yang et al., 2011; Le et al., 2014; Chen et al., 2016, 2019; Donaldson et al., 2017; Park et al., 2020], debuggers [Lehmann and Pradel, 2018], database management systems [Rigger and Su, 2020c,a,b], network protocol implementations [Fiterau-Brostean et al., 2020; Chen and Su, 2015a; Petsios et al., 2017], and cloud backend services [Godefroid et al., 2020; Atlidakis et al., 2019, 2020b,a]. Interestingly, the first ever successful application of randomized testing was back in 1990 [Miller et al., 1990], where a randomized testing tool (fuzzer) was used to assess the reliability of various Unix utilities (such as `cat`, `grep`, `head`) by producing random sequences of characters. By applying their fuzzer to the Unix utilities coming from seven different versions of Unix, Miller et al. [1990] were able to crash 24%–33% of Unix programs on each platform.

2.1.3.1 Constructing Test Inputs

There are two main approaches for constructing test inputs in randomized testing: generating test inputs from scratch (*random program generation*), and generating test inputs from other existing test cases (*mutation-based fuzzing*).

Through random program generation, the fuzzing procedure is responsible for producing test cases completely from scratch. At a naive level, this can be achieved by randomly generating a sequence of bytes / characters. However, such a simple generation approach can rarely produce meaningful test cases when it comes to software that expects structured inputs. To tackle this limitation, more advanced program generation methods consult a grammar (*grammar-based fuzzing*) to provide SUT with syntactically-valid test cases [Maurer, 1990; Coppit and Lian, 2005; Godefroid et al., 2008; Yan et al., 2013; Atlidakis et al., 2019]. However, even grammar-based techniques often struggle to generate inputs that exercise deeper components of SUT, such as semantic procedures. To this end, there are program generators that exploit the semantics of SUT to produce inputs that are *both* syntactically and semantically valid. For instance, as an effort to uncover bugs in the middle- and back-end of C compilers, CSmith [Yang et al., 2011], a popular random generator for programs written in C, constructs test cases that respect the language standard, and are free from undefined behavior. Similarly, other program generators rely on *constraint logic programming* to synthesize well-typed programs that satisfy

some input constraints written in a declarative language (e.g., Prolog). These constraints are essentially used to capture the semantics (e.g., typing rules) of a language under test, e.g., Rust, JavaScript [Dewey et al., 2014, 2015].

Recent advances focus on automatically synthesizing input grammars [Bastani et al., 2017; Wu et al., 2019; Kulkarni et al., 2021]. The idea is that these automatically inferred grammars can be used for fuzz testing. The benefit of such approaches is that they remove the burden of manually deriving a grammar from (often poorly-written) documentations [Bastani et al., 2017]. These learning-based approaches strive to provide a good balance between precision and recall, particularly when dealing with programs that receive inputs with complex formats [Kulkarni et al., 2021].

Unlike random program generation, mutation-based fuzzing produces test cases by randomly modifying some existing inputs, known as *seeds*. These random modifications involve replacements, additions, or deletions of bytes, tokens, or whole statements. As with grammar-based methods, the goal of mutation-based fuzzing is the construction of grammatically valid inputs by making small changes to existing, valid seeds. This helps exercise new behaviors of SUT, while preserving much of the structure and characteristics of the given seed inputs. As we will see next, to make targeted modifications to the given seeds so that the new test inputs are more promising to identify bugs, mutation-based fuzzing is often guided by feedback taken from the execution of SUT. This feedback can be: code coverage information, number of newly-triggered execution paths, number of bugs found, and more. Some popular mutation-based fuzzers are American Fuzzy Lop (AFL) [M. Zalewski, 2013], libFuzzer [LLVM Project, 2022], and Honggfuzz [Google, 2022].

2.1.3.2 Feedback-Directed Randomized Testing

A fuzzer can potentially observe the execution of a SUT to see whether a generated input triggers interesting behaviors, e.g., a previously unnoticed output, execution of previously-uncovered code. The observations made by the fuzzer are then used for guiding the creation of new inputs that exercise similar, interesting behaviors. Depending on whether and how a fuzzer exploits this feedback from a SUT, we have split fuzzing into three categories: *black-box fuzzing*, *grey-box fuzzing*, and *white-box fuzzing*.

Black-box fuzzing: In black-box fuzzing, the fuzzer treats the system under test as a "black-box" that simply takes an input and produces an output, without knowing anything else about its internals. Black-box fuzzing is highly flexible and generally-applicable, as it does not require the source code of SUT, or any instrumentation and modification performed on SUT. This lack of instrumentation makes black-box fuzzing very efficient, as SUT runs unmodified at full speed.

Unfortunately, the shortcoming of black-box testing is that is does not leverage any information stemming from the execution of SUT, and thus much time can potentially be wasted in generating test inputs that explore the same paths with previously generated test cases. Some recent techniques overcome this limitation by generating diverse inputs via machine learning [Reddy et al., 2020], or the introduction of constraints that lead to specialized input grammars [Gopinath et al., 2021].

Grey-box fuzzing: To better observe how SUT behaves on some inputs, and generate the next test inputs based on this feedback, grey-box fuzzing works on a instrumented SUT. The instrumentation can be either static or dynamic, and its purpose is to provide fuzzing with information about the internal behavior of SUT. Examples of information taken from the instrumented SUT include information about code coverage [M. Zalewski, 2013; Padhye et al., 2019], newly-explored paths, hot spots [Lemieux et al., 2018], and more. This runtime information then guides the generation and mutation process so that next inputs trigger noteworthy SUT behaviors. The selection of new test cases is often achieved through the optimization of a fitness function using advanced (evolutionary) algorithms [Wüstholz and Christakis, 2020; M. Zalewski, 2013; LLVM Project, 2022; Google, 2022] or other techniques coming from statistics [Chen et al., 2016, 2019], such as Markov Chain Monte Carlo methods [Chib and Greenberg, 1995].

The instrumentation performed on SUT imposes an overhead on grey-box fuzzing that makes it less efficient than black-box fuzzing. Also, when applying static instrumentation, grey-box fuzzing requires the source code of SUT.

White-box fuzzing: White-box fuzz testing, a variant of symbolic execution, comes to address a fundamental limitation of black-box and grey-box fuzzing methods: they struggle to uncover deep bugs that depend on specific input conditions [Bounimova et al., 2013]. To illustrate this, consider the following code snippet:

```
1  if (x == 1004) {
2    // buggy code
3  }
```

In the above example, to execute the buggy code inside the true branch of the conditional, the input variable `x` should be equal to 1004. The probability of hitting this bug by randomly generating 32-bit integers is pretty small: $1/2^{32}$.

To tackle this, white-box fuzzing initially monitors the execution of SUT based on a specific input and gathers all path constraints that stem from branch conditions. To explore new program paths and branches, white-box fuzzing negates the collected path constraints and employs a theorem prover to produce new test cases that satisfy the given constraints and systematically trigger new paths.

In the above example, assuming that we initially run the program on a input `x = 0`, a white-box fuzzer presumes that the path constraint is `x != 1004`, as the execution proceeds with the false branch of the `if` condition (`0 != 1004`). The next move of white-box fuzzer is to generate an input that satisfies the negated path constraint (i.e., `x == 1004`). This makes fuzzer run SUT on such an input (i.e., `x = 1004`), and thus exploring the true branch of the condition, which hits the buggy code.

Despite its effectiveness in finding deep bugs and systematically exploring SUT's execution paths, white-box fuzzing experiences similar scalability issues as symbolic execution [Bounimova et al., 2013]. This is because of the *path explosion problem* and the huge amount of time spent on solving path constraints [Cadar and Sen, 2013].

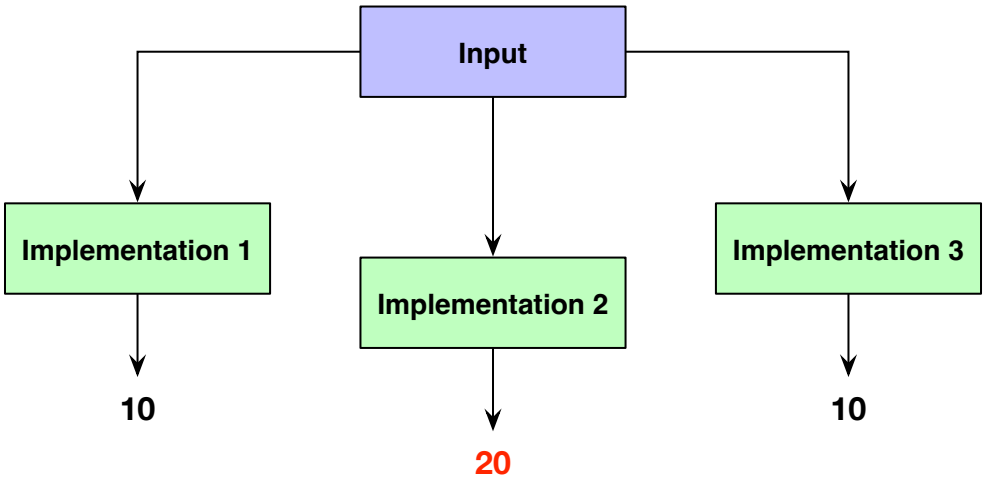


Figure 2.2: The concept of differential testing. A mismatch in the outputs of implementations indicates a bug in at least one of them.

2.1.3.3 Addressing the Test Oracle Problem

All the randomized testing techniques described above assume that there is a test oracle for determining whether the execution of a SUT on a specific test input is buggy. As already mentioned, specifying a test oracle is not always possible due to the test oracle problem [Weyuker, 1982]. There are two methods, namely, *differential testing* and *metamorphic testing*, used for testing programs that lack an oracle. As we will explain shortly, these methods actually establish a *pseudo-oracle*, and not an actual test oracle.

Differential testing: The idea of differential testing [McKeeman, 1998] is demonstrated in Figure 2.2. The main requirement of differential testing is having *at least two equivalent* programs that implement the same specification. Then, we feed the implementations with the same test input I, and compare their outputs. A mismatch found in the results of the implementations under test indicates a potential bug in at least one of them (remember, these implementations are supposed to be equivalent). Note that in order to get meaningful results, differential testing requires that the implementations under test are also deterministic and free from undefined behaviors.

Differential testing is a broadly-applicable method. It has been successfully applied to testing compilers [Yang et al., 2011; Livinskii et al., 2020; Zhang et al., 2017], runtime systems [Chen et al., 2016, 2019], database management systems [Slutz, 1998], debuggers [Lehmann and Pradel, 2018], program analysis tools [Klinger et al., 2019; Kapus and Cadar, 2017], Secure Socket Layer (SSL) libraries [Chen and Su, 2015b], Representational State Transfer (REST) services [Godefroid et al., 2020], and machine learning libraries [Srisakaokul et al., 2018; Dutta et al., 2018].

The main limitation of differential testing is that in cases where there is no a reference implementation, applying differential testing is not doable. For example, it is not straightforward to test the OCaml programming language through differential testing, as the language is not standardized, and there is only a single implementation of the language. Another limitation of differential testing is that it can be used for testing only the features that are common across

the implementations under test. For example, Slutz [1998] uses a subset of the SQL language to test database management systems via differential testing. If a database bug is triggered by a feature, not covered by this SQL subset, a technique that is based on differential testing is unable to find the bug.

Differential testing produces a pseudo-oracle, because it does not precisely capture whether the behavior of SUT is buggy. For example, consider that we have two buggy implementations that result into the same incorrect output. In such a scenario, differential testing does not catch the bug, because the outputs of the two implementations match with each other.

Metamorphic testing: Metamorphic testing leverages the expectations we have about some outputs of a SUT that are derived from related inputs. For instance, consider that we want to test a function $f(x)$ that gives the square number of an input x. Now assume for the sake of the presentation that we do not know what is the precise output of $f(x)$. To test the implementation of $f(x)$ via metamorphic testing, we can produce a new input for function f, which is somehow related to the original input x. From maths, we know that such a related input is $-x$, as $f(x) = f(-x)$. Therefore, we can check $f(-x)$ to see whether it gives the same output as $f(x)$. If there is an inconsistency in the results, we report a bug in the implementation of $f(x)$. The relation of $f(x) = f(-x)$ is known as a *metamorphic* relation. Similarly, we can derive metamorphic relations for other domains. For instance, two equivalent programs written in C (e.g., two programs with reversed `if` branches) can be used for testing the correctness of C compilers via metamorphic testing.

An advantage of metamorphic testing over differential testing is that it does not require a second reference implementation. As with differential testing, metamorphic testing has seen tremendous success in its application to various complex systems that suffer from the test oracle problem. Examples of such systems are compilers [Le et al., 2014; Lidbury et al., 2015b], theorem provers [Winterer et al., 2020b,a], database management systems [Rigger and Su, 2020a,b], model checkers [Zhang et al., 2019], debuggers [Tolksdorf et al., 2019], and libraries [Lascu et al., 2022].

Metamorphic testing introduces a pseudo-oracle, because we still do not know what is the correct answer of SUT on a certain input. The only thing we know is the relation of a program output against another output.

2.2 Compiler Typing Bugs

In this section, we introduce compiler typing bugs by discussing and presenting the findings of the *first* quantitative and qualitative study of their characteristics. Specifically, in this study, we aim to understand their manifestations, their nature, and obtain insights into how these bugs are introduced, triggered, and fixed. Specifically, our study seeks answers to the following research questions.

RQ1 (Symptoms) What are the main symptoms of compiler typing bugs? What is the frequency of these symptoms?

RQ2 (Bug Causes) What are the categories into which we can group typing bugs based on their root cause? What is the frequency of these categories?

RQ3 (Bug Fixes) How are compiler typing bugs introduced? What is the size of their fixes? How long does it take to fix these bugs?

RQ4 (Test Case Characteristics) What are the main characteristics of the bug-revealing test cases? What language features are prevalent in these test cases?

To answer these questions, we examine previously-reported bugs from the compilers of four mainstream JVM programming languages, namely Java, Scala, Kotlin, and Groovy. Using carefully-crafted search criteria and some heuristics, we search the issue trackers of the studied languages and obtain 4,101 *fixed* compiler typing bugs. We analyze a random sample of 320 bugs. Specifically, we study each bug report of this sample, along with the accompanying developers' discussion, bug fix and test case. We finally evaluate many aspects of every bug, such as its symptom, its root cause, and its test case characteristics.

In the following sections, we provide an overview of the studied programming languages, along with a description of the characteristics of compiler typing bugs as derived from the findings and observations of our bug study.

2.2.1 Statically-Typed Programming Languages

We study compiler typing bugs by inspecting the reliability of four mainstream JVM programming languages, namely Java, Scala, Kotlin, and Groovy. All these languages are statically-typed object-oriented languages, their type system is nominal, while they all support parametric polymorphism by using the Java generics framework [Bracha et al., 1998]. They also support (some to a lesser or greater extend) functional programming features, including higher-order functions, function closures, or function composition. Finally, all the examined languages adopt some sort of type inference, e.g., they allow variable declarations with omitted types, instantiations of type constructors with omitted type arguments, or function declarations with omitted return types.

Java: Java is in the list of the most widely-used and popular programming languages [Github Inc., 2021; TIOBE Software BV, 2021]. Due to high competition in the JVM ecosystem, Java is constantly evolving. Starting from Java 8 where functional programming features (e.g., lambdas, higher-order functions) were added to the language, the Java team is announcing more and more releases (in a short time period) that make a lot of improvements to the language. For example, Java 9 added support for modules, Java 10 introduced the `var` keyword that allows developers to omit the type of a local variable, Java 14 enhanced the `instanceof` keyword with pattern matching support [Gavin Bierman, 2017], and more.

The Java team is also working on experimental projects in the Java Development Kit (JDK) that investigate the integration of future Java features. For example, as an addition to primitive and reference types, the project Valhalla [Brian Goetz, 2020] is working on supporting value types (e.g., objects represented as values). This feature will come with many updates in both the runtime and the type system of Java.

Scala: The Scala programming language [Odersky et al., 2004] is a research product that first appeared in 2004. Scala unifies the object-oriented paradigm with functional programming: beyond standard object-oriented features, there is a built-in support for features that are typically seen in functional languages (e.g., algebraic data types, pattern matching). One of the strengths of the language is its type system that offers some sophisticated features, such as higher-kinded types [Moors et al., 2008], path-dependent types [Amin et al., 2016], implicits, or even structural types.

As an example, consider higher-kinded types. Higher-kinded types offer an extra layer of abstraction over type constructors. Figure 2.3 shows a small Scala program that defines a higher-kinded type, i.e., a type constructor that receives a type parameter corresponding to another type constructor. On lines 1–3, the code defines a type constructor named `A` that takes two type parameters. The first type parameter is a type constructor (i.e., denoted as `X[_]`), while the second one is a regular type variable named `T`. The given type constructor `X` can be ultimately used to instantiate other types. For example on line 2, the method `m` receives a variable of type `X[T]` that comes from the application of type constructor `X` to the type argument `T`.

```
1 class A[X[_], T] {
2   def m(x: X[T]): T = ???
3 }
4 ...
5 val obj = new A[List, String]()
6 val x: String = obj.m(List("1", "2"))
```

Figure 2.3: A Scala program with a higher-kinded type.

Scala 3 and its corresponding compiler called Dotty, which initially begun as a experimental project, appeared very recently. Scala 3 involves major enhancements and additions compared to Scala 2, including contextual abstractions, union and intersection types, metaprogramming features, and other.

Kotlin: The Kotlin programming language is developed by the JetBrains team. Although it is quite a new language (first appeared in 2011), it has gained much popularity recently. It is now the Google's preferred programming language for building Android applications [Mateus and Martinez, 2020].

Kotlin provides some distinct features. For example, its type system guarantees null safety: By default, Kotlin types do not hold `null` values, and there is a special kind of types called nullable types for storing `null` values. Nullable types also come with dedicated programming constructs for performing safe property accesses on variables that may point to `null`.

Consider Figure 2.4 where we define a nullable variable whose type is `String?`. This type indicates that the variable can take any string value, or `null`. Attribute accesses of the form `x.f` are not allowed by the type system, when the type of the receiver is nullable (lines 2, 3). Instead, attribute accesses on nullable variables are done through the `?.` operator (line 4), which returns `null` if the receiver is `null`, otherwise it returns the value of the attribute being accessed (i.e., `length`).

```
1 val y: String? = null
2 // compile-time error, y is nullable
3 y.length
4 y?.length // OK, returns null
```

Figure 2.4: A Kotlin program with a nullable type.

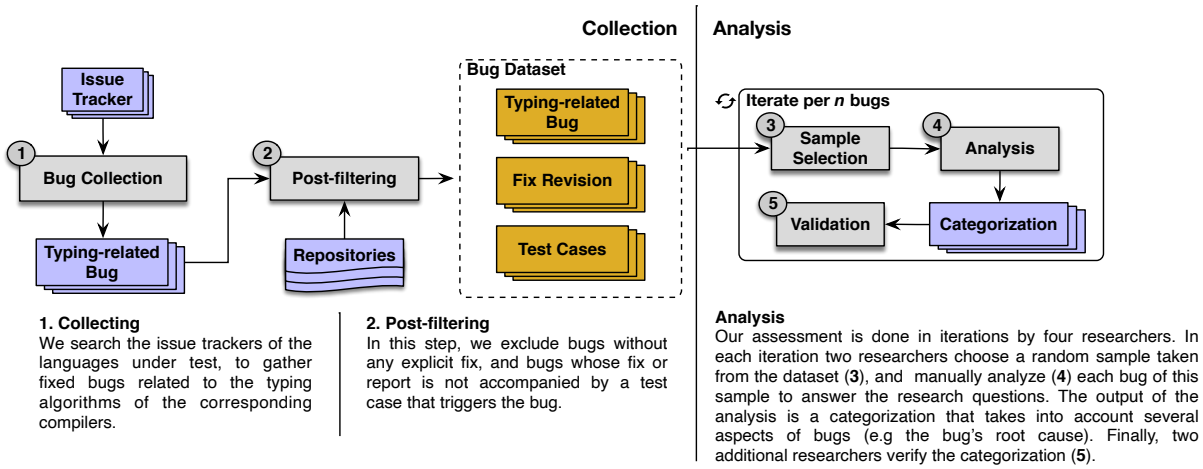


Figure 2.5: The overview of our bug collection and bug analysis approach.

Other important features in Kotlin include (1) delegation, where the implementation of an interface is delegated to a specific object, (2) extensions where developers can extend the functionality of a certain class / method without modifying the corresponding definition, (3) operator overloading, and (4) data classes. Finally, as in the case of Scala, Kotlin supports both declaration-site and use-site type variance. Note that non-nullable types, and extension methods are also part of the new Scala compiler, Dotty.

Groovy: The Groovy programming language supports both dynamic and static typing. A Groovy program can be run as a script, but there is also the Groovy compiler (i.e., `groovyc`) for compiling the code into Java bytecode. The syntax of Groovy is compatible with Java: every program written in Java should be accepted by `groovyc`. Groovy's type system is very similar to that of Java. However, an important difference is that Groovy provides flow-sensitive typing. In this context, the inferred type of a local variable changes depending on its location in the flow of the program.

2.2.2 Collecting and Analyzing Compiler Typing Bugs

Studying compiler typing bugs requires the design and development of a method for collecting and analyzing these bugs from the issue trackers of the studied programming languages. In particular, we first create a corpus of compiler typing bugs taken from the issue trackers of the selected JVM languages (Section 2.2.2.1). Then, we explain how we study and analyze the collected bugs (Section 2.2.2.2), and finally, we discuss the limitations and threats to validity of our method (Section 2.2.2.3).

Our bug collection and analysis approach is summarized in Figure 2.5. As a starting point, we take the issue trackers of the languages under study, and we apply language-specific filters in order to obtain an initial list of compiler typing bugs (*bug collection*). Then, in the next step (*post-filtering*), we filter out typing bugs that are not accompanied by an explicit fix and a test case. To do so, we search the repositories and the issue trackers of the examined languages for commits, pull requests or bug reports that are linked with any of the bugs included in the

output of the previous step. The *bug collection* and *post-filtering* steps are fully automated and constitute our approach for collecting bugs and their fixes (Section 2.2.2.1). The final outcome of this approach is a bug dataset consisting of a set of typing bugs, their fix revisions, and their test cases.

The resulting dataset is used as an input to our bug analysis approach (Section 2.2.2.2). This bug analysis is done in iterations by four researchers. Each iteration involves the examination of a specified number of bugs, n. Specifically, two researchers first choose a random sample of bugs taken from the initial dataset (*sample selection*), and then they manually analyze each bug of this sample to answer our research questions (RQ3 is partly answered through automated means — see Section 2.2.5). The output of the analysis is a categorization that takes into account several aspects of bugs, including their symptom, root cause, and test case characteristics. Finally, two additional researchers verify the proposed categorization. In case of a conflict, they discuss with the original researchers until reaching consensus (*validation*).

2.2.2.1 Collecting Bugs and Fixes

Our bug collection approach consists of two steps, namely, *bug collection* and *post-filtering*. In the first step, we search the issue trackers of the studied languages to gather *fixed* bugs related to the typing algorithms of the corresponding compilers. Our study excludes bugs related to the implementation of lexers and parsers. Similarly, bugs in the implementation of compiler optimizations or code generation are beyond the scope of this thesis. The output of the first phase includes four sets containing the URLs of the retrieved bug reports. Each set $\mathcal{B}_l$ contains bugs related to a language l.

We further filter the collected bugs by performing the *post-filtering* step. This step aims to exclude bugs without any explicit fix, and bugs whose fix or report is not accompanied by a test case that triggers the bug. To do so, we proceed as follows. First, for each language l, we get the ID of each previously collected bug $b \in \mathcal{B}_l$. Second, we search the repository of the corresponding compiler to find all the commits that refer to the given bug identifier. Third, for completeness, we use the GitHub API to retrieve pull requests that have references to the given bug. Note that it is a standard practice for the developers of the studied compilers to include the bug's numeric identifier in the description of their bug fixes.

In the *post-filtering* step, having kept the bugs for which we are able to find corresponding commits, we further examine their revisions to check whether they contain a test case. The development team of each compiler places test cases in dedicated directories, e.g., `tests/`. Therefore, to decide whether the associated commits contain a test case, we look for file updates in the aforementioned directories. When a commit does not contain a test case, we look into the corresponding bug report to discover and retrieve any linked test cases. At the end of the *post-filtering* step, we obtain four sets of bugs (where each set $\mathcal{B}'_l$ is a subset of the corresponding set $\mathcal{B}_l$ produced by the first step of our collection approach) along with the corresponding fix revisions and test cases.

While applying our bug collection method, we had to tackle the following challenge: the development teams of the languages under examination use diverse issue trackers, and adopt various categorization strategies for the reported bugs. Therefore, before collecting bugs, we

Table 2.1: Statistics on bug collection. Each table entry shows per language statistics about (1) the total number of the reported issues (Total issues), (2) the creation date of the oldest and the most recent issue considered in the study (Oldest and Most Recent), (3) the number of the selected bugs after running the *bug collection* step (BC), and (4) the number of the remaining bugs after running the *post-filtering* step (PF).

Language	Issue Tracker	REST Endpoint	Total Issues	Oldest	Most Recent	BC	PF
Java	Jira	https://bugs.openjdk.java.net/rest/api/latest/search	10,872	11 Feb 2004	26 March 2021	1,252	873
Scala 2	GitHub	https://api.github.com/repos/scala/bug	12,315	22 May 2003	11 March 2021	1,180	1,067
Scala 3	GitHub	https://api.github.com/repos/lampepfl/dotty	4,286	1 Feb 2014	28 March 2021	429	366
Kotlin	YouTrack	https://youtrack.jetbrains.com/api/issues	40,998	28 Oct 2011	9 April 2021	2,189	1,601
Groovy	Jira	https://issues.apache.org/jira/rest/api/2/search	9,710	25 Sep 2003	9 April 2021	300	246

carefully examined the corresponding issue trackers to identify all the relevant categories and filtering criteria that can be used to obtain fixed typing bugs.

Table 2.1 shows descriptive statistics of our bug collection effort. After applying the *bug collection* and *post-filtering* steps to each language, we got our final dataset, which consists of 4,153 bugs in total, of which 873 bugs are in the Java compiler (`javac`), 1,433 bugs are in the Scala compilers (either `scalac` or Dotty), 1,601 bugs are in the Kotlin compiler (`kotlinc`), and 246 bugs are in the Groovy compiler (`groovyc`). Note that although not all the examined programming languages used the same issue tracker throughout their lifetime (e.g., Scala recently migrated to GitHub from Jira [Tisue, 2017]), we were able to consider bugs even from the early days of these languages, as all these historical issues were imported to the current issue trackers. In the following, we discuss some language-specific details related to our bug collection approach.

Collecting Java bugs: Focusing on `javac` typing bugs, we inspected bugs reported in the OpenJDK project, which is the open-source implementation of the Java SE platform. The OpenJDK project employs the Jira issue tracker, which, at the time of writing, hosts 292,059 issues associated with a large number of JDK components, such as the JDK runtime, the Just in Time (JIT) compiler, Java's standard library, or other external JDK tools like the bytecode disassembler.

We used the Jira REST API to find JDK issues that meet the following selection criteria: (1) the type of the issue is "bug", (2) its status is either "resolved" or "closed", (3) its "resolution" field is set to "fixed", and (4) the issue is related to the Java compiler (i.e., the bug is assigned to the `javac` sub-component of JDK). Due to the large volume of JDK issues ($>$ 200k), we applied two more filters. First, we selected bugs that affect JDK 7 and onwards. We excluded bugs that affect early versions of JDK where crucial features of Java (e.g., generics) are not present. Second, we filtered JDK bugs based on their priority. Specifically, we selected bugs that are considered important, and their priority is "P1", "P2", or "P3". Running the *bug collection* and *post-filtering* steps yielded the final set of `javac` bugs, namely $\mathcal{B}'_j$, containing 873 JDK issues.

Collecting Scala bugs: We collected Scala bugs from two sources. The first source contains bugs reported for the Scala 2 compiler (`scalac`), while the second one includes bugs related to Dotty, the Scala 3 compiler. Both sources are using the issue tracking system of GitHub. At the time of writing, 12,315 and 4,386 issues have been reported, in total, for `scalac` and Dotty respectively.

The developers of these two compilers perform the classification of the reported issues by

assigning different labels to each issue. We constructed two queries for fetching bugs that contain labels associated with Scala's type system and typing procedures. Specifically, for `scalac` bugs, we looked for *closed* GitHub issues to which at least one of the following labels is assigned: "typer", "infer", "should compile", "should not compile", "patmat", "overloading", "dependent types", "structural types", "existential", "gadt", "valueclass", "typelevel", "compiler crash", "implicit classes", and "implicit". For Dotty, we were interested in *closed* GitHub issues that combine the "itype:bug", "itype:crash" or "itype:performance" label with at least one of the following labels: "area:typer", "area:overloading", "area:gadt", "area:implicts", "area:f-bounds", "area:pattern-matching", "area:erasure", "area:match-types". We used the Github API and fetched 1,180 bugs for `scalac` and 429 bugs for Dotty. After excluding the bugs without an explicit fix or a test case, we were left with 1,067 and 366 bugs for `scalac` and Dotty respectively. The final set of bugs $\mathcal{B}'_s$ includes 1,433 bugs coming from both Scala compilers.

Collecting Kotlin bugs: Kotlin developers use the YouTrack issue tracker. Currently, it hosts 40,998 issues and bugs associated with different aspects of the Kotlin compiler, including type inference, code generation, IDE support and Android support.

We examined the tracker to identify issues with type: "bug" or "performance problem", and status: "fixed". Kotlin developers follow a fine-grained categorization for determining the components affected by the issue. This made it easy for us to identify bugs that occur in the implementations of the semantic analyses and type checker. Specifically, all typing compiler issues are assigned to categories prefixed by the term "Frontend". Thus, we searched for Kotlin issues that belong to such categories. Bugs in the lexer and the parser are placed in a dedicated category named "Frontend. Lexer & Parser", so it was easy for us to exclude them. Our search returned 2,189 Kotlin bugs. After running the *post-filtering* step, we ended up with the final set of Kotlin bugs $\mathcal{B}'_k$. This set contains 1,601 elements.

Collecting Groovy bugs: Groovy issues are hosted on a Jira instance that currently contains 9,710 cases. We were interested in Groovy issues that have the "bug" label, are either "closed" or "resolved", and their resolution status is "fixed". To identify typing bugs, we searched for issues assigned to a category named "Static Type Checker". Our Jira query fetched 300 Groovy bugs. The *post-filtering* step produced the $\mathcal{B}'_g$ set consisting of 246 Groovy bugs

2.2.2.2 Analyzing Bugs

The total bug population contains 4,153 bugs. Since the manual analysis of each bug requires a certain amount of time to understand the root cause and the nature of the bug, it was not feasible for us to study every bug in the population. Therefore, we randomly sampled 80 bugs from the bug set of each language, leaving us with 320 bugs for manual analysis in total.

To better understand the nature of the examined bugs, cover a wide range of scenarios on how these bugs are triggered, and reduce the possibility of getting biased, we chose to uniformly study bugs in the selected compilers rather than primarily focusing on a single compiler. To this end, our manual analysis was done in an iterative manner. Specifically, in every iteration, we randomly picked 20 bugs from each language set (*sample selection*, Figure 2.5), and two researchers made a first pass over the selected 80 bugs and excluded bugs that are outside the scope of this study (e.g., a parser bug in a compiler that was mistakenly selected during bug

collection). After agreement, we randomly chose additional bugs, until we had 20 analyzable bugs for each language. Then, the actual analysis of these bugs began. The same two researchers independently studied each of the selected 80 bugs (*analysis*), and tried to assign every bug to categories based on (1) its symptom (RQ1), (2) its root cause (RQ2), and (3) the characteristics of the bug-revealing test cases (RQ4). In particular, regarding RQ1, the researchers considered the description of each bug report to identify differences between the compiler's expected and actual behavior. For RQ2, the researchers studied the discussion among developers and the fix of each bug to locate which specific compiler procedure was buggy, while for answering RQ4, examining the accompanying test cases was sufficient.

Moving forward, for each bug, the researchers together discussed their categorization, until they reached consensus. The procedure described above was repeated four times, i.e., until studying 320 bugs in total. During these four iterations, the researchers revisited, adapted (i.e., split, merged or renamed) the proposed categories, and, if it was necessary, re-assigned each aspect of bugs to other categories. Finally, two additional researchers verified the resulting categorization and discussed their conflicts with the initial researchers until agreement. Notably, having this extra validation step was beneficial in cases where the first two researchers could not reach consensus. In such cases, there was a discussion during validation, where the researchers considered all possible options, and finally agreed to the most "optimal" one.

2.2.2.3 Threats to Validity

One potential threat to internal validity is associated with the selection criteria and representativeness of the examined bugs. We were interested in *fixed* bugs accompanied with a fix and a test case. Such fixed bugs (1) are real bugs, (2) are important for the developers (since they are fixed), and (3) have enough information (i.e., a fix and a test case) to understand and characterize them. Furthermore, our study considered only real bugs rather than enhancements or features (e.g., situations where the typing algorithm is incomplete and can be further improved). To do so, during the selection process, we chose issues explicitly marked with the label "bug" and avoided issues marked as "enhancement" or "feature". This is in line with prior work [Sun et al., 2016c; Jin et al., 2012; Di Franco et al., 2017], where fixed bugs were also studied.

For selecting bugs related to the typing algorithms of compilers, we carefully examined the categorization adopted by each development team, and applied (if possible) the necessary filters for fetching such bugs (e.g., getting all bugs prefixed with the "Frontend" term in case of Kotlin). When we did not have such information (e.g., in the case of `javac`), we did not apply additional filters. We avoided using keywords to reduce the chance of missing relevant bugs. In all cases, during our bug analysis, the selected bugs were manually examined by the first two authors, who excluded irrelevant ones.

A threat to external validity is the representativeness of the selected bugs. To mitigate this threat, we picked a random sample of 320 typing bugs, which is consistent with the literature of bug studies. Specifically, Jin et al. [2012] have manually analyzed 110 real-world performance bugs, Di Franco et al. [2017] analyzed 269 bugs in numerical libraries, Leesatapornwongsa et al. [2016], and Bagherzadeh et al. [2020] analyzed 104 and 186 concurrency bugs in distributed and actor-based systems respectively. Theoretically, as in the work of Mastrangelo et al. [2019], we

can presume that using a random sample of 320 bugs, there is a probability of $(1-0.01)^{320} \simeq 0.04 = 4\%$ of missing a category whose relative frequency is at least 1%, and a probability of $(1-0.015)^{320} \simeq 0.008 = 0.8\%$ of missing a category whose relative frequency is at least 1.5%. Note that in practice, as we were examining bugs in iterations, we observed that it was easy to categorize bugs coming from the third or the fourth iteration, as most of these bugs fitted well in one of the resulting categories.

Another threat to external validity is the representativeness of the chosen languages and their compilers. We selected these programming languages as they hold an important stake in the JVM technology. We argue that the chosen languages can represent, to some degree, other statically-typed, object-oriented languages (e.g., C++, C#, TypeScript). However, some of the findings of our work may not be generalized to languages such as Haskell, OCaml, or Go.

Our manual analysis of bugs may be subjective. To minimize this threat, two researchers independently studied each bug, and then they had a thorough discussion until agreeing on categorization. To further mitigate this threat, two additional researchers verified the categorization performed by the first two researchers. This is consistent with previous empirical studies [Wang et al., 2017; Di Franco et al., 2017; Bagherzadeh et al., 2020], where each bug was inspected and verified by multiple researchers.

2.2.3 Symptoms of Compiler Typing Bugs

In the previous section (§2.2.2), we described our method for collecting and analyzing compiler typing bugs. We now present the main findings of this study providing answers to each of our research questions. Since compiler typing bugs are not well-understood, in the following sections, we shed light on *each* aspect of typing bugs, including their symptoms, bug causes, fixes, and characteristics of their test cases. All references to specific bugs provided as examples are hyperlinked to the corresponding entry in the compiler project's issue tracking system.

Every bug report of our dataset, which we constructed using the method presented in Section 2.2.2.1, consists of a short description that contains information about how the bug is triggered along with the compiler's expected and actual behavior. We manually examined the differences between the compiler's expected and actual behavior, and grouped these differences into categories. We ultimately identified five categories of symptoms, namely, *Unexpected Compile-Time Error*, *Internal Compiler Error*, *Unexpected Runtime Behavior*, *Misleading Report*, and *Compilation Performance Issue*. Figure 2.6 shows the distribution of the symptom categories. In the following, we discuss each symptom category in detail. Further, we elaborate on its frequency and impact, and present a concrete example of a compiler bug associated with the symptom.

2.2.3.1 Unexpected Compile-Time Error

A bug involving this symptom manifests itself when the compiler rejects a *well-typed* program, producing an informative error message to the developer. Such errors may frustrate developers, leaving them with the impression that their programs are indeed incorrect. *Unexpected compile-time error* is by far the most common symptom, accounting for 50.94% of the examined bugs.

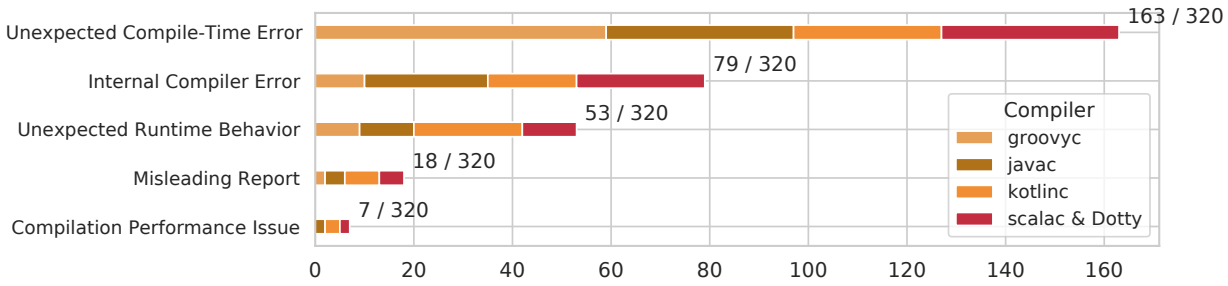


Figure 2.6: The distribution of symptoms.

Example bug: KT-10711

Figure 2.7 shows an instance of this symptom (related to `kotlinc` — see KT-10711). The program includes a parameterized class named `A` that takes one type parameter `T`, and defines a property named `f` whose type is given by the type parameter (line 1). Later, the program creates a list of strings (line 3). To convert every element of this list into an object of class `A`, the code applies the function `map` by passing a reference to the constructor of class `A`. `map` is a parameterized method in class `List` whose signature is `<I, O> List<I>.map(fun: I => O): List<O>`. Specifically, `map` (1) is instantiated with two type parameters, `I` and `O`, (2) is applied to a list of type `List<I>`, (3) expects a function with type `I => O` as input, and (4) returns a value of type `List<O>`.

The Kotlin compiler rejects the above program with the error message: *"error: not enough information to infer type variable T"*. Specifically, `map` is applied to a list of strings. Thus, the type variable `I` of function `map` is instantiated with a `String` type, making `map` expect a function of type `String => O` as input. However, there is a bug in the inference engine of `kotlinc`, which prevents the compiler from instantiating the type variable `T` defined in class `A` (and as a result, the corresponding type of function reference `::A`) based on the expected function type `String => O`.

```
1  class A<T>(val f: T)
2  fun test() {
3    listOf<String>().map(::A)
4  }
```

Figure 2.7: KT-10711: A program that triggers a `kotlinc` bug with an *unexpected compile-time error*.

In the above example, the compiler considers the program as invalid and produces a corresponding diagnostic message. Other similar types of wrong error messages involve type mismatches (e.g., inferred type is `X`, but `Y` was expected), unresolved references (e.g., cannot find method `m`), and accessibility issues (e.g., private variable cannot be accessed in this context).

2.2.3.2 Internal Compiler Error (or Crash)

This is the second most common symptom in our dataset (24.69%). Such errors manifest themselves when the compiler terminates its execution abnormally. This symptom differs from *unexpected compile-time error*, because the compiler is unable to yield a normal diagnostic message, or even generate target code. Internal compiler errors are clear indications that something is not working well in the compiler.

Example bug: GROOVY-7618

Figure 2.8 presents a Groovy program that triggers a bug (see GROOVY-7618) leading to an *internal compiler error*. The program defines a *Single Abstract Method (SAM)* interface named `I` containing an abstract method `m` that takes no parameters and returns an integer (lines 1–3). Note that every SAM interface is also a functional one, meaning that instead of concrete classes, such SAM interfaces can also be implemented by lambda expressions and functions. The program later defines a function called `m2` that expects an instance of `I`, and returns a value of `int` by calling the method `m` of the given instance. Finally, the program calls `m2` by passing a lambda as an argument (line 8). The type of lambda is `() => int`.

```
1 interface I {
2   int m()
3 }
4 int m2(I x) {
5   x.m()
6 }
7 void test() {
8   m2 { -> 1 }
9 }
```

Figure 2.8: GROOVY-7618: A program that triggers a `groovyc` bug with an *internal compiler error*.

`groovyc` internally represents a lambda expression with an object, which among other things, contains a field named `params` that stands for the parameter list of lambda. While coercing the type of lambda to a SAM type for type checking the call at line 8, `groovyc` first computes the arity of lambda by accessing the property `params.length`. Nevertheless, the given lambda is parameterless (line 8), and therefore the value of `params` is `null`. This in turn, leads to a `NullPointerException`, because the `params.length` access is not guarded by a null-check of the receiver (`params`).

Internal compiler errors occur because of the following reasons: there are (1) operations on unvalidated data that trigger unexpected runtime exceptions (e.g., `NullPointerException`, `ArrayOutOfBoundsException`, `ClassCastException`), (2) failures of assertions included in the compiler's source code or custom exceptions thrown when the compiler validates input data that are in an illegal state, and (3) infinite loops in recursive computations leading to a `StackOverflowError`. We found that 35 internal compiler errors were triggered by unexpected runtime exceptions, 35 by assertion failures and custom compiler exceptions, and 9 by infinite loops. As an example of an assertion failure, consider Dotty-7041, where Dotty performs a post-condition check after type erasure to ensure that all types of the program tree are erased and are consistent with the type of system of the JVM. This is not the case in this issue, where after type erasure the program tree contains an illegal type leading to an `AssertionError`.

2.2.3.3 Unexpected Runtime Behavior

The 16.56% of our bugs come with an *unexpected runtime behavior* symptom. Unlike previous symptoms, a bug related to an *unexpected runtime behavior* manifests itself when running the executable generated by the compiler. This involves the successful compilation of a given source program and the generation of a faulty executable that in turn, may lead to errors and wrong outcomes.

There are two reasons why a compiler may generate incorrect executables. First, a compiler bug can break the soundness of the type system. Hence, the compiler accepts an *invalid* program which it should have rejected. Such bugs are important, because they defeat the safety offered by type systems in statically-typed languages [Milner, 1978]. Second, the compiler may perform wrong static linking between methods and objects (e.g., it chooses the wrong overloaded method to call). Like miscompilations caused by optimization bugs, typing bugs with *unexpected runtime behavior* are very confusing for developers, and worse, they may be released unnoticed, as many of these unexpected runtime behaviors are triggered by specific application inputs.

Example bug: JDK-7041019

Consider the Java program of Figure 2.9, which causes a known `javac` bug (see JDK-7041019) associated with an *unexpected runtime behavior* symptom. First, the code defines a parameterized interface `A` which is instantiated with a type parameter `E`. This interface contains an abstract method `m` expecting a value of `E` (lines 1–3). Another parameterized interface called `B` has one type parameter (`Y`), and extends the interface `A` instantiated as `A<Y[]>` (line 4). Later, a class called `C` implements `B<Integer>` by overriding the abstract method `m` (line 7). Furthermore, on lines 8–11, class `C` defines a static parameterized method, `m2`. This method defines a type variable `T` with upper bound `B<?>`. Also, `m2` receives a parameter `x` whose type is `T`, and returns nothing. The body of this method calls `x.m()` by passing an array of strings as input (line 10). Finally, the code defines `main`, which invokes `m2` using an instance of `C` as a call argument (lines 12–14).

```
1  interface A<E> {
2    void m(E x);
3  }
4  interface B<Y> extends A<Y[]> { }
5  class C implements B<Integer> {
6    @Override
7    void m(Integer[] x) { }
8    static <T extends B<?>> void m2(T x) {
9      //Boom! ClassCastException at runtime.
10     x.m(new String[]{"s"});
11   }
12   static void main(String[] args) {
13     m2(new C());
14   }
15 }
```

Figure 2.9: JDK-7041019: A program that triggers a `javac` bug with an *unexpected runtime behavior*.

`javac` compiles this program successfully, and produces the corresponding bytecode. Unfortunately, the JVM throws a `ClassCastException` when running the method call on line 10. This is because the JVM tries to pass an array of strings in a method expecting an array of integers (notice that at runtime, the receiver of the callee method `m` is an object of class `C`, see lines 7, 10, 13)! This soundness issue is caused by a bug in `javac`. Specifically, when typing the method call at line 10, `javac` instantiates the expected type of `m` (which at that time is `X[]`, where `X` stands for a fresh type variable) based on the upper bound of type parameter `T` (i.e., `B<?>`) of method `m2`. `javac` substitutes the type variable `X` with a capture type represented as `CAP#1`, but instead of creating an array type holding elements of type `CAP#1`, it mistakenly creates an array type that stores elements of type `Object`. After this incorrect type substitu-

tion, `m` now expects something of type `Object[]`. In Java though, arrays are covariant, thus, `javac` treats the argument type `String[]` as a subtype of the expected type `Object[]`, and mistakenly allows the call at line 10.

Some common runtime behaviors caused by typing bugs with an *unexpected runtime behavior* include bytecode verification failures (`VerifyError`), dynamic linking and resolution failures (`AbstractMethodError`, `IllegalAccessError`), execution failures (`NullPointerException`, `ClassCastException`), or wrong execution results.

2.2.3.4 Misleading Report

The fourth most common symptom is *misleading report* (5.62%). Such symptoms appear when for a given program, the compiler emits a false warning or a false error message. False warnings and error messages may be misleading because they suggest ineffective fixes (e.g., warning about an unsafe cast, but the cast is actually safe). Furthermore, spurious messages can hide other program errors (e.g., the compiler reports a type mismatch error instead of an uninitialized variable error). Unlike *unexpected compile-time error*, in case of *misleading report*, the compiler correctly accepts (or rejects), a valid (or invalid) program. However, it does so by producing wrong diagnostic messages.

Example bug: KT-5511

Figure 2.10 shows a Kotlin program triggering a bug (see KT-5511) with a *misleading report* symptom. The code defines a parameterized interface named `X` instantiated by one type parameter `T` (line 1). Inside the body of `X`, the code declares an inner enum named `C` that implements `X<T>`.

```
1 interface X<T> {
2     inner enum class C : X<T>
3 }
```

Figure 2.10: KT-5511: A program that triggers a `kotlinc` bug with a *misleading report*.

For this program, `kotlinc` generates two compile-time error messages: (1) *"error (2, 3): Modifier 'inner' is not applicable to enum class"*, and (2) *"error (2, 26): Expression is inaccessible from a nested class 'C', use 'inner' keyword to make the class inner"*. These two error messages are *contradictory*: the first message says that enum class cannot be inner, while the second one suggests developer make the enum class inner. This example program is indeed invalid, and the first error message is correctly reported by the compiler. However, the second error of the compiler is spurious, and it is caused by a bug in the reporting mechanism of `kotlinc`.

2.2.3.5 Compilation Performance Issue

The least common symptom is *compilation performance issue* (2.19%). Bugs related to this symptom cause noticeable degradations in compilation performance. The impact of such bugs is the waste of developers' time and resources, because the compiler requires much time or memory to compile even the simplest fragment of code, and in many cases, compilation never terminates.

Example bug: Dotty-10217

Consider the Scala program of Figure 2.11, which triggers a bug in Dotty (see Dotty-10217). The program defines 23 types (most of them are omitted for brevity): from type `A` to type `W`. Then, the program defines a type constructor `Foo` which is instantiated with one type variable `T`. Finally, the code declares one variable named `f` with a type that comes from the application of type constructor with a union type consisting of types `A` to `W`.

```
1  trait A
2  trait B
3  trait C
4  ...
5  trait W
6  trait Foo[T]
7  val f: Foo[A | B | C | ... | W] =
       ???
```

Figure 2.11: Dotty-10217: A program that triggers a Dotty bug with an *compilation performance issue*.

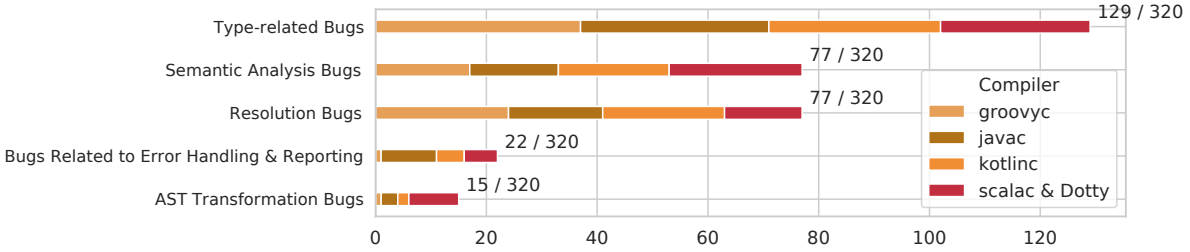
Dotty spends roughly five minutes to compile this program. Specifically, Dotty performs a type optimization on union types: a union type of the form `T | Null` or `Null | T` becomes a regular type `T`. In this context, Dotty examines the union type passed as a type argument of type constructor `Foo` (line 7) to see whether this optimization is applicable to this union type. To do so, Dotty recursively checks if the union type consists of a bottom type (i.e., `Null` or `Nothing`) by using an internal function named `derivesFrom`, which returns true if a given type is an instance of a given class (e.g., `NothingClass`). The complexity of `derivesFrom` is exponential, which means that for a union type containing 23 terms, Dotty performs 2^{23} calls to `derivesFrom`!

2.2.3.6 Comparative Analysis

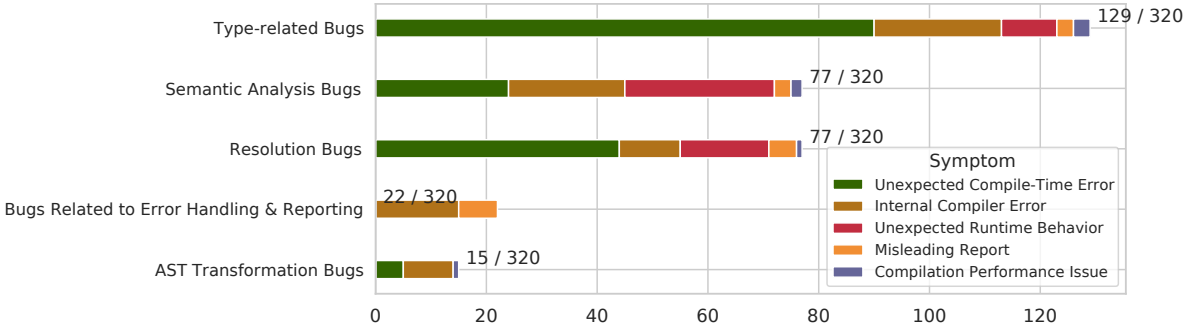
From Figure 2.6, we observe similar trends among studied compilers. The *unexpected compile-time error* symptom is the most common symptom for all compilers followed by *internal compiler error*, and *unexpected runtime behavior*. The only exception is `kotlinc`, where *unexpected runtime behavior* is the second most common symptom category. Specifically, 22 `kotlinc` bugs were marked with an *unexpected runtime behavior* symptom, while 18 bugs were crashes. The high number of `kotlinc` bugs with an *unexpected runtime behavior* symptom is explained by missing well-formed checks on declarations in the compiler's implementation, which may be attributed to Kotlin's immaturity compared to the other compilers. An example of such a missing check is that a Kotlin class must not implement two interfaces containing members with conflicting signatures. Around three quarters of `groovyc` bugs (59 out of 80) make the compiler reject valid code, while we found only ten `groovyc` crashes compared to 18, 25, and 26 crashes found in the Kotlin, Java, and Scala compilers. Finally, we did not observe any `groovyc` bug causing any compilation performance issue.

2.2.4 Bug Causes of Compiler Typing Bugs

We classified the examined bugs into categories based on their root cause. To do so, we studied the fix of each bug and identified which specific compiler's procedure was buggy. From our manual inspection, we derived five categories that include bugs sharing common root causes:



(a) The distribution of bug causes per compiler.



(b) The distribution of bug causes per symptom.

Figure 2.12: The distribution of bug causes.

Type-related Bugs, *Semantic Analysis Bugs*, *Resolution Bugs*, *Bugs Related to Error Handling & Reporting*, and *AST Transformation Bugs*. Figure 2.12 illustrates the distribution of our bug causes. In the following, we provide descriptions and examples for every category.

2.2.4.1 Type-Related Bugs

To type check an input program, a compiler consults the type system of the language. A type system provides a set of rules of what are the language main types, what operations on these types are valid, how these types relate to each other, and how they can be combined. In this context, a compiler internally represents all types and properties of the underlying type system using specialized data structures. Further, when typing an input program, a compiler applies a broad spectrum of operations to these data structures based on the rules and design of the type system. Corresponding examples include type variable substitutions, type constructor applications, subtyping checks, type normalizations, and more.

We define a *type-related bug* when one of these type operations is not implemented correctly. Since types and their operations are at the heart of a compiler, such correctness issues have a great impact on the ability of the compiler to accept the given code. Therefore, type-related bugs are mainly responsible for unexpected compile-time errors (see Figure 2.12b). We classified 129 out of 320 (40.31%) bugs as *type-related*, which makes this bug cause the most common one. Type-related bugs belong to one of the following groups: (1) *incorrect type inference & type variable substitution*, (2) *incorrect type transformation / coercion*, or (3) *incorrect type comparisons & bound computations*.

Incorrect Type Inference & Type Variable Substitution. In languages supporting type inference, explicit types may be omitted in a program. The compiler represents these omitted

types with type variables, which in turn, are replaced with concrete types at compile-time, typically by solving a type constraint problem. Many type-related bugs are caused by building a wrong constraint problem (e.g., the constraint system contains excessive, missing, or contradictory constraints), or instantiating a type variable in a wrong way. As a result, for a certain type variable, the compiler infers a wrong type, or in many cases, it is unable to infer the type at all.

Figure 2.7 gives an example of such a bug. Due to an incorrect handling of function references, `kotlinc` constructs a constraint problem with incomplete constraints. This makes it impossible for the compiler to solve the system and find an optimal solution, leading to an *unexpected compile-time error*. Another example is shown in Figure 2.9. When dealing with an array type containing a type variable, `javac` performs a wrong type variable substitution, which causes a soundness bug.

Incorrect Type Transformation / Coercion. Guided by certain rules, a compiler may transform a certain type into another type for numerous reasons, e.g., type normalization or type erasure. For example, as shown in Figure 2.11, Dotty normalizes a union type of the form `T | Null` to `T`. Another example involves type erasure where all studied compilers erase type information from parameterized types. Similarly, we have the boxing and unboxing processes where a value type becomes a reference type, and vice versa. Diverse bugs in the implementation of these type transformations cause many problems.

Example bug: KT-9630

As an example, consider Figure 2.13, where a Kotlin program triggers KT-9630. Specifically, this program defines a parameterized extension function named `m` instantiated by one type variable `T` that has two upper bounds: `A` and `B` (line 4). The code later calls this function using a receiver of type `C` (line 6). When typing this program, `kotlinc` instantiates type variable `T` with the intersection type `A & B`. Since in Kotlin, intersection types are only used internally for type inference purposes, `kotlinc` needs to convert the intersection type `A & B` into a type that is representable in a program. The problem in this example is that `kotlinc` fails to convert type `A & B` to type `C`. Consequently, `kotlinc` rejects the given code, because it is unable to find the method `m` in a receiver of type `C`, even though this type has been extended with method `m`.

```
1 interface A
2 interface B
3 class C : A, B
4 fun <T> T.m(): Unit where T : A, T : B
      { }
5 fun main() {
6   C().foo()
7 }
```

Figure 2.13: KT-9630: A Kotlin program that triggers a bug related to an *incorrect type transformation*.

Incorrect Type Comparison & Bound Computation. Another instance of type-related bugs are incorrect type comparisons and bound computations. A compiler applies different kinds of comparisons between types, which are underpinned by formal rules and relations included in the underlying type system. For example, a compiler consults the subtyping rules

of the type system to check whether a value of type T_1 is assignable to a variable of type T_2. Beyond that, a compiler implements a number of algorithms dealing with type bounds, such as computation of lowest upper bound and greatest lower bound. We have identified many type-related bugs caused by type comparisons and bound computations that do not obey the rules of the type system.

Example bug: JDK-8039214

Figure 2.14 demonstrates a `javac` bug (see JDK-8039214) caused by an incorrect type comparison. While type checking the call on line 7, `javac` checks whether the argument type `C<?>` is a subtype of the expected type `I<? extends X, X>`. As part of this subtyping check, `javac` tests if the type argument `?` of type constructor `C` is contained in type argument `? extends X` of type constructor `I`. This type argument comparison is guided by the containment relation defined in the Java Language Specification (JLS) [Gosling et al., 2015, §4.1.5]. Unfortunately, the implementation of `javac` does not follow this containment relation to the letter. Hence, it considers that `C<?>` is not a subtype of `I<? extends X, X>`. This makes `javac` reject this well-formed program.

```
1 interface I<X1,X2> {}
2 class C<T> implements I<T,T> {}
3
4 public class Test {
5   <X> void m(I<? extends X, X> arg) {}
6   void test(C<?> arg) {
7     m(arg);
8   }
9 }
```

Figure 2.14: JDK-8039214: A Java program that triggers a bug related to *incorrect type comparisons*.

2.2.4.2 Semantic Analysis Bugs

Semantic analysis occupies an important space in the design and implementation of compiler front-ends. A compiler traverses the whole program and analyzes each program node (i.e., declaration, statement, and expression) individually to type it and verify whether it is well-formed based on the corresponding semantics. A *semantic analysis bug* is a bug where the compiler yields wrong analysis results for a certain program node. The 24.06% of the inspected bugs are classified as semantic analysis bugs. A semantic analysis bug occurs due to one of the following reasons: (1) *missing validation checks*, and (2) *incorrect analysis mechanics*.

Missing Validation Checks. This sub-category of bugs include cases where the compiler fails to perform a validation check while analyzing a particular node. This mainly leads to unexpected runtime behaviors because the compiler accepts a semantically invalid program because of the missing check. In addition to these false negatives, later compiler phases may be impacted by these missing checks. For example, assertion failures can arise, when subsequent phases (e.g., back-end) make assumptions about program properties, which have been supposedly validated by previous stages. Some indicative examples of validation checks include: validating that a class does not inherit two methods with the same signature, a non-abstract class does not contain abstract members, a pattern match is exhaustive, a variable is initialized before use.

Example bug: Scala2-5878

Consider the Scala program of Figure 2.15, which demonstrates a semantic analysis bug related to a missing validation check (see Scala2-5878). The program defines two value classes `A` and `B` with a circular dependency issue, as the parameter of `A` refers to `B`, and the parameter of `B` refers to `A`. This dependency problem, though, is not detected by `scalac`, when checking the validity of these declarations. As a result, `scalac` crashes at a later stage, when it tries to unbox these value classes based on the type of their parameter. The developers of `scalac` fixed this bug using an additional rule for detecting circular problems in value classes.

```
1  case class A(x: B) extends AnyVal
2  case class B(x: A) extends AnyVal
```

Figure 2.15: Scala2-5878: A Scala program that triggers a bug related to *missing validation checks*.

Incorrect Analysis Mechanics. Another common issue related to semantic analysis bugs is *incorrect analysis mechanics*. This sub-category contains bugs with root causes that lie in the analysis mechanics and design rather the implementation of type-related operations, i.e., these bugs are specific to the compiler steps used for analyzing and typing certain language constructs. Incorrect analysis mechanics mostly causes compiler crashes and unexpected compile-time errors.

For example, in Dotty-4487, the compiler crashes, when it types `class A extends (Int => 1)`, because Dotty incorrectly treats `Int => 1` as a term (i.e., function expression) instead of a type (i.e., function type). Specifically, Dotty invokes the corresponding method for typing `Int => 1` as a function expression. However, this method crashes because the given node does not have the expected format. Dotty developers fixed this bug by typing `Int => 1` as a type.

2.2.4.3 Resolution Bugs

One of a compiler's core data structures is that representing scope. Scope is mainly used for associating identifier names with their definitions. When a compiler encounters an identifier, it examines the current scope and applies a set of rules to determine which definition corresponds to the given name. In languages like those examined in our study where features, such as nested scopes, overloading, or access modifiers, are prevalent, name resolution is a complex and error-prone task. A *resolution bug* is a bug where the compiler is either unable to resolve an identifier name, or the retrieved definition is not the right one. We found that the 24.06% of our bugs lie in this pattern. These bugs are caused by one of the following scenarios: (1) there are correctness issues in the implementation of resolution algorithms, (2) the compiler performs a wrong query, or (3) the scope is an incorrect state (e.g., there are missing entries). The symptoms of resolution bugs are mainly unexpected compiler-time errors (when the compiler cannot resolve a given name or considers it as ambiguous) or unexpected runtime behaviors (when resolution yields wrong definitions) — see Figure 2.12b.

Example bug: JDK-7042566

Figure 2.16 presents a test case that triggers the `javac` bug JDK-7042566. For the method call at line 4, `javac` finds out that there two applicable methods (see lines 6, 7). In cases where for a given call, there are more than one applicable methods, `javac` chooses the most specific one according to the rules of JLS [Gosling et al., 2015, §15.12.2.2 and §15.12.2.3]. For our example, the method error defined at line 7 is the most specific one, as its signature is less generic than the signature of `error` defined at line 6. This is because the second argument of `error` at line 7 (`Throwable`) is more specific than the second argument of `error` (`Object`) at line 6. However, a bug in the way `javac` applies this applicability check to methods containing a variable number of arguments (e.g., `Object...`) makes the compiler treat these methods as ambiguous, and finally reject the code.

```
1 class Test {
2   void test() {
3     Exception ex = null;
4     error("error", ex);
5   }
6   void error(Object o, Object... p) { }
7   void error(Object o, Throwable t,
8              Object... p) { }
9 }
```

Figure 2.16: JDK-7042566: A Java program that triggers a *resolution* bug.

2.2.4.4 Bugs Related to Error Handling and Reporting

When an error is found in a given source program, modern compilers do not abort compilation. Instead, they continue their operation to find more errors and report them back to the developers. In the context of type checking this is typically done by assigning a special type (e.g., the top type) to erroneous expressions. Compilers also strive to provide informative and useful diagnostic messages so that developers can easily locate and fix the errors of their programs. A *bug related to error handling & reporting* is a bug where the compiler correctly identifies a program error, but the implementation of the procedures for handling and reporting this error does not produce the expected results. We found that the 6.88% of our bugs are associated with error handling and reporting. All bugs of this category are related to crashes and wrong diagnostic messages (i.e., misleading reports).

For example, the Kotlin program of Figure 2.10 triggers a bug related to error handling and reporting. As already discussed, in this program, `kotlinc` produces an excessive diagnostic message. This message suggests developers to take actions that contradict with previously reported messages.

2.2.4.5 AST Transformation Bugs

The semantic analyses of a compiler works on a program's Abstract Syntax Tree (AST). Before or after typing, a compiler applies diverse transformations and simplifications to the AST so that the given program is expressed in terms of simpler constructs. For example, `javac` applies a transformation that converts a foreach loop over a list of integers `for (Integer x: list)` into a loop that employs iterators as follows: `for (Iterator<Integer> x = list.iterator();`

`x.hasNext();)` An *AST transformation bug* is a bug where the compiler generates a transformed program that is not equivalent with the original one, something that invalidates subsequent analyses. We found that the 4.69% of our bugs are AST transformation bugs, which cause many unexpected compile-time and internal compiler errors.

Example bug: Scala2-6714

Figure 2.17 demonstrates an instance of this bug category (see Scala2-6714). This Scala 2 program defines a class `B` overriding two special methods named `apply`, and `update` (lines 2–5). The function `apply` allows developers to treat an object as a function. For example, a variable `x` pointing to an object of class `B` can be used like `x(10)`. This is equivalent of `x.apply(10)`. Furthermore, the `update` method is used for updating the contents of an object. For example, a variable `x` of type `B` can be used in map-like assignment expressions of the form `x(10) = 5`. This is equivalent of calling `x.update(10, 5)`. Notice that in our example, the `apply` method takes an implicit parameter of type `A`. This means that when calling this function, this parameter may be omitted, letting the compiler pass this argument automatically by looking into the current scope for implicit definitions of type `A`.

```
1  class A
2  class B {
3    def apply(x: Int)(implicit a: A) = 1
4    def update(x: Int, y: Int) { }
5  }
6  object Test {
7    implicit val a = new A()
8    val b = new B()
9    b(3) += 4 // compile-time error here
10 }
```

Figure 2.17: Scala2-6714: A Scala program that triggers an *AST transformation bug*.

Before `scalac` types the expression on line 9, it "desugars" this assignment, and expresses it in terms of method calls. For example, `b(3) += 4` becomes `b.update(3, b.apply(3)(a) + 4)`. However, due to a bug, `scalac` ignores the implicit parameter list of `apply`, and therefore, it expands the assignment of line 9 as `b.update(3, b.apply(3) + 4)`. Consequently, the expanded method call does not type check, and `scalac` rejects the program.

As already discussed in Section 2.2.2.3, we may have missed the identification of categories not included in the sample of analyzed bugs. For example, such a missing category may involve bugs concerning program serialization, a process that translates the AST of a program into another storable format, e.g., see the TASTy files of Dotty.[1] Such representations are useful for inspecting the semantic information of the input program and designing custom program analyses.

2.2.4.6 Comparative Analysis

According to Figure 2.12a, type-related bugs form by far the most common bug cause for all studied compilers. This suggests that reasoning about types is a complex and challenging task for compilers, and that the corresponding type system implementations are susceptible to er-

[1] https://docs.scala-lang.org/scala3/reference/metaprogramming/tasty-inspect.html

rors. Type-related bugs, resolution bugs, and semantic analysis bugs are almost uniformly distributed across the studied compilers. The Scala compilers and `javac` are the outliers though, when it comes to the other bug causes. Specifically, we classify more `javac` bugs as *bugs related to error handling & reporting* compared to the remaining compilers. Furthermore, notice that AST transformation bugs are particularly common in Scala compilers. We attribute this to the fact that Scala is a very powerful language, meaning that individual features are often combined together to establish new features and use cases. `scalac` and Dotty apply a large number of transformations (e.g., Dotty implements more than 50 passes until it performs type erasure) that simplify program tree so that complex features are expressed through simpler primitives. AST transformation bugs in Scala are associated with eta expansion, inlining, and desugaring of various language constructs.

2.2.5 Fixes of Compiler Typing Bugs

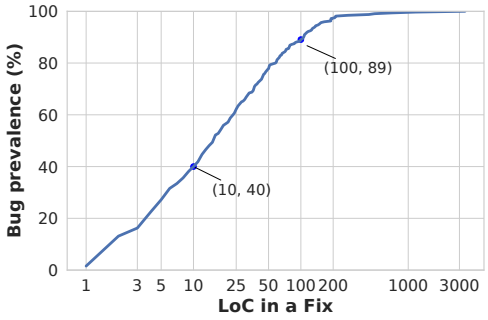
To get an insight into the complexity of compiler typing bugs, we studied how these errors are introduced, and what are the properties of their fixes. We examined the revisions of each bug fix to measure its size and how many components are affected by the fix. Finally, we computed and examined the time compiler developers need to resolve a bug.

2.2.5.1 How Are Bugs Introduced?

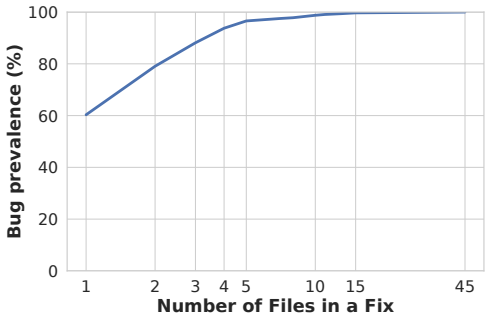
To understand how bugs are introduced, we manually studied every bug fix and the discussion among developers in the corresponding bug reports or commit messages. We found that the bugs of our dataset are mainly introduced by logic errors, algorithmic errors, design errors, or other programming errors.

Logic errors, which stand for defects in logic, sequencing, or branching of a procedure [Zubrow, 2010], are the dominant source of bugs in our dataset (204 / 320). Most of these logic errors are missing cases or steps in the implementation of a routine, or incorrect conditions of an `if` statement. According to Groovy developers, such missing cases are introduced by failing to handle some edge cases (e.g., due to insufficient test coverage) when adding a new compiler feature or making an enhancement. Other instances of logic errors are extraneous computations, incorrect sequence of operations, or wrong/insufficient parameters passed to a function. We observed that the fixes of logic errors usually include changes to a single method or file and consist of few lines of code. For example, many logic errors are fixed by adding a missing `else if` case in the body of a buggy method.

Algorithmic errors are related to errors in the structure and implementation of various algorithms employed by compilers (e.g., inference of a type variable, resolution of a method). Algorithmic errors arise either because the implementation of an algorithm is wrong or because a wrong algorithm has been used. Unlike logic errors, fixes of algorithmic errors usually involve changes in a few dozen lines of code. A characteristic example of this category is Dotty-10217 (Figure 2.11), in which the implementation of the underlying algorithm has exponential complexity. Algorithmic errors were found in 67 / 320 bugs.



(a) Cumulative distribution of lines of code in a fix.



(b) Cumulative distribution of files in a fix.

Figure 2.18: Size of bug fixes.

In contrast to logic and algorithmic errors that describe defects in compilers' implementations, *language design errors* express issues at a higher level. They describe the cases where although the compiler has the intended behavior and is not buggy, a program reveals that this behavior can lead to undesired results. As a result, a re-design is essential for both the language and the compiler. Fixes of design errors include changes from a few code lines to significant refactorings in a compiler's codebase. For example, KT-11280 demonstrates a bug that stems from a design issue in the language. When encountering a condition of the form `if (x == A()) b else c`, `kotlinc` implicitly coerces the type of `x` to type `A` inside the true branch of the `if` statement. However, KT-11280 demonstrates that this behavior is a source of unsoundness. A developer is free to override the method `equals` (this is the method invoked when performing the == operator), meaning that `x` is not guaranteed to have a type of `A` whenever the check `x == A()` returns true. Kotlin designers and developers fixed this by forbidding these implicit coercions whenever `equals` is overridden. We encountered 36 / 320 bugs introduced by an error in the compiler/language design.

Other programming errors we observed include declarations of a variable with an incorrect data type, out-of-bounds array accesses, accesses to null references, and unchecked exceptions. For example, the `groovyc` bug of Figure 2.8 is introduced by a missing null check causing a `NullPointerException`. The fixes of such faults are usually trivial and involve a single change in one line of the code. We ran into such programming errors in only 13 / 320 bugs.

2.2.5.2 Size of Bug Fixes

We considered the revisions of every bug fix of our dataset, and we excluded file modifications and creations related to test files (e.g., test cases) plus all non-source files (e.g., updates in docs). Using automated means, we counted the lines of code modifications made to source files, and we computed how many source files are updated in each fix.

As Figure 2.18a shows, 89% of the bug fixes contain fewer than 100 lines of code, and 40% of the bug fixes are less than 10 lines. These results are consistent with the study of Sun et al. [2016c] that indicates that 92% and 50% of the GCC and LLVM bug fixes include less than 100 and 10 lines of code respectively. On average, the number of lines of code modified in a bug fix is 52, and the median is 16. For completeness, Figure 2.18b shows how many files are modified

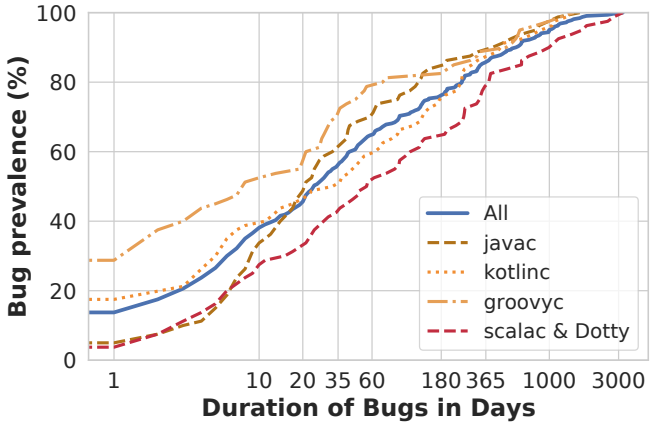


Figure 2.19: Cumulative distribution of bugs through time.

in a bug fix. The majority of fixes change few files: 60% of the patches update a single file, and only 4% of the fixes change more than 5 files. One exception to this pattern is Scala-2742, where the corresponding fix requires updates in the Scala specification, which result in scattered updates across multiple compiler components. In summary, this fix consists of more than 3,000 lines of code and modifies more than 40 source files.

2.2.5.3 Duration of Bugs

Considering the plots in Figure 2.18, a reader may conclude that most of the bugs are simple and easy to fix, because they affect only a small part of the compiler. During our manual inspection we observed that despite the small size of fixes, many bugs are challenging to solve and the developers have long-lasting conversations about potential solutions and their implications. Hence, we decided to investigate the bugs' lifetime to better understand the complexity of bug fixes. To do so, we conducted a quantitative analysis of the time that elapsed in order to fix them. All bug tracking systems of our studied compilers provide details about the creation date and resolution date of each bug report. We defined the duration of a bug as the time interval between its creation and resolution date.

Figure 2.19 shows the bugs' cumulative distribution function over time. The blue plot indicates that over half of the investigated bugs were fixed in one month, and 15% of the bugs took more than a year to be fixed. In terms of days to fix, the median is 24 days and the mean is 186 days. This suggests that many typing bugs are not fixed immediately after a bug report is opened. Indeed, we came across many cases where the corresponding bugs undergo careful examination and risk evaluation by developers and the language committee. This is because fixes of typing bugs can potentially break backward compatibility — a fixed compiler may not be able to compile existing programs that rely on the old compiler's behavior. Therefore, to prevent regression bugs, developers carefully estimate the impact of each suggested fix. For example, after one year of discussions, the Java team decided to address JDK-8075793 so that existing applications written in Java 7 do not break under the new versions of `javac`. Beyond that, many typing bugs are closely related to the language specification and design (e.g., Scala-2742, and KT-22517), and they require fixes and enhancements in both the implementation of

Table 2.2: General statistics on test case characteristics.

Compilable test cases	216 / 320 (67.5%)
Non-compilable test cases	104 / 320 (32.5%)
LoC (mean)	10.2
LoC (median)	8.0
Number of class decl. (mean)	2.0
Number of class decl. (median)	2.0
Number of method decl. (mean)	2.9
Number of method decl. (median)	2.0
Number of method call (mean)	2.5
Number of method call (median)	1.0

the compiler and the design of the language.

2.2.5.4 Comparative Analysis

Consider again Figure 2.19. `groovyc` bugs (see yellow line) need considerably less time to be fixed than the bugs of the other compilers. Specifically, the median duration of `groovyc` bugs is only 8 days, while the median duration is 21, 34 and 55 days for `javac`, `kotlinc`, and Scala bugs respectively. One explanation to this deviation could be that some parts of `groovyc` (e.g., the type checker) may be less mature than the other compilers, and many `groovyc` bugs are programming errors (e.g., a GROOVY-7618, Figure 2.8), which can be fixed easily (e.g., by adding a `null` check), rather than defects that require much domain expertise and knowledge. Another explanation may lie in the motivation and resources associated with the project's development team.

We also performed the Mann-Whitney U test on the distributions of bugs' duration. We found that the duration of Groovy and Scala bugs is statistically different than that of Kotlin and Java bugs, while the durations of Kotlin and Java bugs are not.

2.2.6 How Are Compiler Typing Bugs Triggered?

We now present a discussion on the characteristics of the bug-revealing test cases. Studying the characteristics of test cases gives us an intuition regarding what language features are promising for uncovering typing bugs.

2.2.6.1 General Statistics

Table 2.2 presents some general statistics on test cases. Roughly 67.5% of the inspected bugs are triggered by compilable test cases. However, around one third (32.5%) of typing bugs occurs when compiling invalid code, e.g., the corresponding test case contains type mismatches or ill-formed declarations. This is an important observation, because in addition to using valid test cases (as prior work did for detecting optimization bugs [Le et al., 2014; Yang et al., 2011; Livinskii et al., 2020]), identifying typing bugs requires passing *non-compilable* programs as input to the compiler under test. These incorrect programs mainly trigger bugs that cause internal compiler errors, unexpected runtime behaviors, and misleading reports. The average

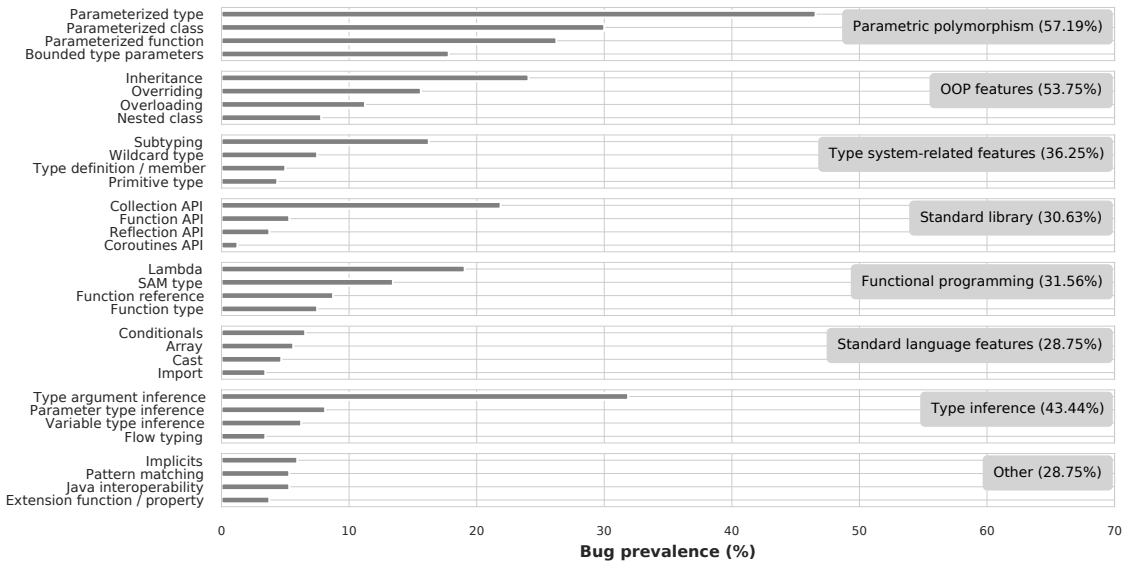


Figure 2.20: The classification of the language features that appear in test cases, along with their frequency. For each category, we show the four most frequent features. Refer to Table A.1 of Appendix A for examining all language features encountered in our bug-revealing test cases.

size of test cases is 10.2 lines of code (LoC), while the median is 8 LoC. This suggests that typing bugs are mainly triggered by small fragments of code.

2.2.6.2 Language Features

We also identified what specific language features are involved in each test case. Since the studied languages are primarily object-oriented, we excluded prevalent object-oriented features that we encountered in almost every test case (e.g., class declaration, object initialization). Then, we grouped the features exercised in every test case into eight categories: (1) *standard language features* containing features seen in almost every language (e.g., exceptions, casts, loops), (2) *Object-Oriented Programming (OOP) features* (e.g., overriding, inheritance), (3) functional programming features (e.g., higher-order functions), (4) *parametric polymorphism* (e.g., parameterized functions), (5) *type inference features* (e.g., type argument inference) (6) *type system-related features* (e.g., subtyping), (7) *standard library* (e.g., use of collection API), and (8) *other* including features not belonging to any of the previous categories (e.g., named arguments).

For every category of features, Figure 2.20 presents its frequency along with its four most frequent features. Parametric polymorphism is pervasive in the corresponding bug-revealing programs: more than half of the examined bugs (57.19%) are caused by test cases containing features, such as declaration and use of parameterized functions, use-site variance, and bounded type parameters. Another interesting observation is that around one third of test cases employ the standard library, and especially the collections API. Collections API includes functions and classes for creating and manipulating data structures (e.g., lists). An example of such a test case is the program of Figure 2.7. Other frequent features are: inheritance for OOP features, subtyping (e.g., `A x = new B()`) for type system-related features, lambda expressions for functional programming features, conditionals for standard features, type argument inference

Table 2.3: The five most frequent and the five least frequent features supported by all studied languages. For another version of this table that displays the 30 most frequent and 30 least frequent features, refer to Table A.2 of Appendix A.

Most frequent features		**Least frequent features**	
Feature	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	46.56%	Multiple 'implements'	2.19%
Type argument inference	31.87%	'this' expression	2.19%
Parameterized class	30.00%	Arithmetic expression	1.88%
Parameterized function	26.25%	Loops	1.25%
Inheritance	24.06%	Sealed class	0.94%

(e.g., `X<String> x = new X<>()`) for type inference features, and Scala implicits for *other*.

Table 2.3 shows which features are the most frequent and which features are the least frequent in the examined test cases. This table presents features that are supported by all studied languages. Features associated with parametric polymorphism (i.e., usage of parameterized types, functions and classes) and their combinations with type argument inference are highly common in the bug-revealing test cases. However, features like arithmetic expressions and loops have a small bug-triggering capability, as they appear only in 1–2% of the bug-revealing programs. This finding contradicts prior testing efforts [Le et al., 2014; Livinskii et al., 2020] for optimization bugs, which rely on programs with complex arithmetic expressions, control- and data-flow (e.g., nested loops).

To find out whether there are any interesting features' combinations that are more likely to trigger bugs, we also computed the lift score, which has been also used in previous bug studies [Jin et al., 2012], For two features A and B, the lift score is given by:

$$\mathit{lift}(A, B) = \frac{P(A \cap B)}{P(A)P(B)}$$

where $P(A \cap B)$ is the probability of a test case containing both features A and B. The lift score gives an estimation of how strongly two features are correlated. A lift score greater than 1 means that the features are positively correlated: when a test case contains feature A, it is also likely to contain feature B. A lift score close to 1 indicates no correlation, while a lift score smaller than 1 denotes that the features are negatively correlated.

The most positively correlated categories are *standard library* with *functional programming features*, and *standard library* with *type inference features* with a lift score of 5. Indeed, many bug-revealing test cases invoke higher-order methods coming from the standard library. As an example, consider the higher-order function `map` used in the program of Figure 2.7. Interestingly, such test cases often let the compiler infer some type information, e.g., the signature of lambda expressions, or the type arguments of a callee parameterized function. Regarding individual features, some interesting combinations are: (1) variable arguments with overloaded methods (e.g., Figure 2.16) with a lift score of 24, (2) use-site variance with parameterized function (e.g., Figure 2.9) with a lift score of 17.1, (3) type argument inference with parameterized function (e.g., Figures 2.9, 2.13) with a lift score of 12.7, (4) Scala implicits with parameterized

Table 2.4: The five most bug-triggering features per language. For another version of this table that displays the 20 most bug-triggering features, refer to Table A.3 of Appendix A.

Java		Scala		Kotlin		Groovy	
Feature	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	51.25	Parameterized type	57.50	Parameterized type	36.25	Parameterized type	41.25
Type argument inference	42.50	Parameterized class	42.50	Parameterized class	33.75	Collection API	35.00
Functional interface	37.50	Inheritance	32.50	Type argument inference	32.50	Type argument inference	35.00
Parameterized function	35.00	Implicits	23.75	Parameterized function	26.25	Lambda	25.00
Parameterized class	30.00	Parameterized function	22.50	Inheritance	25.00	Parameterized function	21.25

class with a lift score of 10.9, (5) type argument inference with collection API (e.g., Figure 2.7) with a lift score of 8.6, and (6) type argument inference with parameterized types with a lift score of of 7.

2.2.6.3 Comparative Analysis

Table 2.4 shows the five most bug-triggering features per language. Again, parametric polymorphism-related features are in the top of every language under study. Another interesting finding is that implicits, a powerful and popular Scala-only feature [Křikava et al., 2019], appears in 23.75% of the examined `scalac` and Dotty bugs. This implies that in addition to parametric polymorphism, Scala implicits is a feature that researchers and Scala developers could profitably invest time to deeply test. Beyond implicits, other language-specific features that are common are: pattern matching (21.25%), higher-kinded types (13.75%), and algebraic data types (13.75%) for Scala, as well as nullable types (16.25%), and extensions (15%) for Kotlin.

2.2.7 Challenges

In the previous sections, we presented the main findings and observations that stem from the study of numerous typing bugs found in the compilers of four popular JVM programming languages. These findings and observations provide us with a first characterization of compiler typing bugs, which entails new implications and challenges for bug-finding techniques. Below, we discuss the main challenges derived from the results of our study.

Challenge 1: Compiler typing bugs have diverse manifestations. Contrary to optimization bugs, which mainly manifest as errors at runtime [Le et al., 2014; Yang et al., 2011], typing bugs can potentially affect both compilation and the runtime (Section 2.2.3). A testing technique must employ appropriate test oracles that can capture typing bugs with a plethora of manifestations. For example, for finding bugs that manifest as unexpected compile-time errors, a fuzzer should generate programs that are valid by construction so that rejection of these programs indicates a potential bug. Similarly, for detecting bugs with a *misleading report* symptom, a fuzzer should generate or use programs with known compile-time errors or warnings, and compare these expectations with the actual ones using a form of pattern matching (e.g., via regular expressions).

Challenge 2: Compiler typing bugs are located in few specific compiler components. According to Figures 2.18a and 2.18b (which show that bug fixes are mostly local), typing bugs are caused by incorrect implementations of few and specific compiler tasks and routines.

In Section 2.2.4, we showed that these buggy tasks and routines are typically associated with operations on types (e.g., type inference), name resolution, semantic analysis of declarations, desugaring, or error handling & reporting. Therefore, a bug-finding tool must employ targeted methods for identifying bugs in these specific compiler components. For example, for finding bugs related to type inference, a mutation strategy could gradually remove type information from a program, e.g., from variable declarations, or type constructor applications. For triggering bugs in resolution algorithms, a promising approach could be the creation of programs that contain and use many overloaded methods or nested declarations. Similarly, for detecting missing validation checks, a potential mutator could inject faults in the program's declarations, e.g., it could inject a circular dependency as in the program of Figure 2.15.

Challenge 3: A large number of typing bugs is triggered by non-compilable programs. Almost one third of the studied bugs is triggered by invalid code (Table 2.2). This observation comes in contrast to existing compiler testing techniques, which feed compilers with compilable programs [Yang et al., 2011; Le et al., 2014; Livinskii et al., 2020]. Generating incorrect programs is quite a challenging task, as the generated programs must be *subtly* incorrect, meaning that the programs should be syntactically correct and contain at most one semantic error. This is because subtly incorrect programs makes it less obvious for the compiler to locate the error. A future research direction could be the proposal of new program generators and mutators that construct such invalid test cases. However, since the search space of invalid programs is enormous, a challenge related to this is to determine the program point to inject the fault, and the nature of the injected fault (e.g., type mismatch error or non-static method in a static context call error). To address this, a technique similar to *Skeletal Program Enumeration (SPE)* [Zhang et al., 2017] could be used to enumerate all subtly invalid programs based on a given program structure.

Challenge 4: Parametric polymorphism is the feature with the most bug-triggering capability, while control-flow constructs and arithmetic expressions do not trigger compiler typing bugs. Test cases that make an extensive use of parametric polymorphism-related features are responsible for more than half of the examined bugs (Tables 2.3, 2.4). Therefore, parametric polymorphism is a promising feature that a program generator should consider for uncovering typing bugs. Finally, our findings suggest that parametric polymorphism works well with type argument inference (Section 2.2.6.2). Therefore, generating programs involving parameterized types and functions with omitted type arguments, or variables with omitted declaration types is another emerging challenge.

At the same time, Table 2.4 shows that control-flow constructs (e.g., loops) and arithmetic expressions barely trigger compiler typing bugs. This *conflicts* with the design and motivation of prior approaches [Yang et al., 2011; Livinskii et al., 2020; Nagai et al., 2014]. For example, as an effort to uncover optimization bugs, the recent work of Livinskii et al. [2020] adopts a generation policy that creates complex arithmetic expressions and bitwise operations. Our findings suggest that the existing techniques should be adapted so that they also consider features that are more likely to cause typing bugs. This would lead to a more holistic testing of compilers.

Challenge 5: Use of the standard library is pervasive in test cases. Based on our observation that around one third of our test cases use the standard library and particularly

```
1  from django.db import models
2  class Person(models.Model):
3    age = models.IntegerField()
4    name = models.CharField(max_length=20)
5  ...
6  p1 = Person(age=31, name="John")
7  p1.save()
8  p2 = Person.objects.get(age=32)
9  p2.delete()
```

Figure 2.21: Example CRUD operations using the Django ORM.

the collections API, an interesting challenge for stress-testing compilers could be the design of a smart program generation algorithm that creates small, self-contained test cases, without requiring the generation of the corresponding definitions (e.g., see Figure 2.7, line 3). These small test cases could involve small expressions that use collections APIs in a complex manner. Interestingly, such APIs heavily rely on parametric polymorphism.

Challenge 6: The characteristics of compiler typing bugs are uniformly distributed across the examined compilers. Almost all the aspects of typing bugs (e.g., symptoms, root causes, and test case characteristics) are uniformly distributed across the studied compilers (see, Sections 2.2.3.6, 2.2.4.6, 2.2.5.4, and 2.2.6.3). For example, as Table 2.4 shows, features related to parametric polymorphism exhibit the same bug-triggering capability in the all the studied languages. This mandates the introduction of generic techniques that are applicable to more than one compiler. For example, a mutator that converts a given class / function into a parameterized one could be invaluable for testing multiple compiler implementations, e.g., such a mutator could be applied to testing both `javac` and `kotlinc`.

2.3 Bugs in Data-Centric Software

In this section, we describe bugs in data-centric software by examining the reliability of ORM systems, a kind of software that is in a widespread use in modern web and cloud applications. However, we believe that our work on ORM systems can also pave the way in doing the same for other data-centric systems, including database and stream processing implementations.

2.3.1 Object-Relational Mapping (ORM) Systems

As already noted in Section 1.2, Object-Relational Mapping provides an abstraction over relational data that enables programmers to interact with their databases through the object-oriented programming paradigm. In this context, a database schema (tables and their inter-relationships) is abstracted through classes, called *models*, and the associated database records are represented via objects of these classes. ORM systems then provide a rich API for basic Create, Read, Update, and Delete (CRUD) operations on database records, as well as more advanced features, like transaction management or query caching.

```
1  q1 = T1.objects.using("mysql")
2  q2 = T2.objects.using("mysql")
3  q3 = T3.objects.using("mysql")
4
5  //ProgrammingError: "You have an error in your SQL syntax"
6  q1.union(q2).union(q3)
7
8  // Generated SQL
9  (SELECT `t1`.`id` FROM `t1`)
10 UNION (
11   (SELECT `t2`.`id` FROM `t2`)
12   UNION
13   (SELECT `t3`.`id` FROM `t3`))
```

Figure 2.22: Django generates MySQL query with invalid syntax.

Figure 2.21 shows an example of database interactions using the Django ORM system [Django Software Foundation, 2020c]. The code first declares a class that maps to a table and to the table's associated columns in the underlying database (lines 2–4). For example, the class `Person` is mapped to a relational table with the same name. This relational table includes two columns: (1) an `integer` column named `age`, and (2) a `varchar(20)` column called `name`. Notably, these columns are derived from the fields enclosed in class `Person` (see lines 3 and 4). Using this class, the code then runs a few simple queries. Specifically, the code creates a class object (line 6), and based on this object, creates a new database record by calling the `save()` method (line 7). Then, the code fetches a single record from the database matching certain criteria (line 8), and then deletes this record (line 9).

ORM system APIs offer a higher level of abstraction that hides the mechanics of SQL queries from the programmer. For example, the `save()` method (line 7) leads to an SQL `INSERT` statement of the form `INSERT INTO PERSON (age, name) VALUES (31, "John")`, the `get()` method (line 8) corresponds to a `SELECT` query of the form `SELECT * FROM PERSON WHERE AGE = 32 LIMIT 1`, while the `delete` method (line 9) results in `DELETE FROM PERSON WHERE ID = 2`. All these SQL queries remain transparent to the programmer.

2.3.2 Bug Definitions and Examples

There are bugs found in a plethora of ORM entities and phases, including (1) the mechanism for translating ORM APIs into concrete SQL queries, (2) the database schema mapper for converting an ORM model into a relational database schema, (3) transaction manager, (4) the component for managing and running database migrations, and more. However, in this thesis, we examine ORM bugs found in the query translation mechanism employed by ORMs. These bugs make ORMs retrieve and process wrong data from the storage. We split them into three main categories, which we discuss below.

```
1  expr = (T.col_a + T.col_b).alias("expr")
2  res = T.select(expr)
3    .order_by(expr)
4    .distinct(expr.alias())
```

Figure 2.23: An internal ORM error detected in peewee ORM.

2.3.2.1 Invalid SQL Query

This category of bugs describes cases where an ORM yields either a grammatically- or semantically *invalid* SQL query. This issue causes a runtime failure when running the erroneous SQL query on the underlying database engine. As such, this bug manifests *only* when the buggy ORM system runs the resulting SQL query on the database. This means that the buggy ORM creates an invalid SQL query without raising any error or exception during the query translation process.

Consider the Django query shown in Figure 2.22 (lines 1–6). This query first fetches the records of tables `t1`, `t2`, and `t3` (lines 1–3), and then performs a chain of unions (line 6) in order to combine the result sets of the individual queries using the `UNION` operator. When we run this Django code on MySQL (version 8.0.4), Django produces and runs the SQL query shown on lines 9–13. The issue is that this SQL query is invalid with respect to MySQL's syntax and semantics and the Django program crashes with a *django.db.utils.ProgrammingError: (1064, "Error in SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'UNION'")*. Note that this bug was detected by our work, and was confirmed by the Django developers.

When the Django code shown in Figure 2.22 is run on another DBMS, such as SQLite or PostgreSQL, Django produces a *valid* SQL query. Such inconsistencies indicate that ORM bugs may appear (or not) depending on the underlying DBMS. Although DBMSs share common functionality, they differ significantly from each other [Rigger and Su, 2020c], because every DBMS supports its own SQL dialect. Therefore, an ORM needs to abstract away such differences and take care of running the same ORM code on different DBMSs reliably. Unfortunately, this complicates the design of ORMs: bugs may occur when an ORM fails to produce a valid SQL query with respect to a certain DBMS.

2.3.2.2 Internal ORM Error (Crash)

An *internal ORM error* (or simply *crash*) is a bug that causes an ORM to terminate its execution unexpectedly, without even producing an SQL query.

Figure 2.23 demonstrates such an internal ORM error, which was found in the peewee ORM system. The code first creates an expression called `expr` by adding two columns (columns `col_a` and `col_b`) included in a table named `T` (line 1). Based on this expression, the code later defines a `SELECT` query (line 2). This query applies the `distinct` operator to get unique results with respect to the value of the expression `expr` (line 4). Notice that these unique results are displayed in a descending order (line 3). Unfortunately, the above ORM query triggers a bug that makes peewee crash with the following Python exception: *"TypeError: 'tuple' object does not support*

```
1 expr = (1 + T.col)
2 squared = (expr * expr)
3 T.select(fn.sum(expr), fn.avg(squared)).all()
4
5 // Generated SQL
6 SELECT SUM(1 + "t"."col"),
7          AVG(1 + "t"."col" * 1 + "t"."col")
8 FROM "t" AS "t"
```

Figure 2.24: Logic error detected in peewee ORM.

item assignment". The exception is thrown while peewee is producing the SQL query based on the invoked API methods. Specifically, there is a programming error in handling the AST nodes of the resulting SQL query.

Internal ORM errors are different from the bugs related to *invalid SQL query*, as the former do *not* require the execution of the produced SQL query on the underlying database engine to reveal the error. Instead, ORM execution terminates abnormally prior to SQL query creation.

2.3.2.3 Logic Error

Logic errors contain cases where an ORM produces an SQL that is *both* grammatically- and semantically valid, but this query does not fetch the right data from the database. This is because the generated SQL query does not come in line with the semantics of the invoked ORM methods.

To better illustrate this category of ORM bugs, consider Figure 2.24 that shows another ORM bug detected by our approach. On lines 1–3, the code creates a simple query using the peewee ORM. The query defines a simple expression `expr` given by the addition of a table's column with the integer constant "1" (line 1). The code then forms a query that applies the function `SUM` to `expr`, and `AVG` to the square of `expr` (lines 2, 3). The peewee ORM translates this high-level query into the *incorrect* SQL query shown on lines 6–8. In this SQL query, the expression passed to the aggregate function `AVG` is not in the expected format because the sub-expressions are not wrapped in parentheses: peewee incorrectly produces $\texttt{AVG}(1 + col * 1 + col)$ instead of $\texttt{AVG}((1 + col) * (1 + col))$. This bug was confirmed and fixed by the peewee developers immediately after our report.

Unlike the Django and peewee bugs discussed earlier, the bug of Figure 2.24 is more subtle: Although peewee generates a semantically valid SQL query, this query is not the correct one. Contrary to ORM bugs that result in invalid SQL queries or internal ORM errors, logic errors do not cause runtime failures, crashes, exceptions, or other indications of error. This means that it is harder to detect the flaw in the buggy ORM implementation.

2.3.3 Challenges

To find bugs similar to the ones discussed above, we need to systematically determine whether the SQL query generated by an ORM system is the correct one or not. To do so, we need to define a test oracle. Nevertheless, establishing a test oracle for ORM-specific bugs is not

straightforward. For example, we are unable to decide whether the SQL query generated by Django (Figure 2.22) is incorrect, unless we have domain knowledge that nested unions are indeed supported by MySQL, and therefore, it is a bug from Django which failed to produce a grammatically correct SQL query involving nested unions. Worse, there is no an easy way to tell that the peewee bug of Figure 2.24 is buggy. Although this query runs successfully on all DBMSs, we cannot be sure that this query indeed fetches the expected results from the database.

One of the possible solutions to address the test oracle problem [Weyuker, 1982] (recall Section 2.1.3.3) is to employ *differential testing* [McKeeman, 1998], a generally-applicable method for testing equivalent implementations. In the context of ORM systems, differential testing provides us with an oracle as follows. We feed the same test input (e.g., query) to two equivalent implementations (e.g., Django and peewee), and then compare their results. A mismatch found in the results of the implementations under test indicates a potential bug in at least one of them. For example, using differential testing, we run the query associated with nested unions (Figure 2.22) on MySQL, but this time using the API of peewee. Peewee executes the given query on MySQL without errors. This helps us identify that there is a bug in the Django implementation. Similarly, for the peewee query shown in Figure 2.24, we construct its counterpart written in Django, only to see that Django and peewee produce different results.

However, applying differential testing on ORM systems is not straightforward, and it involves several new challenges.

Challenge 1: Lack of a common specification and input language. ORM systems do not implement a common specification or standard. Therefore, differences in ORM results may be due to valid but inconsistent implementations and not due to actual bugs. Furthermore, each ORM offers its own APIs and, to make matters worse, these APIs may even be exposed through different programming languages. As a result, differential testing cannot be uniformly applied to test ORM systems in a straightforward manner.

Challenge 2: Non-deterministic query results. In some ORM systems, it is possible to write a query that leads to an SQL statement that produces a non-deterministic result, or the result depends on the implementation of the underlying DBMS. An example of such query is when the results are not ordered. In this case, the DBMS is free to return results in any order. Another example is when the resulting SQL query has a column a and an aggregate function in the `SELECT` part, but the query does not define a `GROUP BY` clause on the column a. According to the SQL standard, selecting a column and an aggregate function, without specifying a `GROUP BY` clause leads to an *ambiguous* query whose results are not deterministic. To compare the results of the ORM systems under test in a meaningful way, we have to deal with this non-determinism and ambiguity.

Challenge 3: DBMS-dependent results. As shown previously (see bug of Figure 2.22), there are ORM bugs that are DBMS-specific, i.e., a bug is triggered *only* when the ORM code works on top of a certain DBMS. To effectively capture such bugs, we need to differentially test the ORM systems on multiple DBMSs. At the same time, though, differences between the underlying DBMSs (e.g., two DBMSs may have different semantics on arithmetic expressions) must not affect the comparisons between ORM results. Finally, for performing safe compar-

isons, the ORM code needs to run on a common reference, i.e., the ORM queries need to run on the same relational database, containing the same tables, columns, and setup (including column constraints and indices).

Challenge 4: Data generation. Beyond ORM queries, we have to generate appropriate data to populate the databases so that ORM systems produce *non-trivial* results in response to given queries. In this way, we can reveal logic errors that cause ORMs to fetch the wrong data from the database. For example, it is impossible to detect the peewee bug of Figure 2.24 when the underlying database contains no records, because in such a scenario, the incorrect SQL query returns the same results as the correct query: an empty result set.

2.4 Dependency Bugs in File System Resources

In this section, we aim to understand the nature and the consequences of missing dependency information associated with file system resources. To this end, we study dependency bugs in the context of two popular and everyday development tasks: *automated system configuration* and *automated builds*.

2.4.1 File System-Centric Software

We investigate systems and programs that are vulnerable to issues that arise from dependency bugs in file system resources. In particular, the purpose of these programs is to automatically (1) configure systems (Section 2.4.1.1), and (2) build software artifacts from source code (Section 2.4.1.2).

2.4.1.1 Automated System Configuration

The prevalence of cloud computing and the advent of microservices have made the management of multiple deployment and testing environments a challenging and time-consuming task [Morris, 2016; Delaet et al., 2010; Plummer and Warden, 2016; Spinellis, 2012]. *Infrastructure as Code* (IaC) methods and tools automate the setup and provision of these environments, promoting reliability, documentation, and reuse [Spinellis, 2012]. Specifically, IaC (1) boosts the reliability of an infrastructure, because it minimizes the human intervention which is both laborious and error-prone; (2) ensures the predictability and consistency of the final product, because it eases the repetition of the steps followed to produce a specific outcome; and (3) allows the documentation and reuse of a system's configuration, because it associates the system's configuration with modular code [Morris, 2016; Humble et al., 2006; Visser et al., 2016a; Spinellis, 2012; Xu et al., 2013].

Puppet [Loope, 2011] is one of the most popular system configuration tools used to manage infrastructures [Shambaugh et al., 2016; Rahman et al., 2019]. It abstracts the state of different system entities such as files, users, software packages, or running processes, in a declarative manner using built-in primitives called *resources*. A Puppet program consists of a collection of resources that the underlying execution engine applies one-by-one so that the system eventually reaches the desired state.

By default, any execution sequence of resources is valid, unless there are specific ordering constraints imposed by their inter-dependencies, e.g., an Apache service should run only after the installation of the corresponding package. Developers need to declare these ordering constraints in the program to avoid erroneous execution sequences, such as trying to start a service before the installation of its package. Conceptually, Puppet captures all the ordering relationships defined in a program through a directed acyclic graph and applies each resource in topological ordering. In this context, all the unrelated resources are processed *non-deterministically*. Furthermore, Puppet allows programmers to apply certain resources whenever specific events take place via a feature called *notification*. Notifications propagate changes to related resources, ensuring that their state is up-to-date. For instance, when a configuration file changes, the corresponding service has to be notified so that it will run with the new settings.

As noted above, Puppet allows developers to describe the desired state of a system through a declarative specification language. For example:

```
1  $service_name = "apache2"
2  $conf_file = "/etc/apache2/apache2.conf"
3  package {$service_name:
4    ensure => "installed"
5  }
6  file {$conf_file:
7    ensure => "file"
8  }
9  service {$service_name:
10   ensure => "running"
11 }
```

To abstract the state of the system, the code above declares three types of resources. Specifically, Puppet offers the `file` resource for abstracting the state of the file system, the `package` resource for managing packages, and the `service` resource for abstracting running processes. Using the these resources, the code indicates that the `apache2` package should be installed in the host (lines 3–5), the file `apache2.conf` should exist in the `/etc/apache2/` path (lines 6–8), and that the Apache server should be running (lines 9–11).

Beyond the aforementioned resources, there are other resource types for abstracting diverse aspects of the host machine, including `mount`, `cron`, `sshkey`, `exec`. Also, the Puppet language provides variable declarations that begin with the "`$`" symbol (e.g., see the variables `$service_name` and `$conf_file` on lines 1 and 2), conditionals, loops, and—for reusability—supports the creation of custom resources and classes.

Puppet code is stored in files called *manifests*. Puppet compiles manifests into executables called *catalogs*. Catalogs are JSON documents that specify all the resources that Puppet needs to apply in a particular system to reach the desired state [Puppet Labs, 2018]. Figure 2.25 shows a JSON snippet that is part of the catalog derived from the previous Puppet code. The field `resources` contains the Puppet resources declared in the initial program along with their parameters. During catalog compilation, every variable defined in manifests resolves to its value, e.g., the variable `$conf_file` resolves to `/etc/apache/apache2.conf` (see line 3 of the compiled catalog).

```
"resources": [
  {
    "title": "/etc/apache2/apache2.conf",
    "type": "File",
    "parameters": {
      "ensure": "file"
    }
  },
  {
    /* another resource... */
  }
]
```

Figure 2.25: An example of a Puppet catalog.

Puppet evaluates the compiled catalogs and applies potential changes if the system is not in the appropriate state. For example, if a file does not exist at a certain location, Puppet will create it. Notably, the execution of a catalog is *idempotent* [Puppet Labs, 2016] meaning that the evaluation proceeds only if the current and the desired state of the system do not match. Finally, Puppet supports both a client-server and a stand-alone architecture for applying catalogs in hosts.

2.4.1.2 Automated Builds

Automated builds are an integral part of software development. Developers spend considerable amounts of time on writing and maintaining scripts [McIntosh et al., 2011, 2015] that implement the build logic of their project. Such scripts may involve the compilation of source files, application testing, and the construction of software artifacts such as libraries and executables. The advent of Continuous Integration (CI) together with the complexity of modern software systems have made prominent two important properties related to automated builds: efficiency and reliability [Vakilian et al., 2015; Gligoric et al., 2014a; Visser et al., 2016a; Hilton et al., 2016]. To save computing resources and development time [Hilton et al., 2016; Licker and Rice, 2019], build tools must be capable of coping with complex systems quickly, but without sacrificing the reliability of the final deliverables. Following this direction, new build systems have emerged providing features such as parallelism [Gligoric et al., 2014a; Bazel, 2020; Coetzee et al., 2011], caching [Gradle Inc., 2020a], incrementality [Erdweg et al., 2015; Konat et al., 2018], the lazy retrieval of project dependencies [Celik et al., 2016], and reliable builds by construction [Christakis et al., 2014; Spall et al., 2020; Curtsinger and Barowy, 2022].

Among these features, parallelism and incrementality are in the heart of almost every modern build system. Parallel builds reduce build times by processing independent build operations on multiple CPU cores. Incrementality saves time and resources by executing only those build operations affected by a specific change in the codebase. Both features are vital for a smooth development process, as they significantly shorten feedback loops [Konat et al., 2018; Visser et al., 2016a]. For example, thanks to parallelism, building systems, which consist of million lines of code and thousands of source files, such as the Linux Kernel or LLVM, can be a matter

of a few minutes.

Conceptually, a build is a sequence of tasks that work on some input files, and produce results (output files), potentially used by other tasks. In most build systems, to avoid failures and race conditions caused by parallelism, developers must specify all dependencies in their build scripts, so that the underlying build system does not process dependent tasks in the wrong sequence or in parallel (e.g., linking before compilation is erroneous). For correct incremental builds, developers need to enumerate all source files that a build task relies on. This ensures that after an update to a source file, all the necessary tasks are re-executed to generate the new build artifacts reflecting this change.

Make and Gradle are two popular build systems that support both incremental and parallel builds. These systems are heavily used for automating builds of C/C++ and JVM-based (e.g., Java, Kotlin, and Groovy) projects respectively. Make is one of the most well-established build tools [McIntosh et al., 2015], while Gradle is a modern JVM-based system that has become the de-facto build tool for Android and Kotlin programs [Karanpuria and Roy, 2018; Pelgrims, 2015; Derr et al., 2017].

Make: Make is the oldest build system used today [Feldman, 1979; Licker and Rice, 2019]. It provides a Domain-Specific Language (DSL) that allows developers to write definitions of rules that instruct the system how to build certain targets. For example, the following rule states that building the target `source.o`, which depends on the file `source.c`, requires the invocation of the `gcc` command (line 2):

```
1  source.o: source.c
2      gcc -c $^
```

By default, Make builds every target incrementally, meaning that it generates targets only when they are missing or when their dependent files are more recent than the target. Make uses file timestamps to determine whether a file has changed or not. Also, it provides some built-in variables starting with the symbol "`$`". The most common ones are `$@` and `$^`, which refer to the name of the target (e.g., `source.o`), and the dependencies (e.g., `source.c`) of the current rule respectively. Developers write their Make rules in files called *Makefiles*. In particular, developers can either write their own Makefiles, or use higher-level tools, such as CMake [Martin and Hoffman, 2010] or GNU Autotools [Calcote, 2020], that automatically generate Makefile definitions. CMake offers its own DSL and enables programmers to write rules which in turn are translated into Makefiles. CMake is useful for managing systems with complex structure. Autotools is a collection of tools that configure and generate Makefiles from templates.

Gradle: Although newer than other JVM-based build tools, such as Ant and Maven, Gradle has gained much popularity recently. Currently, around 55% of the most popular Java Github projects use Gradle [Hassan et al., 2017], and it has become the preferred build tool for Kotlin and Android programs [Karanpuria and Roy, 2018; Pelgrims, 2015; Derr et al., 2017]. Gradle is at least two times faster than Maven [Gradle Inc., 2020d], as it offers features, such as parallelism, and a build cache.

Gradle provides a Groovy- and a Kotlin-based DSL which adopts a task-based programming model. In this sense, Gradle programmers assemble build logic in a set of tasks. A task is a fundamental component in Gradle that describes a piece of work needed to be done as part

of a build. Developers can impose constraints on the execution order of tasks. Then, Gradle represents the build workflow as a directed acyclic graph and processes every task in topological ordering. To enable incremental builds, developers need to enumerate the files consumed and produced by each task. In this context, a task is executed only when there is a change to any of its input or output files. Gradle adopts a content-based approach to identify updates: it compares the checksum of the input / output files with that coming from the last build. Consider the following snippet:

```
1 task extractZip {
2   inputs.file "/file.zip"
3   outputs.dir "/extractedZip"
4   from zipTree("/file.zip")
5   into "/extractedZip"
6 }
```

The listing above demonstrates a task named `extractZip` written in Gradle. This task extracts the contents of an archive, namely `/file.zip`, into the directory `/extractedZip`. The input and the output files of this task are declared at lines 2 and 3 respectively. Declaring the input / output makes the task `extractZip` incremental. In this context, Gradle re-executes this task only when any of those files are modified. Notice that an input or an output file can be a directory (see line 3). In this case, Gradle recursively examines the contents of the directory for updates.

Gradle provides an API that developers can rely on to customize their builds or create plugins. A plugin consists of a set of common tasks that can be reused across multiple projects, e.g., consider a plugin that applies a linter to the source files of a project. Up to now, there are more than 3,600 Gradle plugins available for use [Gradle Inc., 2020c].

2.4.1.3 Execution Model of File System-Intensive Software

From Sections 2.4.1.1 and 2.4.1.2, we observe some key similarities in the execution model of configuration management systems and build systems. Indeed, these systems come with a DSL that allows programmers to customize the configuration of their computer systems, or the build process of their software projects. In particular, using a corresponding DSL construct, (e.g., a Puppet resource, a Gradle task, or a Make target rule) developers define a set of *tasks* that perform various updates to the host system, (e.g., creation of new files) by consuming files or other resources that are either already present in the system or produced by other tasks. Based on the above, we can treat a configuration management execution or a build execution as *a sequence of tasks*, where each task is a function that *produces* some files based on some *given* files. Notably, a task is the smallest execution unit in the programming model of these systems.

Every user-defined task is then processed and executed by the underlying execution engine (e.g., Puppet or build system). Since system operations (e.g., operations on the file system) performed by each task are time-consuming, both configuration management and build systems support parallelism by running unrelated tasks simultaneously. Dependencies and ordering constraints between tasks are *explicitly* provided by the programmers, and *not* automatically inferred by the execution engines. Finally, the Puppet notification feature and incrementality

found in build systems work in a pretty similar way: an update to a particular system resource (e.g., a file) triggers the execution of dependent tasks.

From now onwards, and unless specified otherwise, in the context of automated system configuration and automated builds, we use the term *task* to either refer to the application of a system configuration resource (e.g., Puppet resource) or a build task (e.g., a Gradle task or a Make target rule)

2.4.2 Bug Definitions and Examples

Unfortunately, the nature and the features provided by the configuration management and build systems described in Section 2.4.1 can introduce some issues that lead to improper handling of file system resources. All these issues stem from errors introduced by programmers when they fail to enumerate all dependencies and ordering constraints between tasks. As a result, a buggy program may not work as expected when enabling features such as parallelism, incrementality, or resource notification.

In the context of Puppet, tracking all the required ordering constraints, task dependencies, and notifications is a complicated task, mostly because developers are not always aware of the actual interactions of Puppet with the underlying operating system. Notably, such errors can have a negative impact on the reliability of an organization's infrastructure leading to inconsistencies [Shambaugh et al., 2016] and outages [Github, 2014]. For example, Github's services became unavailable when a missing notifier in their Puppet codebase caused a chain of failures, such as Domain Name System (DNS) timeouts [Github, 2014].

Similarly, parallel and incremental builds pose threats to the reliability of the build process and artifacts when they are not used with caution. In particular, build scripts are susceptible to faults because declaring all task dependencies is a challenging and error-prone task [Licker and Rice, 2019; Vakilian et al., 2015; Morgenthaler et al., 2012]. Even best practices [GNU Make, 2020a], and tools [Martin and Hoffman, 2010] for managing dependencies automatically are often insufficient for preserving correctness [Licker and Rice, 2019]. Build failures, non-deterministic and inconsistent build outputs, or time-consuming builds, are inevitably the result of such faults [Licker and Rice, 2019; McIntosh et al., 2011].

Below, we present four categories of dependency bugs that lead to mishandling of file system resources. All the categories of bugs are caused by improper use of advanced features (e.g., parallelism or incrementality) supported by the studied configuration management and build systems. The description of each category is accompanied with a corresponding concrete bug example.

2.4.2.1 Ordering Violations

An execution engine supporting parallelism runs independent tasks *non-deterministically*. This means that the system is free to process unrelated operations in any order for achieving high performance. Non-determinism does not cause any problems to the process, when two tasks are indeed independent, and the one does not depend on the other. However, race conditions emerge, when two tasks are conflicting, but are executed concurrently. As mentioned earlier,

```
1  apply plugin: "java"
2  apply plugin: "com.github.johnrengelman.shadow"
3  shadowJar {
4    classifier = ""
5  }
```

Figure 2.26: A Gradle script with an ordering violation.

```
1  package {"mysql-server":
2    ensure => "installed"
3  }
4  file {"/etc/mysql/my.cnf":
5    ensure => "file",
6    content => "user db settings..."
7    require => Package["mysql-server"]
8  }
9  exec {"Initialize MySQL DB":
10   command => "mysqld --initialize",
11   require => Package["mysql-server"]
12 }
```

Figure 2.27: A Puppet program with an ordering violation.

developers can introduce ordering constraints in their scripts as a side effect of explicitly defining dependencies among conflicting tasks. An *ordering violation* occurs when a developer does not specify ordering constraints between two dependent tasks. Due to non-determinism, ordering violations can lead to unstable code that behaves correctly in some circumstances, but breaks in others.

Figure 2.26 shows an example of an ordering violation found in a build script. Here we have an excerpt of a real-world Gradle script (from the `nf-tower` project) whose goal is to create the fat JAR of a Java application. A fat JAR packages all `.class` files of the current project along with the `.class` files of project dependencies, forming the executable distribution of the project. The code first applies the built-in Gradle plugin `"java"` (line 1). This plugin—among other things—runs two tasks: (1) the task `classes` that compiles all Java files into their corresponding `.class` files, and (2) the task `jar` that generates a JAR file containing only the classes of the current project. In turn, the code employs an external plugin (line 2) containing the task `shadowJar` (lines 3–5) that eventually generates the fat JAR of the project. The problem here is that the name of the fat JAR generated by the task `shadowJar` conflicts with the name of the naive JAR produced by the task `jar`. The tasks `jar` and `shadowJar` do not depend on each other, so Gradle is free to schedule `jar` after `shadowJar`. This erroneous ordering results in incorrect output, i.e., the task `jar` overrides the contents of the JAR file produced by the task `shadowJar`. A fix to this problem is to create a fat JAR with a different name (e.g, changing its classifier at line 4 to `"-all"`). Through our work we have identified this issue and reported it to the developers of `nf-tower` who confirmed fixed it.

As another example, consider the real-world Puppet program shown in Figure 2.27 that sets up a MySQL database in a server. First, the code declares the installation of the `mysql-server`

```
1  CXXFLAGS=-MD
2  OBJS=CMetricsCalculator.o QualityMetrics.o
3  qmcalc: $(OBJS) qmcalc.o
4      gcc -g qmcalc.o $(OBJS) -o $@
5  -include $(OBJS:.o=.d)
```

Figure 2.28: A Make definition that does not capture the dependencies of the object file `qmcalc.o`.

package (lines 1–3), which—among other things—creates the `/etc/mysql/my.cnf` file that contains the default database settings. Then, it configures this file (lines 4–8) whose contents are specified by the `content` parameter at line 6. Note that Puppet evaluates the `file` resource after `package`. This is expressed through the `require` property at line 7. In lines 9 to 12, the program declares the execution of a shell script (`mysqld -initialize`) that prepares the database according to the settings specified in the `/etc/mysql/my.cnf` file. Although the shell command needs to be invoked only after the file `/etc/mysql/my.cnf` is configured (lines 4–8), the `require` parameter at line 11 omits this dependency. Therefore, applying the `exec` resource before `file` makes Puppet set up the database with incorrect settings. The fix to this issue is fairly straightforward: the `require` parameter at line 11 should contain the value `File["/etc/mysql/my.cnf"]`. Again, our work identified this issue and the developers fixed it immediately.

2.4.2.2 Missing Inputs

A *missing input* issue is found in faulty build scripts. We say that a build definition manifests a missing input issue, when a developer fails to define all input files of a particular build task. This leads to faulty incremental builds because whenever there is an update to any of the missing input files, the dependent build task is not executed by the build system. Consequently, the build system produces stale targets and outputs.

As an example, consider the fragment of a Make definition taken from the `cqmetrics` project, illustrated in Figure 2.28. This build creates the executable `qmcalc` by linking the object files `CMetricsCalculator.o`, `QualityMetrics.o`, and `qmcalc.o` (lines 3–4). Every object file is created by a built-in Make rule that compiles each implementation file `.c` with a command of the form `$(CC) $(CXXFLAGS) -c`. By default, the input file of these built-in rules is only the underlying implementation file. For example, the input file of the rule `qmcalc.o` is `qmcalc.c`. However, an object file might also depend on a set of header files. Thus, changing a dependent header file requires the re-generation of the object file. The developers tackle this issue by compiling every object file with the `-MD` flag (line 1). This flag stores all header files that a target relies on into a dedicated dependency file whose suffix is `.d`. The developers include these dependency files in their Makefile on line 5. Although compiling source files with `-MD` follows the best practices for managing Make dependencies automatically [GNU Make, 2020a], the above script is faulty, because only the dependency files of the object files included in the variable `$OBJS` (line 2) are considered. The issue here is that when there is an update to a

header file that the rule `qmcalc.o` depends on, the object file is not re-created. Thus, the final executable `qmcalc` can be linked with stale object files. The fix is to this issue is to add the object file `qmcal.o` to the value of variable `$OBJS`. In this way, the dependency file corresponding to the creation of `qmcalc.o` is also imported into Makefile during the `include` command on line 5. Note that we have identified the aforementioned issue by using our approach for detecting dependency bugs. In addition, we reported the bug to the upstream developers who confirmed and fixed it.

2.4.2.3 Missing Outputs

Another type of dependecy bug found in build systems is *missing outputs*. Missing output bugs are similar to missing input bugs. However, this time the cause of the problem is that a developer does not properly enumerate the output files of a task. As with missing inputs, this issue makes incremental builds skip the execution of some build tasks even if their outputs have changed. Note that missing outputs do not appear in Make-based builds, because Make considers only the timestamp of input files to decide if a target rule must be re-executed.

In case of Gradle, beyond the reliability of incremental builds, missing outputs also affect build efficiency. Gradle caches the output files of a task from previous builds, and reuses them in subsequent ones when input files remain the same. Hence, missing outputs make a Gradle build run slower. Notably, this build cache is an important Gradle feature that makes Gradle much more efficient than other build systems [Gradle Inc., 2020d].

In addition to preserving the correctness and efficiency of incremental builds, Gradle also uses the declared inputs and outputs of tasks to improve the reliability of parallel builds by implicitly specifying ordering constraints between dependent tasks. In particular, Gradle examines the declared outputs / inputs of tasks and adds an *auto-dependency* between a task that creates a file (this file is declared as an output of the task), and another task that consumes the same file (the task declares this file as its input). This prevents Gradle from running two conflicting tasks (e.g, the one produces something consumed by the other) in parallel. Therefore, a missing input / output may also degrade the reliability of parallel builds.

2.4.2.4 Missing Notifiers

This bug category is associated with the Puppet notification feature. Specifically, notifiers are highly important for the configuration of services that operate on a host machine. An update to a resource (e.g., configuration file) could directly affect the state of a service. For example the configuration file of an Apache service lists additional modules that should be loaded into memory during the service's boot time. To ensure that all services are running on a fresh environment, Puppet triggers the restart of a service whenever there is a change to one of the service's dependent tasks via notifications declared by the programmers. A *missing notifier* issue is triggered when the specification of a `service` resource does not include all necessary notification dependencies with other tasks. This can result in services operating on stale environments, something that can have pernicious consequences on the reliability of applications and services that run on a specific machine, affecting millions of users [Github, 2014].

```
1  package {["libssl", "apache2"]:
2    ensure => "latest"
3  }
4  file {"/etc/apache2/apache2.conf":
5    ensure => "file",
6    require => Package["apache2"]
7  }
8  service {"apache2":
9    ensure => "running",
10   subscribe => [
11     Package["apache2"],
12     File["/etc/apache2/apache2.conf"]
13   ]
14 }
```

Figure 2.29: A Puppet program that contains a missing notifier.

A missing notifier issue is illustrated in Figure 2.29. The code first installs the latest version of the `libssl` and `apache2` packages (lines 1–3), configures the file located at `/etc/apache2/apache.conf` (lines 4–7), and then, boots the Apache server (lines 8–14). The `subscribe` primitive (line 10–13) creates a notifier that restarts Apache whenever there is a change to the corresponding configuration file or an update to the `apache2` package (e.g., a newer version is installed in the system). However, the code lacks a notifier from the `libssl` package to the Apache service. During the kickoff of the service, Apache maps the `/usr/lib/libssl.so` library to memory (through the `mmap` system call). This shared library is created in the system during the installation of the `libssl` package. A change to `libssl` requires the restart of Apache so that the server maps the updated version of the library to its memory. Failing to do so makes the server not get the latest updates or security patches of the library. Adding `Package["libssl"]` to the `subscribe` list of the Apache service fixes this missing notifier bug.

As we will see in Chapter 5, the issues presented above are widespread and affect the reliability of many software deliverables and applications. This motivates the design of a generic approach for easing the adoption of file system-intensive software in practice.

2.4.3 Challenges

Detecting dependency bugs in file system resources as the ones described in the previous section involves three important challenges, which we discuss in detail in the following sections.

2.4.3.1 Capturing the Effects of Tasks on the File System

Identifying problematic tasks that result in mishandling of file system resources due to missing dependency information requires computing the actual effects of tasks on the file system. Using this information, we can then compare the actual behavior of tasks with the expected one as declared by developers in the corresponding scripts.

(a) Overconstrained graph

(b) Precise graph

Figure 2.30: Overconstrained dependency graph computed by an approach modeling build execution as sequence of processes vs. precise graph produced by an approach modeling build execution as a sequence of tasks.

However, precisely capturing how every task updates and affects the system is not straightforward. Adopting static analysis techniques for computing these side-effects is not a realistic solution, mainly because static analysis is unable to reason about all kinds of updates that a particular task might perform on the system at runtime. This is justified by the fact that a task typically involves the execution of arbitrary scripts and programs. For example, *Rehearsal* [Shambaugh et al., 2016] employs static code verification to detect faults in Puppet programs. Nevertheless, it cannot manage real-world cases because it is unable to handle Puppet resources that abstract arbitrary shell commands (through the `exec` Puppet resource). Notably, such `exec` resources are highly pervasive as they appear in the *56%* of the top-1000 most widely used Puppet modules found in Forge [Puppet, 2019b].

Since static analysis cannot help with handling the unpredictable behavior of tasks, using dynamic analysis to monitor task execution sounds a more fruitful approach. However, even in such a case, extracting all the actual runtime effects and mapping them to the task they come from can be challenging. To better illustrate this, consider the following representative example.

Prior work on detecting bugs in Make incremental builds, namely mkcheck [Licker and Rice, 2019], models build execution as a set of system processes created by the build system during execution, e.g., Make spawns a new `gcc` process for compiling every individual source file. This model treats every process as a function that takes an input, and produces an output. The input stands for the set of files that are read by the process, while the output is the set of files written by it. The inputs and outputs of every process are computed by analyzing the system call trace of a build. Through this model, they infer the inter-dependencies among files by considering each output to be dependent on every input file. All dependencies are then transitively propagated using the process hierarchy.

However, modeling execution as a set of system processes is problematic. Specifically, the main assumption made by Make-based tools is that the build system *always* spawns a separate process when proceeding to a new build task. However, this assumption is *no* longer valid in modern build systems (e.g., Gradle, Maven, or Scala's sbt) or configuration management software (e.g., Puppet) where the same system process (e.g., a JVM process, a Ruby process) involves multiple tasks. Tools that model execution as a sequence of processes become ineffective when applied to such systems, as their analysis precision significantly drops.

To highlight how this feature affects the precision of existing work we provide a representative example. Consider a Gradle task `A` that reads a file `A.in` and creates a file `A.out`, and

a Gradle task `B` with file `B.in` as its input, and `B.out` as its output. An approach that works on granularity of processes produces the dependency graph of Figure 2.30a. The main Gradle process runs both tasks A and B; therefore, the analysis considers files `A.in` and `B.in` as the inputs of that process, and files `A.out` and `B.out` as its outputs. Conceptually, this *merges* the two tasks into a single task. The resulting graph is overconstrained [Licker and Rice, 2019], because the analysis *over-approximates* the set of dependencies. For example, when there is a change in the input file `A.in`, the analysis incorrectly considers that both output files (i.e., `A.out` and `B.out`) must be updated, even if `A.in` only affects `A.out`. Overconstrained graphs lead to dozens of false positives and negatives [Licker and Rice, 2019].

Similarly to non Make-based build systems, the granularity of processes is also ineffective for the domain of Puppet, as the same system process may involve different tasks (Puppet resources). In such cases, the existing work [Licker and Rice, 2019] is not able to distinguish which file system resources are affected by which task. Therefore, it is unable to infer the precise inter-relationships among execution steps.

2.4.3.2 Modeling File System Operations

Beyond distinguishing which operations are performed by which tasks, careful modeling is also required for abstracting the updates of the underlying system that take place during the execution of a build or a configuration management script. This is because the execution trace that stems from such scripts is extremely complex. Indeed, build and configuration management scripts involve the application of diverse entities—such as execution of arbitrary scripts, configuration of services, installation of packages, and more—that apart from simply reading and writing files, they perform many operations on the transient Operating System (OS) structures (e.g., file descriptor table, process table, etc.).

For a precise trace analysis (e.g., correctly resolving the absolute file paths that a system call works on), we need to accurately track all the operations performed on these structures. This requires a careful design to deal with the complexities and semantics of the system calls as well as the underlying structure of the file system. For instance, many processes spawned by Puppet share their file descriptor table or working directory with their child processes.[2] Hence, any update to any of those entities performed by one process, also affects the other one. By ignoring this behavior, processes will hold stale information about their file descriptor table and working directory.

As another example, consider the following execution trace:

```
1  100 open("/usr/lib/perl5", O_RDONLY) = 3
2  100 rename("/usr/lib/perl5", "/usr/lib/perl") = 0
3  100 openat(3, "5.26/Socket6.pm", O_RDONLY) = 4
```

To determine the absolute path that the system call `openat` operates on (`/usr/lib/perl/5.26/Socket6.pm`), we have to interpret the file `5.26/Socket6.pm` relative to the directory related to the file descriptor 3. To do so, we need to consider that in

[2]For example, in Linux, this can be achieved through the `CLONE_FD` and `CLONE_FS` flags passed in the `clone` system call.

a Unix-like file system, every file is associated with an `inode` rather than a path. Thus, the file-related OS structures (e.g., file descriptor table) have to refer to `inodes` instead of path names. Although the correct resolution of absolute paths is an important issue, existing approaches ignore the organization of the file system. For example, they either cannot resolve the absolute path of the `openat` system call [van der Burg et al., 2014] (they do not support file descriptors), or they describe files through their paths [Licker and Rice, 2019]. The latter makes the corresponding file descriptor table hold the stale entry (3, `/usr/lib/perl5`), after the `rename` at line 2.

2.4.3.3 Efficiency and Applicability

Another important challenge in finding dependency bugs is efficiency and applicability. In particular, a core limitation of existing work is that it only captures OS-level facts (e.g., file accesses and file dependencies) which are computed while analyzing an execution trace. To verify the inferred file dependencies against the specification of scripts, prior work [Licker and Rice, 2019] triggers incremental builds by touching each source file, and checking whether the expected output files are re-generated in response to the updated input files. This makes the verification task extremely slow as it requires substantial resources when applying multiple incremental builds in large-scale projects [Licker and Rice, 2019] (see also Section 5.3.6).

A critical reader may think that combining static analysis with dynamic analysis is a workaround for this efficiency issue [Bezemer et al., 2017]. Specifically, another approach could perform static analysis on user-defined scripts to extract task specification, and then compare this specification against the actual behavior of task observed at runtime. Nevertheless, reliably extracting task specifications from the corresponding scripts through static analysis is particularly challenging (and in many cases not possible) for multiple reasons [Licker and Rice, 2019]. First, static analysis cannot reason about tasks whose inputs / outputs are dynamically computed and are not known in scripts. The same applies for tasks not explicitly mentioned in the underlying scripts, e.g., consider tasks defined in external Gradle plugins as illustrated in Figure 2.26. Second, static analysis needs to reason about the complex semantics of each system's DSL. This *limits* generalizability, as applying the approach to a new system requires implementing a new static analyzer, which involves a lot of engineering effort. Third, even when a static analyzer is available, OS-level facts (inferred dynamically by the existing model) are not comparable with task specifications (computed statically), when the system does not follow the *target-prerequisites* pattern for expressing custom tasks. This happens in various systems, including Gradle, Scala's sbt, or Puppet. To further clarify this, consider the following example.

```
1 task A {
2   inputs.file "/file/A"
3   // Arbitrary operation...
4 }
5 task B {
6   inputs.files ("/file/A", "/file/B")
7   // Arbitrary operation...
8 }
```

```
1 open("/file/A")
2 ... // other file system operations
3 open("/file/B")
4 ... // other file system operations
5 open("/file/A")
```

In the Gradle script on the left, we have two incremental tasks (task `A` and `B`) performing some arbitrary operations. The specification of the task `A` says that this task is expected to consume the file `/file/A`, while the task `B` reads the files `/file/A` and `/file/B`. Note that the specification only indicates the intent of the developer, and *not* the actual interactions of task with the system. The latter is shown in the execution trace on the right. In this scenario, it is not possible to compare the actual behavior of tasks (inferred by analyzing the execution trace on the right) against build specification (extracted statically from the build script on the left). This is because existing dynamic analysis techniques are unable to map the file accesses shown on the right to the task they belong to (see also Challenge 1, Section 2.4.3.1). This is necessary for deciding correctness. For example, if the first access comes from task `A` while the remaining ones stem from task `B`, the build script is not faulty. On the other hand, if it is the other way around (i.e., the last two accesses belong to the first task), the task `A` manifests a missing input on file `/file/B`, as it consumes a file not mentioned in the build script.

Make adopts the target-prerequisites pattern (i.e., each Make task is uniquely identified by a target file), and this is why it is supported by the existing work. Specifically, it is easy for the existing work to verify the dynamic file dependencies against the static build specification as both are expressed in terms of file paths.

2.5 The Need for Abstractions

After studying the types of bugs examined in this thesis, a key question arises at this point: *How can we systematically identify them through automated means?* Unfortunately, the traditional workflow of automated software testing (Figure 2.1) comes with two important limitations that prevent us from addressing the major challenges associated with the discovery of the studied bugs (Sections 2.4.3, 2.3.3, and 2.2.7), that is, *test oracle specification and reasoning*, and *applicability*.

Test oracle specification and reasoning: As we extensively covered in Section 2.1.2, the test oracle problem [Weyuker, 1982] involves the creation of a proper mechanism that determines whether a particular program behavior is expected or not. The test oracle problem is quite common in systems that manifest complex behavior. Indeed, there is no evident way to say when a particular configuration management and build script manifests a dependency bug, an ORM contains a logic error, or a compiler is full of typing bugs. To do so, we need to devise ways to reason about certain properties of the software under test or its inputs. For example, for detecting typing bugs, we need somehow to tell that the input program is free from type errors

so that we can report cases where the compiler under test miscompiles a well-typed program. For ORM bugs, it is important to construct a reference result set with which we can compare the output of the ORM under test for mismatches. In case of dependency bugs, we have to extract all the static dependencies declared in a given configuration management or build script and compare them with the actual dependencies that stem from the runtime effects on the file system.

Applicability: While devising methods for test oracle specification and reasoning, we also need to preserve an important property: the applicability of our testing methods. This is because we aim to find dependency bugs in scripts written in diverse systems that involve different DSLs and semantics. We target finding bugs in ORM systems that offer dissimilar APIs exposed on top of different programming languages. We want to test type checkers of various compilers. For example, it is important to come up with a technique that reasons about a Puppet script or a Puppet execution, but this technique is still effective in dealing with other domains, such as build scripts.

It becomes clear that it is a requirement to introduce appropriate layers of abstractions that allow us to reason about properties of a system under test and its input in a way that we can effectively apply our testing efforts to systems with dissimilar interfaces, implementations, and semantics.

Chapter 3

Methods

We now start addressing the problem of finding the classes of bugs presented in Chapter 2. Specifically, in this chapter, we introduce approaches for testing software that rely on two forms of abstractions, namely, *execution abstraction*, and *test input abstraction*.

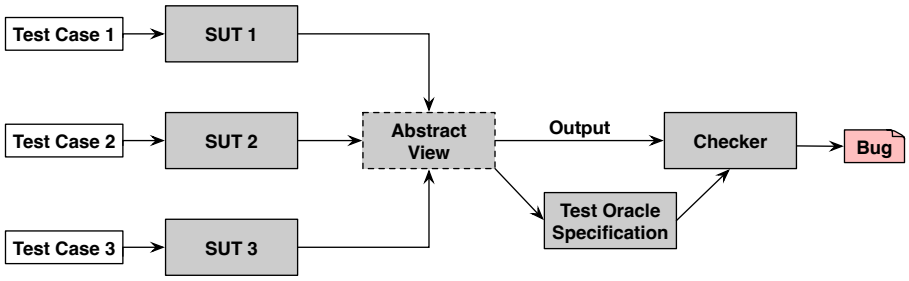
When employing execution abstraction (Figure 3.1a), we extract an abstract view of SUT (System Under Test) from its execution. This abstract view enables (1) reasoning about SUT's behavior and (2) test oracle specification, which in turn entail the presence or absence of bugs in the current execution. Notably, executions of diverse SUTs can be mapped into the same abstract view, thus addressing the applicability issue of testing.

In case of test input abstraction, we abstract the input of SUT. An abstract test input is mapped into multiple concrete test cases, which in turn are passed as input to dissimilar implementations. Reasoning is done on an abstract test case, which helps derive concrete test inputs that manifest certain desired properties. We leverage these properties to extract the test oracle used for determining whether the results of a SUT exhibit an undesired behavior. In other words, we verify that the actual behavior of SUT is compatible with the properties of its test input.

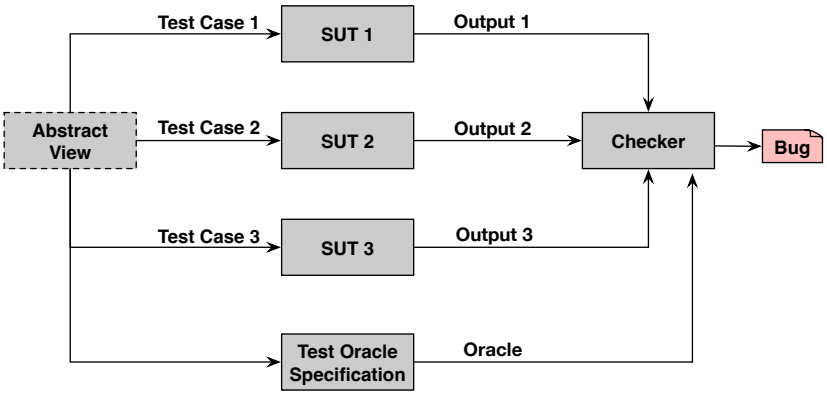
Section 3.1 addresses the problem of finding typing bugs in compilers, and Section 3.2 proceeds with our testing approach for detecting bugs in ORM implementations. We conclude this chapter by introducing our approach for detecting dependency bugs in configuration management and build scripts (Section 3.3). The first two approaches are based on test input abstractions, while the last one relies on an execution abstraction.

3.1 Finding Compiler Typing Bugs

In this section, we present our method for detecting typing bugs in multiple compiler implementations. Our method (shown in Figure 3.2) employs a combination of *random program generation* and *transformation-based* compiler testing, which are guided by a test input abstraction. The first component of our approach is a *program generator* (Section 3.1.2) that comes with three important characteristics. First, it produces *semantically-valid* programs, because typing bugs mainly cause the compiler to reject *well-formed* programs (Challenge 1, Section 2.2.7). Rejecting a well-formed program produced by our generator indicates a potential compiler bug (test



(a) Abstracting SUT's execution



(b) Abstracting SUT's test inputs.

Figure 3.1: Two forms of abstractions for software testing.

oracle). Second, the resulting programs rely heavily on parametric polymorphism (Challenge 4, Section 2.2.7), while we avoid generating complex arithmetics or nested loops, because such features are irrelevant to the types of bugs we aim to detect. Third, to test compilers of different languages (Challenge 6, Section 2.2.7), our generator yields programs at a higher-level of abstraction and then uses translation mechanisms to convert the "abstract" programs into the target language (see test input abstraction, Figure 3.1a). The "abstract" programs are written in an intermediate language (IR), a simple object-oriented language that supports parametric polymorphism and type inference. Finally, our program generator takes as input a configuration that can either disable certain features (e.g. bounded polymorphism), enable them, or affect their probability distribution.

The second component is based on our design of two novel transformation-based methods (Section 3.1.4): *type erasure mutation* and *type overwriting mutation* whose goal is to exercise compilers' type inference algorithms and other type-related operations (Challenge 2, Section 2.2.7). Given an input program P written in the IR, the type erasure mutation removes declared types from variables and parameters, or type arguments from parameterized constructor and method calls, while preserving the well-formedness of P. The type overwriting mutation adopts a fault-injecting approach (Challenge 3, Section 2.2.7), and introduces a type error in P by replacing a type t_1 with another incompatible type t_2. The type overwriting mutation updates the test oracle, as compiling a program obtained from this mutation indicates a potential soundness bug in the compiler under test. To perform sound transformations with respect to

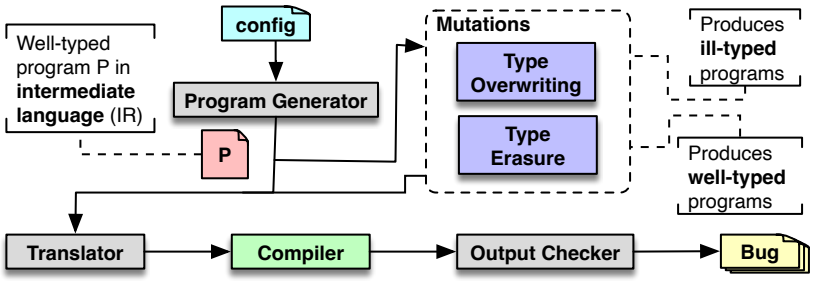


Figure 3.2: Overview of our approach for detecting typing bugs in compilers.

the test oracle, both mutations rely on a model (Section 3.1.3), which is defined based on the semantics of our test input abstraction, that is IR. The model is also accompanied by an intra-procedural type inference analysis (reasoning) that captures (1) the declared and inferred type of each variable, (2) how each type parameter is instantiated, and (3) dependencies among type parameters.

In contrast to previous work [Yang et al., 2011; Livinskii et al., 2020; Zhang et al., 2017; Sun et al., 2016a], which requires differential testing [McKeeman, 1998], our approach does not need to employ it, as each program derived either from the generator or our mutations also acts as an oracle, based on the way it was derived.

Remark on test input abstraction: Our approach for detecting compiler typing bugs uses a dedicated language (IR) for abstracting test inputs (programs). Our IR has the following benefits. First, it eases reasoning by supporting only the features that are promising for triggering compiler typing bugs (see Section 3.1.1 for more details). This in turn helps us focus on generating test inputs that indeed exhibit such type-intensive characteristics (e.g., parametric polymorphism). Second, on top of our IR, we define a model for reasoning about type information. As we will explain in Sections 3.1.3 and 3.1.4, using this model, we construct test inputs containing certain desired properties that act as test oracles. This in turn helps us exercise specific compiler components, and uncover (1) type inference bugs, (2) soundness compiler bugs, and (3) other type-related bugs. Finally, our IR promotes applicability, as we can use the mechanisms and the components of our approach to test more than one compiler. To achieve this, after all reasoning (program generation, mutation), we use a language-specific translator to build a concrete program based on a program written in the IR. In the following, we present the technical details behind each component of our approach.

3.1.1 IR and Preliminary Definitions

Figure 3.3a shows the syntax of the IR. Some advanced features of the IR and its type system, such as use- and declaration-site variance, interfaces, and abstract classes are omitted from the figure for the sake of simplicity. A program in the IR consists of a sequence of declarations. A declaration is either a class, a function, or a variable. The language also supports parameterized declarations by introducing a type parameter in the body of the declaration (see $\Lambda\alpha.d$). The IR contains constant values of a type t ($val(t)$) and the typical expressions encountered in an object-oriented language (e.g., conditionals, parameterized method and constructor calls),

$$
\begin{array}{rcl}
\langle p \in \mathit{Program} \rangle & ::= & \bar{d} \\
\langle d \in \mathit{Decl} \rangle & ::= & \texttt{class}\ \mathcal{C}\ \texttt{extends}\ e\ \bar{d} \\
& & \mid\ \texttt{fun}\ m(x : t \ldots) : t = e \\
& & \mid\ \texttt{var}\ x : t = e \ \mid\ \Lambda\alpha.d \\
\langle e \in \mathit{Expr} \rangle & ::= & \mathit{val}(t) \ \mid\ x \ \mid\ e.f \ \mid\ e \oplus e \ \mid\ \{d \ldots e \ldots\} \\
& & \mid\ (e.m\ t)(e \ldots) \ \mid\ (\texttt{new}\ \mathcal{C}\ t)(e \ldots) \\
& & \mid\ e.x = e \ \mid\ \texttt{if}(e)\ e\ \texttt{else}\ e \ \mid\ e :: m \\
& & \mid\ \lambda x : t.e \\
\langle \oplus \in \mathit{BinaryOp} \rangle & ::= & == \ \mid\ != \ \mid\ \&\& \ \mid\ || \ \mid\ > \ \mid\ \geq \ \mid\ < \ \mid\ \leq \\
\langle x \in \mathit{VariableName} \rangle & ::= & \textit{is the set of variable and field names} \\
\langle m \in \mathit{MethodName} \rangle & ::= & \textit{is the set of method names} \\
\langle \mathcal{C} \in \mathit{ClassName} \rangle & ::= & \textit{is the set of class names}
\end{array}
$$

(a) Syntax of the IR.

$$
\begin{array}{rcl}
\langle t \in \mathit{Type} \rangle & ::= & \top \ \mid\ \bot \ \mid\ \alpha \ \mid\ \mathcal{T} : t \\
& & \mid\ \Lambda\alpha.t \ \mid\ (\Lambda\alpha.t)\ t \\
\langle v\ \alpha \in \mathit{TypeParameter} \rangle & ::= & \phi : t \\
\langle v \in \mathit{Variance} \rangle & ::= & \texttt{co} \ \mid\ \texttt{contra} \ \mid\ \bot \\
\langle \mathcal{T} \in \mathit{TypeName} \rangle & ::= & \textit{is the set of type names}
\end{array}
$$

(b) Types in the IR.

$$
\textsc{top}\ \frac{}{\Gamma \vdash t <: \top} \qquad
\textsc{bottom}\ \frac{}{\Gamma \vdash \bot <: t} \qquad
\textsc{self}\ \frac{}{\Gamma \vdash t <: t} \qquad
\textsc{reg}\ \frac{}{\Gamma \vdash \mathcal{T} : t <: t} \qquad
\textsc{trans}\ \frac{\Gamma \vdash t_1 <: t_2 \qquad \Gamma \vdash t_2 <: t_3}{\Gamma \vdash t_1 <: t_3}
$$

$$
\textsc{covariance}\ \frac{\Gamma \vdash t_1 <: t_2}{\Gamma \vdash (\Lambda\ \texttt{co}\ \phi : t)t_1 <: (\Lambda\ \texttt{co}\ \phi : t)t_2} \qquad
\textsc{contravariance}\ \frac{\Gamma \vdash t_2 <: t_1}{\Gamma \vdash (\Lambda\ \texttt{contra}\ \phi : t)t_1 <: (\Lambda\ \texttt{contra}\ \phi : t)t_2}
$$

(c) Selected subtyping rules.

Figure 3.3: The syntax and the types in the IR.

along with functional features, i.e., method references and lambdas. Our IR supports all interesting features for revealing typing bugs as described in Section 2.2.6. Specifically, it contains all standard object-oriented features (field accesses, constructors, inheritance), parameterized declarations, parameterized types, declaration- and use-site variance, bounded polymorphism, and functional programming constructs. Arithmetic expressions, loops, exceptions, access modifiers (e.g., `public`, `private`) are not supported.

Regarding types (Figure 3.3b), the language involves a nominal type system. A type is either (1) a regular type ($\mathcal{T} : t$) labeled with a name $\mathcal{T}$ and a supertype t, (2) a type parameter ($\phi : t$) with an upper bound t, (3) a type constructor, or (4) a type application that instantiates a type constructor with a set of concrete types. Notably, a type parameter comes with a variance choice (it can be either invariant, covariant or contravariant), something that affects the subtyping rules of type applications (see the [COVARIANCE] and [CONTRAVARIANCE] rules in Figure 3.3c).

In the following definitions, we use the symbol $<:$ to denote the subtyping relation, and the operation $S(t)$ to give the supertype of type t (e.g., $S(\mathcal{T} : t) = t$)).

Definition 3.1.1 *(Type substitution) The substitution $[\alpha \mapsto t] : Type \longrightarrow Type$, where α is a type parameter, and t is a type $t \in Type$, is inductively defined by:*

$$
\begin{aligned}
[\alpha \mapsto t]\alpha &= t \\
[\alpha_1 \mapsto t]\alpha_2' &= \alpha_2 && \alpha_1 \neq \alpha_2' \\
[\alpha \mapsto t_1]\mathcal{T} : t_2 &= \mathcal{T} : [\alpha \mapsto t_1]t_2 \\
[\alpha \mapsto t_1]\Lambda\alpha.t_2 &= [\alpha \mapsto t_1]t_2 = (\Lambda\alpha.t_2)t_1 \\
[\alpha_1 \mapsto t_1](\Lambda\alpha_2.t_2)t_3 &= (\Lambda\alpha_2.t_2)[\alpha_1 \mapsto t_1]t_3
\end{aligned}
$$

A type substitution replaces all occurrences of a type parameter α in a type t_1 with another type t_2. Also, observe that a type application $(\Lambda\alpha.t_1)t_2$ substitutes the type parameter of a type constructor with the applied type t_2 ($[\alpha \mapsto t_2]t_1$]).

Definition 3.1.2 *(Type unification) Type unification $(Type \times Type \longrightarrow S)$ is an operation that takes two types ($t_1, t_2 \in Type$) and returns a type substitution σ so that $\sigma t_1 <: t_2$:*

$$
\begin{aligned}
\mathit{unify}(\alpha, t) &= [\alpha \mapsto t] \\
\mathit{unify}((\Lambda\alpha.t)t_1, (\Lambda\alpha.t)t_2) &= \mathit{unify}([\alpha \mapsto t_1]t, [\alpha \mapsto t_2]t) \\
\mathit{unify}(t_1, t_2) &= \mathit{unify}(t_1, S(t_2)) \;\; t_1 \notin \mathit{TypeParam}
\end{aligned}
$$

Type unification identifies a substitution σ so that the type σt_1 is a subtype of t_2. We explain how we use the above definitions, when detailing our techniques.

3.1.2 Program Generation

We now describe the internals of our program generator used to produce programs written in the IR.

Context: Our program generator maintains a data structure called *context*, which stores all declarations and types in their namespace. Specifically, we use context to store the following entities: class-, method- and variable declarations (i.e., local variables, class fields, and method parameters), as well as type parameters and lambdas. Every time our generator uses a declaration or a type (e.g., initializing an instance of a class), it consults the context to determine which declarations are available in the current scope.

Generating declarations: The entry point of our program generator is the creation of random declarations (i.e., either a class, a method, or a variable) in the top-level scope. The maximum number of these top-level declarations is given as an input. When our generator constructs a declaration, it adds it to the context so that subsequent declarations and expressions can refer to the initial one.

For example, for generating a class c, our algorithm first constructs an optional number of type parameters (in case it chooses to generate a parameterized class). Then, the algorithm may use a previously constructed class as the superclass of c (inheritance). Each class is constructed on-the-fly, meaning that before generating class fields and methods, our generator adds the current class declaration c (even though it is incomplete) to the context so that its enclosing fields and methods are able to refer to c, e.g., `class A { A field; }`. Note that if the currently generated class inherits from another class, our generator may choose to override some inherited methods. Most importantly, when the current class inherits from an abstract class or inherits from an interface, our generator always provides an implementation for each inherited abstract method. Finally, the generation of methods and functions works in a similar fashion.

Generating types: To generate a type, our generator first computes the set of available types in the current scope, and then picks one type at random. This set contains types from three different sources, namely, (1) built-in types (e.g., `Int`, `String`, `Array`) supported by the language under test, (2) types derived from previously generated classes, and (3) type parameters that are available in the current scope. Notably, for obtaining the second and the third source of types, our generator consults the context, while the first set is a constant given as an input to the generator. If the selected type is a type constructor, our generator instantiates it by recursively generating types with which the corresponding type parameters are instantiated (see below).

Instantiating Type Constructors: After selecting a random type t, our generator performs the following check. If t is a type constructor, the generator instantiates it as shown in Algorithm 1. The algorithm expects (1) a set of available types (*types*) from which it randomly picks type arguments for type constructor t, and (2) the input parameter σ corresponding to an (incomplete) type substitution in case we want to instantiate some type parameters with specific types. Note that receiving an empty type substitution denotes that the corresponding type parameters are not restricted to specific instantiations.

When encountering a bounded type parameter α (lines 5–13), Algorithm 1 works as follows. If there is already a type substitution for α and the upper bound b is another type parameter (e.g., `class A<T1, T2: T1>`), we recursively update previous substitutions of b so that the type assigned to α remains compatible with that assigned to b (lines 7–9). In other words, the statement at line 9 creates a substitution for b so that $\sigma\alpha = \sigma b$. Otherwise, if $\sigma\alpha = \alpha$ we

Algorithm 1: Random instantiation of type constructors.

```
1  fun instantiate(t, types, σ)=
2      types ← exclude primitives from types
3      for α ∈ TypeParams(t) do
4          match α with
5              case υ ϕ : b                                  // with an upper bound
6               ⇒
7                  if σα ≠ α then
8                      if b ∈ TypeParam then
9                          σ ← σ ⊔ unify(b, σα)
10                     t′ ← σα
11                 else
12                     t′ ← findSubtype(b, types)
13             case υ ϕ ⇒
14                 if σα = α then t′ ← random(types)
15                 else t′ ← σα
16     if isTypeConstructor(t′) then
17         t′ ← instantiate(t, types, ∅)
18     σ ← σ[α ↦ t′]
19 return σt
```

examine the type pool (i.e., *types*) to find a subtype of b in order to instantiate α with a type that respects its bound (lines 11–12). When α does not involve an upper bound and a type substitution, we simply assign a random type to α. As a final step, when the selected type t' is a type constructor, we recursively instantiate it using an empty type substitution (line 16–17) and finally update the current type substitution by assigning t' to α (line 18). The output of the algorithm is then the application of the resulting type substitution σ to the given type constructor t (line 19).

Generating expressions: To avoid producing ill-typed expressions and programs, our program generator adopts a *type-driven* approach for generating expressions. This means that it first constructs a random type t, and then creates a random expression e of a type t', where $t' <: t$. Generating such expressions helps us exercise the implementation of subtyping rules in the compiler under test. To find a subtype of a given type t (including itself according to the [SELF] rule presented in Figure 3.3c), our generator uses the *findSubtype* procedure as Algorithm 1 (line 12) does. Although there are some interesting challenges in the presence of variance and bounded polymorphism, we omit the technical details of this procedure for simplicity. Now, we discuss some technicalities behind expression generation.

Object initialization: Expression generation is done up to a certain depth provided as input to the generator. However, infinite loops may occur, especially when initializing objects of classes with circular dependencies [Liu et al., 2020]. To prevent this from happening, after reaching the maximum depth, the generator initializes objects with constant values (i.e., $val(t)$), which are typically translated into cast `null` expressions.

Resolving matching methods and fields: When constructing a method call, a method reference, or a field access of a type t, the generator examines the context to resolve existing methods and fields that match the given type t.

Algorithm 2 illustrates the resolution process performed when generating a method call of a type t. When dealing with field accesses and method references, the resolution process works

Algorithm 2: Resolve methods by return type t.

```
1  fun resolveMethod(t, context, n)=
2      methods ← resolveMatchingFunctions(t, context, n)
3      methods ← methods ∪ resolveMatchingObjects(t, context, n)
4      if methods = nil then
5          methods ← resolveMatchingClass(t, context, n)
6      if methods = nil then
7          (rt, method) ← generateMatchingMethod(t, context, n)
8      else (rt, method) ← random(methods)
9      return (rt, method)
10
11 fun resolveMatchingClass(t, context, n)=
12     methods ← []
13     for c ∈ getClasses(context, n) do
14         for m ∈ getMethods(c) do
15             r ← getRetType(m)
16             σ ← unify(r, t)
17             if σr <: t then
18                 rt ← instantiate(toType(c), types, σ)
19                 σ' ← σ ∪ getTypeSubstitution(rt)
20                 if isParameterized(m) then
21                     m ← instantiate(toType(m), types, σ')
22                 methods ← methods ∪ (rt, m)
23     if methods = nil then return []
24     (rt, method) ← random(methods)
25     return generateExpr(rt, context, n), method
```

in a similar manner. Specifically, resolution involves three steps. In the first step, we inspect the current scope to find methods whose return type is either a subtype of t (line 2), or live objects containing at least one instance method whose signature matches t (line 3). Specifically, the call at line 2 finds methods in the current scope that can be invoked without an explicit receiver, while the call at line 3 identifies variables that point to objects with instance methods that match t. These variables are then used for generating a call of the form `x.m(...)`.

If the above search fails (i.e., *methods* = **nil**), we examine all previously declared classes and check whether there is any class containing such a method (line 5). To answer this question we use the `resolveMatchingClass` procedure (lines 11–25). For every class c and method m, our resolution algorithm unifies the return type r of m with type t (line 16), and if $\sigma r <: t$, it instantiates the corresponding receiver type that stems from class c using the (partial) substitution obtained by type unification (line 18). Note that if m is parameterized, our algorithm instantiates it. Finally, the procedure picks a receiver type *rt*, and a method *m* at random (line 24), and generates an expression of type *rt* corresponding to the receiver of method call (line 25).

When the search of `resolveMatchingClass` fails (line 23), `resolveMethod` ultimately produces a fresh method with a return type t, and adds it to the current scope or to an existing class (line 7). Otherwise, it randomly selects a pair of a receiver and a method, which is the output of the algorithm (line 9). Our generator then uses this output to finish the generation of method call accordingly.

Example of resolution: To further illustrate the above resolution process through an example, consider the following code snippet.

```
1 class A<T> {
2   fun <X: T, Y> foo(): B<X, Y> = TODO()
3   fun bar(): B<Double, T> = TODO()
4   fun baz(): B<String, T> = TODO()
5 }
```

Suppose now that we want to generate method call of type $t = \texttt{B} < \texttt{Double}, \texttt{A} < \texttt{String} >>$. Unifying the return type of method `foo` with t yields a type substitution

$$\sigma = [T \mapsto \texttt{Double}, X \mapsto \texttt{Double}, Y \mapsto \texttt{A} < \texttt{String} >]$$

Therefore, instantiating receiver and parameterized method based on this type substitution yields the following method call whose return value is of type t:

```
(new A<Double>()).foo<Double, A<String>>()
```

Similarly, a valid call of `bar` that returns t is:

```
(new A<A<String>>()).bar())
```

Observe though that the method `baz` (line 4) does not match, as there is no valid substitution σ such that $\sigma t' <: t$, where $t' = \texttt{B} < \texttt{String}, \texttt{T} >$.

Note that our resolution process works in a similar manner when dealing with field accesses and method references. Specifically, for field accesses, we omit the call at line 2, while for method references, beyond checking the return type of a method for compatibility with the given type t, we also examine the type of method's formal parameters.

Customizing for individual language features: Our program generator can be easily customized using a configuration given as an input (recall Figure 3.2). This configuration includes probabilities that guide program generation. These probabilities include (but are not limited to): (1) a probability of selecting a type instead of another one (e.g., favoring use of type parameters over regular types), (2) a probability of adding an upper bound to a type parameter (bounded polymorphism), (3) a probability of generating a particular expression (e.g., method reference). Setting a probability to zero allows us to disable the corresponding feature. The configuration also contains some bounds regarding the size of classes, methods, and expressions. For example, the user can specify the maximum number of class fields / methods, the maximum depth of expressions, and more.

Interestingly, disabling and enabling individual features or generating programs with different characteristics can be promising when combined with a sort of swarm testing [Groce et al., 2012].

Translating programs to the target language: Translation is a straightforward process, which is done in a bottom-up manner. Each individual expression is converted into its counterpart in the language under test, and then translation incrementally constructs the parent declarations. In languages not supporting global variables and functions, such as Java and Groovy, the corresponding translators convert the top-level variables and functions in the IR in static fields and methods included in a class named `Top`. Then, references to these variables and

functions are prefixed with `Top.x`. Moreover, conditional expressions are transformed to expressions that use the ternary operator (i.e., "?"). Finally, all *val(t)* expressions in the IR (recall Figure 3.3a) are translated into `(t) null` in Groovy and Java, and `TODO() as t` in Kotlin.

3.1.3 Modeling Type Information

We introduce a model for reasoning about type information in a program written in the IR. The model is based on the notion of a *type graph* (Sections 3.1.3.1), a program representation that captures how type information flows between declarations and type parameters. We present an intra-procedural type inference analysis for building type graphs (Section 3.1.3.2). Finally, based on type graph, we introduce the properties of *type preservation* and *type relevance* (Section 3.1.3.3) which, as we will show in Section 3.1.4, our testing approaches are based on.

3.1.3.1 Type Graph Formulation

The type graph captures (1) the declared and inferred type of program declarations, (2) how each type parameter is instantiated, and (3) the inter-dependencies between type parameters. We define a type graph as $G = (V, E)$. There are nodes of two kinds: a node $n \in V$ is either a declaration $d \in \textit{Decl}$, or a type $t \in \textit{Type}$. The set of edges $E \subseteq V \times V \times L$, where $L = \{\texttt{decl}, \texttt{inf}, \texttt{def}\}$ indicate the following: given a type graph G, the edge $n \xrightarrow{\texttt{decl}} t$ denotes that the type of node n is explicitly declared in the program as t. For example, for a variable declaration of the form `String x = ...`, there is a `x` $\xrightarrow{\texttt{decl}}$ t edge, where $t =$ `String`. The edge $n_1 \xrightarrow{\texttt{inf}} n_2$ indicates that the type of node n_1 is inferred by node n_2. For example, for an assignment of the form `String x = "str"`, beyond a $\xrightarrow{\texttt{decl}}$ edge, there is also an `x` $\xrightarrow{\texttt{inf}}$ t edge, where $t =$ `String`. This is because the type of variable `x` is inferred as `String`, which is the type of the constant at the right-hand side of the assignment. Finally, the edge $t_1 \xrightarrow{\texttt{def}} t_2$ shows that type t_1 consists of another type t_2. For example, for each type application of the form $t_1 =$ `A<String>`, we have the edges $t_1 \xrightarrow{\texttt{def}}$ `T` and `T` $\xrightarrow{\texttt{decl}} t_2$, where $t_2 =$ `String`. These edges indicate that parameterized type `A<String>` contains type parameter `T`, and the corresponding type argument is `String`.

3.1.3.2 Constructing Type Graphs

To construct type graphs, we design an intra-procedural, flow-sensitive analysis that operates on programs written in the IR. The analysis $A(G, n)$ constructs a type graph G by visiting every declaration and expression n of the given program. Figure 3.4 summarizes our analysis rules. For what follows, *unify′* is a variant of type unification that adds the following rules:

$$\textit{unify}'((\Lambda\alpha.t)t_1, (\Lambda\alpha)t_2) = [\alpha \mapsto \alpha] \;\text{ if } t_1 = t_2$$
$$\textit{unify}'((\Lambda\alpha_1.t_1)t_2, (\Lambda\alpha_2.t_3)t_4) = [\alpha_2 \mapsto \alpha_1]$$
$$\text{if } \textit{unify}(S(t_3), [\alpha_1 \mapsto t_1]t_2) \neq \emptyset \wedge [\alpha_2 \mapsto t_4]t_3 <: [\alpha_1 \mapsto t_2]t_1$$

TYPE APPLICATION

$$\frac{t = (\Lambda\alpha.t_1)t_2}{A(G,t) \Rightarrow G \cup \{(\Lambda\alpha.t_1)t_2 \overset{\text{def}}{\rightarrow} \alpha, \quad \alpha \overset{\text{decl}}{\rightarrow} t_2\}}$$

VAR DECL

$$\frac{e = \texttt{val}\ \ x : t_1 = e \qquad t_2 = \textit{getType(e)}}{A(G,e) \Rightarrow G' \cup \{x \overset{\text{decl}}{\rightarrow} t_1, \quad x \overset{\text{inf}}{\rightarrow} t_2\}}$$

VAR PARAM CONSTRUCTOR

$$\frac{\begin{array}{c} e = \texttt{val}\ \ x : t_1 = \texttt{new}\ \ (\mathcal{C}t_2)() \\ \Lambda.\alpha.t = \textit{toType}(\mathcal{C}) \qquad \sigma = \textit{unify}'(t_1, (\Lambda.\alpha.t)t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{x \overset{\text{decl}}{\rightarrow} t_1,\ \ x \overset{\text{inf}}{\rightarrow} (\Lambda\alpha.t)t_2\} \cup \{\alpha \overset{\text{inf}}{\rightarrow} \sigma(\alpha) | \alpha \in \sigma\}}$$

VAR PARAM METHOD CALL

$$\frac{\begin{array}{c} e = \texttt{val}\ \ x : t_1 = e_1.(mt)() \\ t_2 = \textit{getRetType}(e_1, m) \qquad \textit{typeParam(m)} \in t_2 \qquad \sigma = \textit{unify}'(t_1, t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{x \overset{\text{decl}}{\rightarrow} t_1,\ x \overset{\text{inf}}{\rightarrow} \textit{getType(e)}\} \cup \{\alpha \overset{\text{inf}}{\rightarrow} \sigma(\alpha) | \alpha \in \sigma\}}$$

PARAM CALL

$$\frac{\begin{array}{c} e = e_1.(mt)(e_2) \\ t_1 = \textit{getType}(e_2) \qquad t_2 = \textit{getParamType}(e_1, m) \qquad \textit{typeParam(m)} \in t_2 \qquad \sigma = \textit{unify}'(t_1, t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{\alpha \overset{\text{inf}}{\rightarrow} \sigma(\alpha) | \alpha \in \sigma\}}$$

Figure 3.4: Analysis rules for building type graphs.

In essence, this modification finds dependent type parameters between two type applications. For example, suppose we have (1) two parameterized classes: `class A<T>` and `class B<T> extends A<T>`, and (2) two type applications derived from these classes, namely, t_1 = `A<String>` and t_2 = `B<String>`. In this scenario, $\textit{unify}'(t_1, t_2)$ returns $[\alpha_2 \mapsto \alpha_1]$, where α_1 and α_2 are the type parameters of type constructors `A` and `B` respectively. This dependency information indicates that instantiating the type parameter α_2 with a type t also instantiates α_1 with the same type, as α_2 flows to the type parameter of superclass.

[TYPE APLICATION]: For each type application $t = (\Lambda\alpha.t_1)t_2$, the resulting type graph contains two edges. The first edge ($\overset{\text{def}}{\rightarrow}$) connects type application with the underlying type parameter α, and the second edge ($\overset{\text{decl}}{\rightarrow}$) connects α with type argument t_2.

[VAR DECL]: For a variable declaration, we connect variable `x` with two types: t_1 is the declared type of `x`, and t_2 is the type inferred by the right-hand side of declaration.

[VAR PARAM CONSTRUCTOR]: For a variable initialized by a parameterized constructor (e.g., `A<String> x = new A<String>()`), beyond adding the edges for the declared and inferred type of variable `x`, we unify the type of the right-hand side with that of the left-hand side to identify any dependent type parameters. If this is the case, we add the corresponding $\overset{\text{inf}}{\rightarrow}$ edges. This rule models the case where the type parameter of a constructor invocation is instantiated by using type information from the variable's declared type (e.g., `A<String> x = new A<>()`).

[VAR PARAM METHOD CALL]: When initializing a variable with the value of a parameterized method call, the type graph contains an $\overset{\text{inf}}{\rightarrow}$ edge, which connects the method's type parameter (in case it appears in the method's return type) with any dependent type or type parameter included in the declared type of `x`. This edge captures the case where a method's type parameter is instantiated based on the declared type of the target variable `x`.

```
1  open class A<T>
2  class B<T>(
3    val f: A<T>
4  ): A<T>()
5
6  fun m(): A<String> {
7    return B<String>(
8      A<String>()
9    )
10 }
```

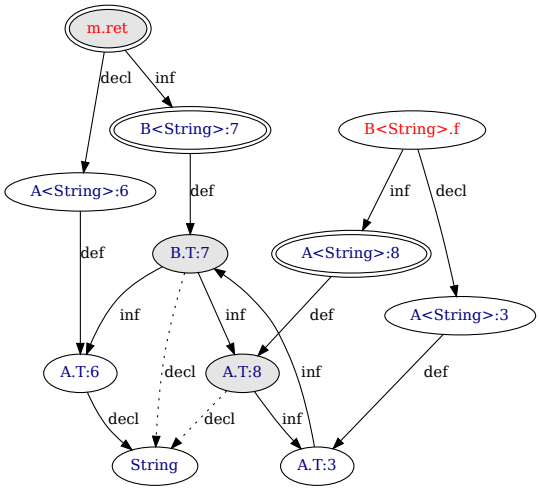


Figure 3.5: A Kotlin program and its type graph. Red nodes represent declarations and blue nodes are types. Types are annotated the line they come from. Double circled nodes are candidate nodes for the type erasure mutation, and shadowed nodes are candidate nodes for the type overwriting mutation.

[PARAM METHOD CALL]: When calling a parameterized method with arguments, the method's type parameter included in the type of the formal parameter is instantiated by the type of the expression e_2 passed as a call argument.

We treat any other case using one of the rules above. For example, a return value of a method is treated as the initial value of a virtual variable named `ret`. We model this using one of the [VAR .*] rules depending on the body of the method. Invoking a parameterized constructor with call arguments (i.e., `A<String>("f")`) is modeled as calling a parameterized method (i.e., [PARAM METHOD CALL]).

Example Type Graph: Figure 3.5 shows an example program and its type graph. The program consists of two parameterized classes with a parent-child relationship (lines 1–4), and a function `m` that returns a value of type `A<String>`. The produced type graph contains two declarations depicted with red color. The one declaration stands for the return value of function `m`, and the other corresponds to the field `f`, after initializing its receiver object at line 7. Observe the dependencies between type parameters. For example, the edge from node `B.T:7` to node `A.T:3` demonstrates that the type parameter of the parameterized constructor call on line 7 is instantiated by the type parameter coming from the call argument at line 8. This edge is captured by the [PARAM METHOD CALL] rule.

3.1.3.3 Type Preservation and Type Relevance

Assume that $\sqcup$ is the least upper bound operator, and *visitedTypes(G, n)* returns all *type nodes* in G that are reachable from the given node n through either $\xrightarrow{\text{decl}}$ or $\xrightarrow{\text{inf}}$ edges.

Definition 3.1.3 *(Type inference) Type inference ($G \times V \longrightarrow Type$) is an operation that takes a*

type graph $G = (V, E)$ *and a node* $n \in V$*, and returns a type. It is defined as:*

$$\mathit{infer}(G, n) = \bigsqcup_{t \in \mathit{visitedTypes}(G,\ n)} t$$

This definition gives the type of a particular node n. This type stands for the least upper bound of all types that are reachable from n.

Definition 3.1.4 *(Type erasure) Type erasure* ($G \times V \longrightarrow G$) *operates on a type graph* $G = (V, E)$ *and a node* $n \in V$ *and returns a new type graph. It is defined as:*

$$\begin{aligned}
\mathit{erasure}(G, \alpha) &= G \setminus \{\alpha \overset{\text{decl}}{\rightarrow} n\},\ \ n \in G \\
\mathit{erasure}(G, (\Lambda.\alpha.t_1)t_2) &= \mathit{erasure}(G, \alpha) \\
\mathit{erasure}(G, t) &= G \ \ \mathit{if}\, t \notin \mathit{TypeParam} \\
\mathit{erasure}(G, d) &= (G \setminus \{d \overset{\text{decl}}{\rightarrow} t\}) \cap \mathit{erasure}(G, t),\ t \in G
\end{aligned}$$

Type erasure takes a node n and a graph G, and removes all $\overset{\text{decl}}{\rightarrow}$ edges associated with the given node n. Conceptually, type erasure removes variables' declared types, or types used as type arguments from the corresponding type parameters.

Based on the *infer* and *erasure* operations, we now present the *type preservation* and *type relevance* properties.

Definition 3.1.5 *(Type preservation) Given a type graph* $G = (V, E)$*, and a node* $n \in V$*, we say that* n preserves *type* $t \in \mathit{Type}$*, when* $\mathit{infer}(G, n) = t$*,* $G' = \mathit{erasure}(G, n)$ *and* $\mathit{infer}(G', n) = t$*.*

Definition 3.1.5 says that a node n preserves its type t, when even after erasing type information from n (e.g., a variable's declared type), the inferred type for n is still t. We can generalize the type preservation property for multiple nodes.

Definition 3.1.6 *(Generalized type preservation) Given a type graph* $G = (V, E)$ *and nodes* $n_1, n_2 \ldots n_k \in V$*, we say that these nodes* preserve *their type when* $t_1 = \mathit{infer}(G, n_1), \ldots, t_k = \mathit{infer}(G, n_k)$*,* $G' = \bigcap_{i=0}^{k} \mathit{erasure}(G, n_i)$ *and* $t_1 = \mathit{infer}(G', n_1), \ldots, t_k = \mathit{infer}(G', n_k)$*.*

Definition 3.1.7 *(Type relevance) Given a type graph* $G = (V, E)$ *and a node* $n \in V$*, we say that* n *is relevant to type* $t \in \mathit{Type}$*, when* $G' = \mathit{erasure}(G, n)$ *and* $\mathit{infer}(G', n) <: t$*.*

The definition above says that a type t is relevant to a node n, when, after performing type erasure, t is a supertype of the inferred type of n.

3.1.4 Mutations

We now introduce our novel testing approaches for detecting inference and soundness bugs. The input of both approaches is a program produced by the generator. Our techniques mutate the input program by leveraging the type preservation and type relevance properties presented earlier.

Algorithm 3: Algorithm for type erasure mutation.

```
1  fun typeErasureMutation(P)=
2      for m ∈ Methods(P) do
3          G ← A(∅, m)                                   // Builds type graph
4          nodes ← findCandidateNodes(G)
5          nodes ← {n ∈ nodes | n preserves its type on G}
6          for k = len(nodes) to 1 do
7              for ⟨n_1, n_2, ..., n_k⟩ ∈ combination(nodes, k) do
8                  if ⟨n_1, n_2, ..., n_k⟩ preserve their type on G then
9                      erase ⟨n_1, n_2, ..., n_k⟩
10                     break
```

3.1.4.1 Type Erasure Mutation

The insight of the *type erasure mutation* (hereafter TEM) is that omitting types (wherever is possible) exercises the implementation of compilers' type inference algorithms. Given an input program P, TEM removes type information from P in a way that this modification does *not* change the semantics of P. Our IR supports type removal for the following cases: (1) removing a variable's declared type (e.g., `var x = 1`), (2) removing type arguments from a parameterized constructor or method call (e.g., `new A<>("")`), (3) removing a return type from a method's signature, (e.g., `fun m() = "f"`), and (4) removing a type from a parameter of a lambda (e.g., `(x) -> x + 1`).

Removing types is not always benign, as it may lead to cases where type inference is impossible or the compiler infers a different type from the one initially declared, something that may cause type errors. For example, consider the following code snippet:

```
1  class A<T>(val f: T)
2  val x: Any = "str"
3  val y: A<Any> = A<Any>(x)
```

Removing the declared type of variable `x` (i.e., `val x = "str"`), and the type argument of the constructor call (i.e., `val y: A<Any> = A(x)`) makes the program ill-typed. This is because the compiler now infers the type of `x` as `String` and the type of the right-hand side of line 3 as `A<String>`. Therefore, there is a type mismatch while type-checking line 3, as we assign something of type `A<String>` to a variable of type `A<Any>`.

To prevent such situations from happening, we need to identify which types and which combinations of them can be safely disregarded. To answer this question, TEM leverages the type graph of the input program. In particular, TEM chooses to erase the types of nodes for which the type preservation property (Definition 3.1.6) holds.

Algorithm 3 summarizes the implementation of TEM, which we describe using the example program and the type graph shown in Figure 3.5. The algorithm takes an input program P, and for every method in P, TEM builds the corresponding type graph (lines 2, 3). On line 4, the algorithm examines the type graph to identify candidate nodes where type erasure is permissible (recall the four cases enumerated in the beginning of Section 3.1.4.1). In the example of Figure 3.5, there are three candidate nodes shown with double circles. Next, TEM excludes every candidate node that does not preserve its type based on Definition 3.1.5 (line 5). In our

example, TEM filters out node `m.ret`, as after type erasure the return type of method `m` becomes `B<String>`. For the remaining set of nodes, our algorithm finds the *maximal* set of nodes that is omittable, meaning that the generalized type preservation property holds for the included nodes (lines 6–9). The intuition is that removing the maximal set of types allows us to explore more paths in the compiler, as there is much hidden type information that the compiler needs to infer.

Back to our example, TEM checks whether the combination of nodes `B<String>:7` and `A<String>:8` can be erased. Indeed, this is the case, as after type erasure both `B.T:7` and `A.T:8` are still instantiated with type `String` (observe that type node `String` is reachable from both nodes, using the graph produced by the *erasure* operation, where dotted edges are removed). As a result, TEM mutates the program accordingly, namely, it transforms the body of method `m` from `return B<String>(A<String>())` to `return B(A())`:

```
1  open class A<T>
2  class B<T>(
3    val f: A<T>
4  ): A<T>()
5
6  fun m(): A<String> {
7    return B(A()) // The type arguments of B and A have been removed
8  }
```

Remarks: By construction, TEM yields well-typed programs. Based on the type preservation property (Definitions 3.1.5 and 3.1.6), TEM considers only types for which it knows that their removal does not affect the typing of declarations and type parameters. An enhanced version of TEM could erase types that although they change the typing of declarations, the resulting program remains well-typed. For example, consider the listing presented previously. Although removing the type of variable `x` alters the type of `x` from `Any` to `String`, it does break the program's well-formedness. To permit such modifications, TEM would require some of sort of def-use analysis to identify "benign" uses of variables.

TEM has a worst case exponential complexity, as it enumerates the combinations of candidate nodes of any size k (Algorithm 3, lines 6–7). However, such an exponential behavior does cause performance problems in practice, because (1) our algorithm disregards any candidate node that is trivially non-omittable (i.e., the node does not preserve its type on its own), (2) our algorithm stops the enumeration when it finds the maximal combination of nodes.

3.1.4.2 Type Overwriting Mutation

The goal of the *type overwriting mutation* (hereafter TOM) is to find soundness compiler bugs. To trigger such bugs, TOM provides the compiler under test with wrongly-typed programs. Specifically, TOM takes a well-typed program as input and mutates it by injecting a program error. Accepting and compiling such an invalid program indicates a potential soundness bug in the compiler. During the semantic analysis phase, the compiler detects program errors of different kinds including type mismatches, circular inheritance problems, use of uninitialized variables, and more. Among these types of errors, TOM focuses on type errors. In particular,

Algorithm 4: Algorithm for type overwriting mutation.

```
1 fun typeOverwritingMutation(P)=
2   m ← random(methods(P))
3   G ← A(∅, m)                                   // Builds type graph
4   nodes ← findCandidateNodes(G)
5   n ← random(nodes)
6   t ← gen a type t such that n is not relevant to t on G
7   replace the decl type of n with t
```

TOM introduces a type error in the input program by replacing a type t_1 with another type t_2 so that type mismatches arise. TOM performs these types of replacements on either the declared types of variables, or upper bounds and type arguments of type parameters and type constructors.

Algorithm 4 illustrates the algorithm of TOM. The algorithm of TOM is similar to Algorithm 3. It starts by randomly picking one method from P and producing its type graph G. (lines 2–3). As in the case of the TEM algorithm, TOM examines G to identify nodes where the mutation is applicable. In the context of TOM, such nodes reflect either variable declarations or type parameters. Next, TOM selects a node n at random, and exploits the type relevance property (Definition 3.1.7) to generate a type t so that the selected node n is *not* relevant to type t. Rather than creating an incompatible type from scratch (i.e., creating `class A {}`), our algorithm generates t at random using the available types at the current scope. In this way, the compiler compares types with diverse shapes and characteristics, which in turn, triggers more subtyping checks and type-related operations in the compiler codebase. After generating such a type, TOM substitutes the declared type of n, with the newly created type t. When n is a type parameter, this replacement occurs in the type parameter's upper bound or explicit type argument.

Consider again the example in Figure 3.5, and suppose that among candidate nodes (shadowed nodes), TOM chooses to mutate node `A.T:8`. TOM generates a random type t (e.g., type `Int`) such that the type relevance property does not hold for the selected node (i.e., *infer(G, n)* = `String` /<:`Int`). The output of TOM is then an updated program where the body of `m` is `return B<String>(A<Int>())` as shown below:

```
1  open class A<T>
2  class B<T>(
3    val f: A<T>
4  ): A<T>()
5
6  fun m(): A<String> {
7    return B<String>(
8      A<Int>() // A<Int> replaces A<String>
9    )
10 }
```

We expect the compiler to reject the mutated program by raising a diagnostic message of the form: *"type mismatch: inferred type is A<Int> but A<String> was expected"*.

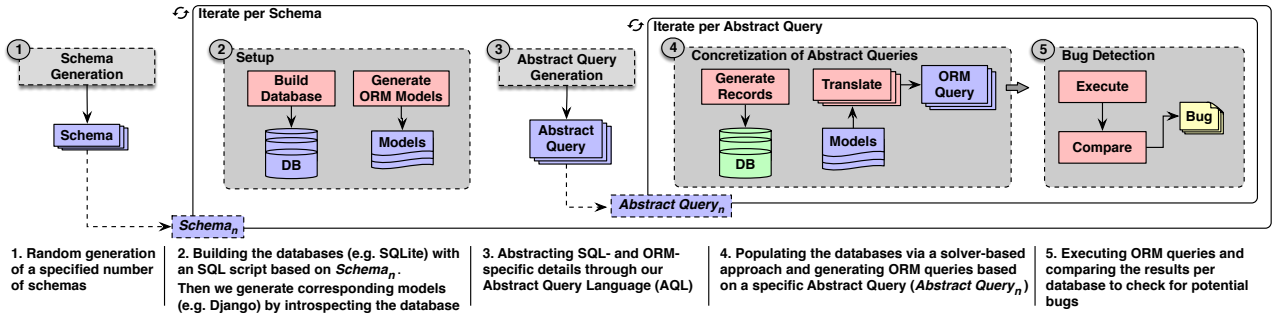


Figure 3.6: Overview of our data-oriented approach for automatically validating ORM implementations through differential testing.

3.2 Data-Oriented Differential Testing of ORM Systems

In this section, we present our approach for finding bugs in ORM implementations, which tackle the challenges presented in Section 2.3.3. In particular, as shown in Figure 3.6, the proposed approach for testing ORMs relies on two test input abstractions and is fully automated. It takes as input the ORM systems to test, and the DBMSs where the ORM queries will run. *Schema Generation* is an initial phase where we generate a number of *abstract* relational database schemas (first test input abstraction, Figure 3.1a). During the *Setup* phase, we build the different databases—one for each provided DBMS—with respect to the schema generated during the first step. Then, we proceed to the *Abstract Query Generation* phase, which involves the generation of queries written in the *Abstract Query Language (AQL)* (second test input abstraction, Figure 3.1a). We design this language to abstract ORM- and SQL-specific details and provide a common reference for testing ORMs, thus addressing "Challenge 1". By design, AQL queries never lead to ambiguous ORM queries ("Challenge 2"). However, AQL queries may be unordered. In this case, we interpret query results as a set of rows rather than a sequence (see also Section 3.2.4). In the *Concretization of Abstract Query* phase we use ORM-specific translators to translate each query into a concrete one. To deal with "Challenge 4" and minimize the number of cases where ORMs produce empty results, we synthesize database records using a solver-based approach (reasoning). In the last step, that is *Bug Detection*, we execute the ORM queries on diverse DBMSs, and compare their results via differential testing. A mismatch in the outputs indicates a potential bug in at least one ORM. Notably, testing the ORM code across different DBMSs enables us to find DBMS-dependent bugs ("Challenge 3").

Remark on test input abstraction: Our data-oriented approach for detecting ORM bugs defines two test input abstractions. The first one is used to abstract a relational database schema. This helps us set up multiple concrete databases and run the resulting ORM queries on multiple DBMSs. The second abstraction of our approach is our abstract query language: AQL. AQL abstracts away SQL-specific details and guides the generation of queries that exhibit complex chains of ORM API calls. Furthermore, building upon AQL, we have designed a mechanism for encoding the constraints of an AQL query into Satisfiability Modulo Theories (SMT) formulae. Then, using a theorem prover, we synthesize database records by deriving a model from the given SMT formulae. As we will see in Section 5.2.4, these targeted database records signifi-

$$
\begin{array}{rcl}
\langle s \in \mathit{Schema} \rangle & ::= & \texttt{schema}\ w : n \\
\langle n \in \mathit{Table} \rangle & ::= & \texttt{table}\ t : m \mid n;n \\
\langle m \in \mathit{Column} \rangle & ::= & c : \tau \mid m;m \\
\langle \tau \in \mathit{ColumnType} \rangle & ::= & \texttt{serial} \mid \texttt{integer} \mid \texttt{real} \\
 & & \mid \texttt{string} \mid \texttt{bool} \mid \texttt{foreign}\ t \\
\langle w \in \mathit{SchemaName} \rangle & ::= & \textit{is the set of names for schemas} \\
\langle t \in \mathit{TableName} \rangle & ::= & \textit{is the set of names for tables} \\
\langle c \in \mathit{ColumnName} \rangle & ::= & \textit{is the set of names for columns}
\end{array}
$$

Figure 3.7: The syntax for representing schemas.

```
1  schema "s1":
2    table "belief":
3      "id": serial;
4      "yank": integer;
5      "rigidness": string;
6
7    table "reported":
8      "id": serial;
9      "greatly": bool;
10     "belief_id": foreign "belief";
```

Figure 3.8: An example schema generated by our approach.

cantly improve the effectiveness of our differential-testing-based oracle, as they help identify subtle ORM bugs. Finally, AQL enables the applicability of our data-oriented approach. Testing a new ORM system simply requires the implementation of a translator that converts an AQL query into a concrete ORM query.

In the following sections, we describe every step of our data-oriented approach in detail.

3.2.1 Schema Generation & Setup

We generate a number of abstract schemas that capture the structure of the databases on which each ORM under test operates. Schemas are represented using the grammar shown in Figure 3.7. Each schema s is a collection of tables and their associated columns. Each column has a type that can be a `serial` (primary key of the table), a number (i.e., `integer` or `real`), a string, or `foreign` t which indicates a table's relationship with another table t of the schema. We omit schema details such as indexes, views or column constraints (e.g., `unique`), as these constructs do not affect the querying and translation mechanisms of ORMs, and therefore, are beyond the scope of this thesis.

Our method randomly generates a user-defined number of schemas. Each schema contains *at least* y tables, and each table includes *at least* k columns. The y and k constants are given as inputs to our approach. For each table, the schema generation algorithm creates a `serial` column named "id" that stands for the primary key of the table, to guarantee that each record in the table is unique and that there is no ambiguity in the data inserted into the table. The remain-

ing fields of the table are randomly generated (optionally based on a deterministic procedure). Figure 3.8 shows an example schema consisting of two tables.

To set up and instantiate the respective DBMSs and ORMs, we use the schemas generated in the schema generation step, This setup involves two parts. First, for every provided DBMS (e.g., SQLite or MySQL) we use each schema s generated earlier, to build the corresponding databases. Second, for each ORM under test, we create the appropriate model classes that stand for the object-oriented representation of each input schema.

To setup the provided DBMSs, we automatically construct an SQL script containing all `CREATE TABLE` statements for creating the tables defined in a provided schema along with their columns. Then, we automatically generate the models for each ORM under test by examining the structure of the newly-created databases. To this end, we leverage tools used to ease ORM porting to existing databases. These tools make a connection to an existing DBMS, introspect its structure, and automatically construct the respective ORM model classes. An example of such tool is the command `manage.py inspectdb` found in the Django project [Django Software Foundation, 2020b].

3.2.2 Abstract Query Generation

Following the *Schema Generation & Setup* phases, we start a testing session for each individual schema. A testing session involves the generation of multiple valid queries (with respect to the provided schema) that are likely to reveal bugs in the ORMs under test. These queries are represented in the *Abstract Query Language (AQL)*, which is close to the APIs and the functionality of ORMs, and provides a wide range of operations (through a functional notation) that are commonly supported by the querying mechanism of ORMs. AQL operations include filtering, sorting, aggregate functions, creation of compound expressions, field aliasing, or union and intersection of queries. By contrast, raw SQL dialect is too low-level and many ORMs are not aware of SQL constructs. Also, the SQL language is not rich enough to express and capture the different API calls of ORMs. For example, the same SQL query can be produced by calling different combinations of ORM's API methods. Since our focus is on detecting bugs in ORMs by exercising different combinations of their API calls, we design AQL.

3.2.2.1 Abstract Query Language

Figure 3.9 shows the syntax of AQL. A query in AQL is the evaluation of a query set (`eval` *qs*). Conceptually, a query set evaluates to a set or to a sequence of records (in case the query set is ordered). Operations such as indexing or slicing, can be applied to the result of a query set, while AQL also supports folding. The function `fold` aggregates the result of a query set into labeled scalar values by applying one or more aggregate functions (see $\alpha \in$ *AggrFunc*).

The simplest form of a query set is `new` t, which creates a new query set from the specified table t. When this query is evaluated, it returns all records of the table t. For example, `new` t is equivalent of `SELECT * FROM t`. Then, various operations can be applied to a query set through the `apply` construct. In particular, AQL provides the `filter` p function that returns all records of the query set that satisfy the given predicate p. The `map` function is used to

$$
\begin{array}{rcl}
\langle q \in \textit{Query} \rangle & ::= & \texttt{eval}\ qs \mid qs[i] \mid qs[i:i] \mid \texttt{fold}\ \{\, (l : \alpha\ e)^{+}\ \}\ qs \\
\langle qs \in \textit{QuerySet} \rangle & ::= & \texttt{new}\ t \mid \texttt{apply}\ \lambda\ qs \mid qs \cup qs \\
 & & \mid qs \cap qs \\
\langle \lambda \in \textit{Func} \rangle & ::= & \texttt{filter}\ p \mid \texttt{map}\ d \mid \texttt{unique}\ \phi \\
 & & \mid \texttt{sort}\ (\phi\ \texttt{asc}) \mid \texttt{sort}\ (\phi\ \texttt{desc}) \\
\langle d \in \textit{FieldDecl} \rangle & ::= & l : e \mid \texttt{hidden}\ l : e \mid d; d \\
\langle p \in \textit{Pred} \rangle & ::= & \phi \oplus e \mid p \wedge p \mid p \vee p \mid \neg p \\
\langle e \in \textit{Expr} \rangle & ::= & c \mid \phi \mid \alpha\ e \mid e + e \mid e - e \mid e * e \mid e / e \\
\langle \phi \in \textit{Field} \rangle & ::= & t.c \mid l \mid \phi.c \\
\langle \alpha \in \textit{AggrFunc} \rangle & ::= & \texttt{count} \mid \texttt{sum} \mid \texttt{avg} \mid \texttt{max} \mid \texttt{min} \\
\langle \oplus \in \textit{BinaryOp} \rangle & ::= & = \mid > \mid \geq \mid < \mid \leq \\
 & & \mid \texttt{contains} \mid \texttt{startswith} \mid \texttt{endswith}
\end{array}
$$

Figure 3.9: The syntax of the Abstract Query Language (AQL).

```
1  apply (filter "addCol" > 5
2    apply (map "addCol": t1.colA + t1.t2.colB
3      new t1)
4  )
5
6  SELECT t2.colA + t2.colB AS "addCol"
7  FROM t1 as "t1"
8  JOIN t2 AS "t2" ON (t1.t2_id = t2.id)
9  WHERE (t2.colA + t2.colB > 5)
```

Figure 3.10: Example AQL query and its equivalent SQL query.

create new compound fields using existing fields found in the given query set. Specifically, `map` expects a sequence of field declarations of the form $l : e$. This declaration creates a new field in the current query set by binding the expression e to the label l. Optionally, a field can be marked as hidden meaning that it is not part of the query set, but it is used for creating other fields (hidden fields are similar to temporary variables). The function `sort`, sorts the provided query set according to the field ϕ in an ascending or a descending order, while the `unique` primitive removes duplicate records with respect to the provided field ϕ. Finally, AQL supports the combination of two query sets through the union and intersection operations

A predicate consists of comparison operators (i.e., $\phi \oplus e$) which are used to compare the value of a field ϕ with the result of an expression e. A predicate may also contain the usual logical operators. An expression can be a constant c (e.g., a number), a field reference ϕ, an application of an aggregate function, or an compound expression derived from the usual arithmetic operators. Finally, a field $\phi \in \textit{Field}$ may be a reference to a column of a table (i.e., $t.c$), a label l created by the `map` function, or a reference to a column of a table's relationship (e.g., $t_1.t_2.c$).

Figure 3.10 shows an example query written in AQL (lines 1–4) and its equivalent query written in SQL (lines 6–9). In this AQL query, we apply two functions. First, we apply `map` to the query set given by `new t1` (lines 2–3) in order to create a new field named `"addCol"`

given by the addition between the `t1.colA` and `t1.t2.colB` columns. Notice that since the latter column refers to a column of the table `t2`, which has a relationship with the original table `t1`, in SQL this is interpreted as a `JOIN` between `t1` and `t2` (line 7). Notably, this SQL-specific detail is abstracted away by our query language. Finally, we apply `filter` to get the records satisfying `addCol > 5` (line 1). This corresponds to the condition found in the `WHERE` part of the SQL query.

3.2.2.2 Generating AQL Queries

Algorithm 5: Generating Abstract Queries

```
1  fun genQuerySet(σ, min, max)=
2      stopCond ← σ[depth] > min ∧ (σ[depth] > max ∨ randBool())
3      if stopCond then σ[qs]
4      else
5          match chooseFrom(σ[qsNodes]) with
6              case NewNode ⇒
7                  t ←chooseTable(σ[schema])
8                  σ2 ← σ[qs → New(t), t → t]
9                  genQuerySet(σ2++, min, max)
10             case FilterNode ⇒
11                 p ←genPred(σ++, min, max)
12                 σ2 ← σ[qs → Apply(filter, p, σ[qs])]
13                 genQuerySet(σ2++, min, max)
14             case ... ⇒
15
16 fun genPred(σ, min, max)=
17     match chooseFrom(σ[predNodes]) with
18         case EqPredNode ⇒
19             f ←choosseField(σ)
20             Eq(f, genExpr(σ++, min, max))
21         case AndPred ⇒ And(genPred(σ++, min, max), genPred (σ++, min, max))
22         case ... ⇒
23
24 fun genExpr(σ, min, max)=
25     stopCond ← σ[depth] > min ∧ (σ[depth] > max ∨ randBool())
26     exprNodes ←
27         if stopCond then (FieldRefNode, ConstantNode)
28         else σ[exprNodes]
29     match chooseFrom(exprNodes) with
30         case ConstantExpr ⇒ Constant(genConstant())
31         case FieldRefNode ⇒ Field(chooseField(σ))
32         case SumExpr ⇒
33             σ2 ← removeAggrExprs(σ[exprNodes])
34             Sum(genExpr(σ2++, min, max)
35         case ... ⇒
```

Algorithm 5 shows how we generate random AQL queries. Our algorithm generates queries that exercise all of the features supported by AQL, as well as different combinations of them.

The main component of Algorithm 5 is the `genQuerySet` function (lines 1–14). This function generates an AQL query set by recursively constructing a valid AST node based on the

syntax of Figure 3.9. The algorithm ensures that the depth of the resulting query set ranges within specific limits specified by the user-provided parameters *min* and *max* (see *stopCond*, line 2). The parameter σ keeps track of the state of the query set that is being generated. The initial state contains the schema ($\sigma[schema]$) based on which the algorithm creates table and column references. For what follows, the operation σ_{++} results in a new state where the value of $\sigma[depth]$ is incremented.

Our algorithm first constructs a new query set (`new` t) that queries a certain table (lines 6–9). To do so, we randomly choose a table to query from the underlying schema (line 7). Then, the algorithm updates the state σ in order to properly build the next available AST node in the next iteration. In particular, it initializes the AST of the current query set to *New(t)*, while it sets the queried table to t (line 8). Then, it recursively calls `genQuerySet` to construct the next available AST nodes (line 9). For example, when it is time to construct a `filter` operation applied on the current query set given by $\sigma[qs]$ (lines 10–14), the algorithm randomly generates a predicate p using the function `genPred` (lines 11, 16–22), and then extends the AST of the current query set to *Apply(filter* $p, \sigma[qs]$*)* (line 12).

The AQL predicates and expressions are generated in a similar manner (see lines 16–22, 24–35). For example, on line 31, the algorithm generates a field reference using the function *chooseField()*. This function randomly builds either a reference to a column of the queried table (e.g., $t.c$), or a reference to a label previously created by an application of the `map` function.

Finally, after producing a valid query set qs via Algorithm 5, we randomly decide for any operations applied to qs, i.e., slicing, indexing, or folding.

3.2.3 Concretization of Abstract Queries

The next phase of the proposed, data-oriented approach is the concretization of each abstract query produced in the way we presented in the previous section. In particular, during concretization, our approach derives multiple, concrete ORM queries (one for each target ORM) using ORM-specific translators (Section 3.2.3.2). Before producing these queries and executing these queries, our method populates the underlying databases with targeted data in order to enable differential testing (Section 3.2.3.1).

3.2.3.1 Generating Database Records

We follow a solver-based approach for generating a small number of targeted database records that satisfy the constraints of a given AQL query. Specifically, we model an AQL query and its constraints into SMT formulae which we pass to a theorem prover. The theorem prover then solves the given SMT formulae and generates assignments that stand for the records inserted into the database. This approach improves the effectiveness of differential testing, as the corresponding ORMs will likely return non-empty results which in turn, can be used for detecting discrepancies in ORM outputs. In the following, we explain how we model an AQL query to SMT formulae.

Modeling table columns: We introduce a sequence of SMT variables for every column of the queried table. Each variable in this sequence, namely x_i, represents the value of the column

$$
\begin{aligned}
P(c, i) &\to c \\
P(f, i) &\to f_i \\
P(e_1 \oplus e_2, i) &\to P(e_1, i) \oplus P(e_2, i) \\
A(c, g) &\to c \\
A(f, g) &\to P(f, i) \quad i \in g \\
A(\texttt{count}\ e, g) &\to \mathit{len}(g) \\
A(\texttt{sum}\ e, g) &\to \sum_{i \in g} P(e, i) \\
A(\texttt{avg}\ e, g) &\to (\sum_{i \in g} P(e, i))/\mathit{len(g)} \\
A(\texttt{max}\ e, g) &\to \mathit{max(e,\ g)} \\
A(\texttt{min}\ e, g) &\to \mathit{min(e,\ g)} \\
A(e_1 \oplus e_2, g) &\to A(e_1, g) \oplus A(e_2, g)
\end{aligned}
$$

$$
\mathit{max}(e, g) = \begin{cases} P(e, i) & g = \{i\} \\ \mathit{ite}(P(e, i) > P(e, j), P(e, i), P(e, j)) & g = \{i, j\} \\ \mathit{ite}(P(e, i) > \mathit{max}(e, g'), P(e, i), \mathit{max}(e, g')) & g = i \cdot g' \end{cases}
$$

$$
\mathit{min}(e, g) = \ldots
$$

Figure 3.11: Translating AQL expressions into SMT formulae.

x in the i^{th} record of the table, where $1 \leq i \leq n$, and n is a specified number of records inserted into the database. After declaring these variables, we model the uniqueness of the table's id. To this end, we introduce the following constraint: $id_i \neq id_j$ for $1 \leq i \leq n$, where id_i refers to the id of the i^{th} record.

Now, for what follows, $F(t_1, t_2)_i$ is the value of the foreign key defined in t_1 and refers to the table t_2, in the i^{th} record of t_1, while $V(t)$ gives the set of columns defined in table t, except for its id column.

Modeling joins: An AQL query may refer to columns defined in tables joined with the initial one, e.g., a $t_1.t_2.c$ reference. We traverse the AST of the given AQL query to identify such column references and compute the set of joins. For example, when encountering the $t_1.t_2.c$ reference, we know that there is join from table t_1 to t_2. After computing the set of joins, we introduce new SMT variables for the columns of every joined table as we did for the root table. Then, for a join between two tables t_1, t_2, we create the following constraints, for $1 \leq i < j \leq n$:

- $F(t_1, t_2)_i = id(t_2)_i$
- $F(t_1, t_2)_i = F(t_1, t_2)_j \Rightarrow \bigwedge_{v \in V(t_2)} v_i = v_j$

The first constraint indicates that the foreign key of the source table t_1 must be the same with the id of the target table t_2 for all the records of t_1. The second constraint denotes that when there are two records in t_1, namely i and j, where the values of the foreign keys for t_2 are equal,

all column values of the joined table t_2 must be also equal in the respective rows (e.g., $v_i = v_j$ for $v \in V(t_2)$). The last constraint ensures that two records of t_2 with the same id are identical.

Modeling AQL predicates: We model AQL predicates using two different ways, depending on whether the given predicate contains expressions consisting of an aggregate function (e.g., `sum`) or not. The simplest case is when a predicate does not contain an aggregate function. Such a predicate operates on all the records of the table. Converting a non-aggregate predicate is straightforward. For example, we convert the AQL equality predicate $t.c = e$ into:

$$\exists i.\ t.c_i = P(e, i) \text{ for } 1 \leq i \leq n$$

In the above formula, $t.c_i$ is the SMT variable that represents the value of the column $t.c$ in the i^{th} record of the table, while the function $P(e, i)$ encodes the given AQL expression e into a logical formula as shown in Figure 3.11. The above logical formula encodes the constraint that there must be at least one record in the table where the value of the column $t.c$ is equal with the value of the expression e.

An AQL predicate containing an aggregate function works on aggregated data formed by groups of records, and is conceptually similar to a condition that appears in the `HAVING` clause of an SQL query. To model such predicates as logical formulae, we first create a set G, consisting of a specified number of groups of records. Each group $g \in G$ includes all records that are identical based on a set of grouping fields GF. For example, $G = \{\{1, 2\}, \{3\}\}$ means that we have two groups. The first group contains the first and the second record of the table, while the second group only includes the third record. To compute the set of grouping fields GF, we traverse the AST of the given AQL query and add all column references that are not passed to an aggregate function. We then generate constraints so that the records of the same group are identical with respect to each field found in GF. Finally, we model aggregate predicates and their AQL expressions using the function $A(e, g)$ as defined in Figure 3.11. For example, the AQL predicate $t.a =$ `sum` $t.b$ is translated into:

$$\exists g \in G.\ A(t.a, g) = A(\texttt{sum}\ t.b, g)$$

In the example above, $A(t.a, g)$ gives the SMT variable of the column $t.a$ associated with a *random* record of the group g. This is because $t.a$ is a grouping field (it is not part of an aggregate function) and all the records of g are the same with respect to the value of $t.a$. On the other hand, $A(\texttt{sum}\ t.b, g)$ aggregates all records of the group g based on the column $t.b$, i.e., $\sum_{i \in g} P(t.b, i)$. As Figure 3.11 indicates, the main difference between the functions $P(e, i)$ and $A(e, g)$ is that the former encodes the expression e as an SMT formula with regards to the record i, while the latter reasons about a group of records.

Modeling Unions & Intersections. Modeling unions and intersections is straightforward. Each sub-query of such an operation (e.g., $qs_1 \cup qs_2$) is translated into an SMT formula separately. Then the individual formulae are combined through logical operators. For unions, we use the disjunction operator ($\vee$), while we use $\wedge$ in case of intersection.

```
1  import os, django
2  from django.db.models import *
3  os.environ.setdefault("DJANGO_SETTINGS_MODULE",
4                        "djangoproject.settings")
5  django.setup()
6  from project.models import *
7
8  addCol = F("colA") + F("t2__colB")
9  q = T1.objects.using("sqlite")\
10   .annotate(addCol=addCol).filter(addCol__gt=5)\
11   .values("addCol")
12 for r in q:
13   print("addCol", r["addCol"])
```

Figure 3.12: The Django code related to the AQL query of Figure 3.10.

3.2.3.2 From Abstract Queries to Concrete ORM Queries

We employ ORM-specific translators to convert a given AQL query into the corresponding concrete ORM query. Note that for each ORM under test, there is a corresponding translator. A translator takes an AQL query, converts it into an ORM query, and produces an executable file that runs the ORM query on a specified DBMS. Hence, a translator produces multiple executable files, one for each provided DBMS.

Every translator consists of three components. The first component adds the necessary boilerplate code for running the ORM query (e.g., imports, creating the connection with the database, etc.). The second component performs the translation. Specifically, it uses the API of the corresponding ORM to generate the actual ORM query. The last component dumps the results of the query to standard output, again by using the API of the specified ORM. When the query produces a sequence of records, the translator produces code that iterates over each element of the sequence and prints this element to standard output. To properly dump a record, the translator emits code that prints the value of every field defined in the AQL query. For example, when the query contains an application of `map`, the translator produces code that prints the value of every non-hidden field defined in `map`. When the given query does not apply `map`, then the id of the fetched records is printed. Finally, for queries returning scalar values (i.e., `fold`), the translator emits code that prints these scalar values.

Figure 3.12 shows the executable file that corresponds to the AQL query of Figure 3.10 and is produced by the Django translator. Notice that this file runs the Django query on SQLite. Lines 1–6 contain the necessary setup code for running the query, the actual Django query is on lines 8–11, while on lines 12–13, we print the results of the query.

3.2.4 Bug Detection

The last step of our testing approach is to run the executables produced by the translators and compare the output of these executables for mismatches. To do so, we run every executable and capture its standard output and standard error.

Our approach makes DBMS-specific comparisons: the output of a query q written in ORM

o_1 and run on DBMS x is compared against the same query q written in another ORM o_2 and run on the *same* DBMS x. We do this because certain query features may be unsupported by some DBMSs (e.g., MySQL does not support intersection queries.) This means that a particular ORM query might run smoothly on a certain DBMS, but fail when running on another DBMS. Therefore, comparisons across different DBMSs might given unreliable results.

Based on the above, our approach identifies mismatches and flags them as bugs, when one of the following conditions holds:

- The same query written in two different ORMs produces different results on the same DBMS.
- A query written in a certain ORM runs successfully on a specific DBMS, but the same query written in another ORM fails on the same DBMS. This condition allows us to detect cases where an ORM produced either a grammatically or semantically invalid SQL query with regards to a certain DBMS (recall Section 2.3.2.1).

Remark: As already discussed, when a query is not ordered, the underlying DBMS is free to fetch the records in any order. To make safe comparisons between unordered queries, our approach first sorts the outputs of these queries, and then compares them. Moreover, to eliminate inconsistencies in the ORM outputs caused by the fact that the executable files are written in different programming languages, we proceed as follows. First, we represent `null` values uniformly across languages. This is because different programming languages may output `null` values differently (e.g., Python uses `None` instead of `null`), Second, to avoid differences caused by accuracy issues of floating-point arithmetics, we use a small delta value ϵ when performing comparisons between floats.

3.3 A Model for Detecting Dependency Bugs in File System Resources

We present our approach for detecting dependency bugs in configuration management and build scripts. Our approach relies on an execution abstraction, and consists of three major phases, as shown in Figure 3.13. During the first phase (*Generation* step), we execute a given *instrumented* configuration management or build script and we monitor its execution by tracking all system calls of the main system process and its descendants. By tracing all system operations, we generate an *abstract view* of the execution (see execution abstraction, Figure 3.1b) using a model called FSMOVE (File System MOdel VErifier), which is used to model the semantics of every system call that affects the file system.

Then, we proceed with the *Analysis* phase, where we analyze the generated FSMOVE program and produce two outcomes (reasoning). First, we construct *task graph*, a directed acyclic graph that contains all the task inter-dependencies as *declared* in the original build and configuration management scripts by the programmers. Notably, a task graph acts as a test oracle that describes the *expected* behavior of the script under test. Second, we track all the *actual* file accesses of every task by evaluating the given FSMOVE program based on a semantics.

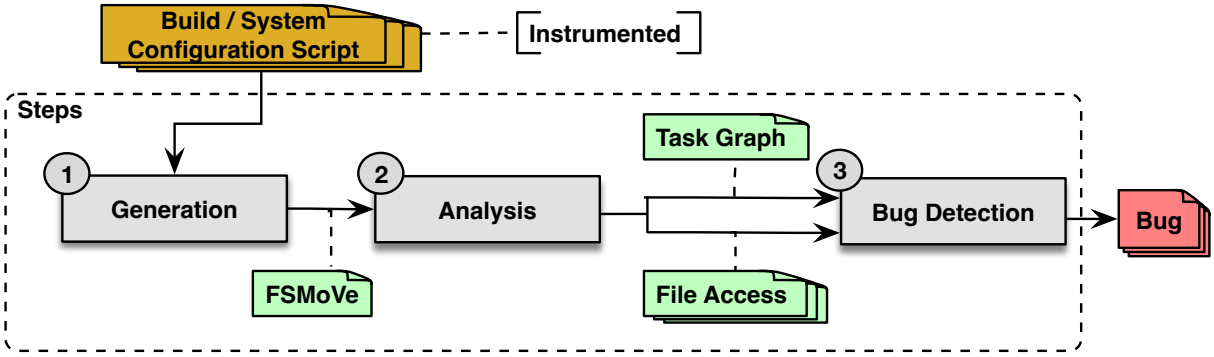


Figure 3.13: The overview of our approach for detecting bugs in file system-intensive software. First, we monitor the execution of an instrumented script (i.e., build, configuration management), and then we report bugs by analyzing the generated FSMoVe programs.

In the final phase (*Bug Detection*), we verify the correctness of the given execution (modeled in FSMoVe) using the file accesses and the task graph computed by the previous step. In particular, this phase compares the set of file accesses against a task graph (i.e., test oracle), and reports those file accesses that violate the correctness of the given execution, i.e., they lead to missing inputs, missing outputs, missing notifiers, or ordering violations. These issues are reported according to a set of formal FSMoVe definitions, which we will explain in Section 3.3.3.

Remark on execution abstraction: Our approach for detecting dependency bugs in file system resources is based on a execution abstraction, that is, the FSMoVe model. To enable reasoning for bug detection, FSMoVe captures (1) the specification of a script under test (written in a high-level DSL), and (2) the runtime effects of a script under test on the file system. Then, the FSMoVe model formally describes four different types of dependency bugs in term of the conditions under which a file system operation (based on runtime information) violates the intended behavior (specification) of the given script. To do so, FSMoVe models high-level specifications through the task graph, while it also computes all the runtime file accesses through a semantics. Using the above FSMoVe definitions, bug identification is then straightforwardly done by cross-checking the expected behavior (i.e., task graph) with the actual one (i.e., set of file accesses) of the script under test. FSMoVe is a generally-applicable model, as an arbitrary execution can be easily expressed in terms of FSMoVe constructs. The only requirement to do so is the implementation of an instrumentation used to augment execution with information taken from the original high-level script specifications (see Section 3.3.4 for more details). Finally, Section 3.3.1 provides more details about the advantages of our FSMoVe model over the existing ones.

Now, we give a formal definition of the FSMoVe model that underpins our dynamic approach, and then we proceed with the detailed description of each phase of our method.

3.3.1 FSMoVe: An Abstract View of File System-Oriented Executions

In Section 2.4.3, we enumerated the main challenges associated with identifying dependency bugs in file system resources. We also discussed how existing work suffers from fundamental issues related to method's design and underlying model, and *not* with its implementation. We

now introduce FSMOVE, a model for thinking about and abstracting executions that addresses the challenges and the limitations of the existing work.

The proposed model treats every execution (i.e., a build execution, a configuration management execution) as a sequence of *tasks* rather than system processes. Every task corresponds to the execution of an operation. For example, a task in FSMOVE stands for the execution of a target rule in Make and Ninja, a goal in Java Maven, a resource in Puppet, or a Gradle task in Gradle. This tackles low precision introduced by prior work, because it enables us to relate every task to its correct input and output files regardless of the internal behavior of the execution engine (e.g., whether it spawns a separate process or not). For example, unlike the overconstrained dependency graph of Figure 2.30a, FSMOVE allows us to infer the precise graph shown in Figure 2.30b. FSMOVE separates file accesses based on which task they belong to. Therefore, it does not perform unnecessary merges when encountering tasks governed by the same process, which is the main source of imprecision in previous work [Licker and Rice, 2019]. Conceptually, this design decision allows us to split the main process that governs the execution into different blocks that indicate the boundaries of every task. In this way, we are able to capture precisely the interactions of each task with the underlying operating system, and thus addressing Challenge 2.4.3.1.

To tackle the challenge of modeling low-level file system operations (Section 2.4.3.2), FSMOVE enables us to: (1) translate every system call into primitives that conquer the large number and complexity of POSIX (Unix/Linux) system calls by decomposing them into simpler building blocks and (2) formally model the effects of system calls on the file system and the transient OS structures (see Figure 3.16 for more details).

For dealing with efficiency and applicability (Section 2.4.3.3), FSMOVE provides each task with a specification that consists of (1) a set of files that the task is expected to consume, (2) a set of files that the task is expected to produce, and (3) a set of task dependencies. A task dependency indicates that a task depends on another, i.e., it is executed only after the dependent task. Beyond specification, every task has a definition containing all the (low-level) file system operations performed while executing the task, e.g., reading and writing files, or changing the OS transient structures, such as the file descriptor table. Combining the (high-level) specification and the actual behavior (low-level file system operations) of each task makes our approach efficient and applicable, for we can verify correctness (i.e., whether the actual behavior conforms to the specification), by analyzing the execution of a single execution. Therefore, there is no need to run subsequent executions or static analysis on scripts written in the corresponding DSLs. The verification process is explained in detail in Section 3.3.3.

Figure 3.14 shows the complete model for abstracting file system-oriented executions. An execution $b = \langle t^1, t^2 \dots \rangle$ consists of a sequence of tasks. Every task t is described by a unique name ($\tau \in \mathit{TaskName}$), and contains a specification and a definition. The specification declares the input / output files and the dependencies of each task, while its definition consists of statements. For example, `task A (/file/in): /file/out after` $\bot = s$ means that the task named `A` consumes the file `/file/in`, produces the file `/file/out`, has no dependencies (`after` $\bot$), while its definition is given by s. As another example, `task A B: C after D` $= s$ indicates that task `A` takes as input all files produced by task `B`, produces all files consumed by task `C`, while

$$
\begin{array}{rcl}
\langle b \in \mathit{Execution} \rangle & ::= & t^* \quad \texttt{[execution]} \\
\langle t \in \mathit{Task} \rangle & ::= & \texttt{task } \tau \texttt{ } k\texttt{: } k \texttt{ after } d \texttt{ = } s \quad \texttt{[task]} \\
\langle d \in \mathit{Dep} \rangle & ::= & \bot \mid \tau \mid (\tau \ldots) \quad \texttt{[task deps]} \\
\langle k \in \mathit{FileSpec} \rangle & ::= & p \mid (p \ldots) \mid \tau \mid (\tau \ldots) \mid \bot \mid \top \quad \texttt{[file spec]} \\
\langle s \in \mathit{Statements} \rangle & ::= & \texttt{sysOp in } z \texttt{ = } o \quad \texttt{[operation]} \\
 & & \mid \texttt{newproc } z \quad \texttt{[new process]} \\
 & & \mid \texttt{newproc } z \texttt{ from } z \quad \texttt{[fork]} \\
 & & \mid s\texttt{;}s \quad \texttt{[compound stmt]} \\
\langle o \in \mathit{Op} \rangle & ::= & \texttt{fd}_f \texttt{ = } e \quad \texttt{[create fd]} \\
 & & \mid \texttt{del(fd}_f\texttt{)} \quad \texttt{[destroy fd]} \\
 & & \mid \texttt{consume(}e\texttt{)} \quad \texttt{[consume path]} \\
 & & \mid \texttt{produce(}e\texttt{)} \quad \texttt{[produce path]} \\
 & & \mid \texttt{mv(}e\texttt{, }p\texttt{)} \quad \texttt{[move]} \\
 & & \mid o\texttt{;}o \quad \texttt{[compound op]} \\
\langle e \in \mathit{Expr} \rangle & ::= & p \quad \texttt{[path]} \\
 & & \mid \texttt{fd}_f \quad \texttt{[fd]} \\
 & & \mid p \texttt{ at } e \quad \texttt{[}p \texttt{ relative to } e\texttt{]} \\
\langle p \in \mathit{Path} \rangle & ::= & \textit{is the set of paths} \\
\langle \tau \in \mathit{TaskName} \rangle & ::= & \textit{is the set of task names} \\
\langle f \in \mathit{FileDesc} \rangle & ::= & \textit{is the set of file descriptors} \\
\langle z \in \mathit{Proc} \rangle & ::= & \textit{is the set of processes}
\end{array}
$$

Figure 3.14: The syntax for representing executions in FSMOVE.

it is executed after task D. The symbol $\bot$ is used to indicate the absence of a value, while $\top$ is used as a wildcard symbol representing any value. For example, when we say that the declared output of a task is $\top$, we mean that this task is valid to produce *any* file.

A definition is one or more statements. There are two types of statements. First, the `sysOp in` z `=` o statement executes a system operation o in a process given by z. Every process defines a scope for file descriptor variables ($\texttt{fd}_f$), where each file descriptor variable points to an file resource. An operation ($o \in \mathit{Op}$) executed inside a process z may introduce new file descriptor variables in the current process (scope) ($\texttt{fd}_f = \ldots$), or delete existing ones (`del`). Moreover, an operation may perform various file system updates, including file creation (`produce`) and file consumption (`consume`). Finally, the operation `mv`(e, p) arranges that the resource accessed by expression e is also accessed by new path p.

An expression ($e \in \mathit{Expr}$) can be a constant path, a file descriptor variable, or p `at` e. The latter allows us to interpret the path p relative to the path given by the e expression. As we will see later when studying the semantics of FSMOVE, the result of an expression is a file resource indicated by a file path. Finally, the `newproc` z_1 statement creates a fresh process (scope) z_1, and it optionally copies all file descriptor variables of an existing process z_2 to z_1 (`newproc` z_1 `from` z_2). This models process forking.

```
1  # Copying file using a Make rule.
2  target: "/source"
3      cp $^ $@
```

```
1  // Copying file using a Gradle task.
2  task target {
3    inputs.file "/source"
4    outputs.file "/target"
5    from file("/source")
6    into file("/target") }
```

(a) Copying the contents of `source` into `target`.

```
1  task target ("/source"): "/target"
       after ⊥ =
2    newproc p
3    sysOp in p =
4      fd3 = "/source"
5      consume(fd3)
6      fd4 = "/target"
7      produce(fd4)
8      del(fd4)
9      del(fd3)
```

(b) Modeling the execution of the task `target` that stems from Make and Gradle scripts of Figure 3.15a.

Figure 3.15: An example of FSMove modeling.

As an example of modeling, consider a simple build scenario where we want to copy the contents of the file `/source` into the file `/target`. Figure 3.15a shows how we can express this using Make and Gradle. When we execute these build scripts, the build system first opens the file `/source`, reads its contents, then opens the file `/target`, and finally writes the contents of `/source` to the file descriptor corresponding to the second file. Figure 3.15 illustrates how we model the execution stemming from these scripts. Every build consists of a single task named `target`. This task consumes `/source` to create `/target`. The definition of the task `target` creates a new process (line 2) and uses it to execute all the file-related operations performed when running Make and Gradle (lines 3–9). For instance, the operation $\texttt{fd}_3 = \texttt{/source}$ creates a new file descriptor (in the current process) pointing to file `/source`, while the operation at line 5 consumes this file descriptor. These two operations model file opening. On the other hand, the operation $\texttt{del(fd}_4\texttt{)}$ deletes the given file descriptor once the task closes the corresponding file (line 8).

The semantics of FSMove is shown in Figure 3.16. Every task $t \in \mathit{Task}$ is evaluated on a state $\sigma \in \mathit{State} = \mathit{ProcT} \times \mathit{InodeT}$. A state is a pair of a process and an `inode` table. The former is a map that provides the file descriptor table (*ProcFD*) of every process. A file descriptor table $\pi \in \mathit{ProcFD} = \mathit{FileDesc} \hookrightarrow \mathit{INode}$ is a map that provides the `inode` that each file descriptor of a process points to. We use this component to map the open file descriptors of a process to the resource they handle. Conceptually, *ProcT* models the file descriptor feature of POSIX-compliant operating systems. An `inode` table $\tau \in INodeT$ is a map of a pair, consisting of an `inode` and a file name to another `inode`. The first element of the pair is the `inode` of the directory where the file name exists. An `inode` is a positive integer that acts as the *identifier* for a certain file system resource. Note that we also keep the special `inode` ι_r which corresponds to the `inode` of the root directory "/". The `inode` table mimics the `inode` structure implemented in Unix-like operating systems. For example, the `inode` of the `/foo` file, whose value is 3, is stored as follows: $[(\iota_r, \text{“foo”}) \to 3)]$.

The result of a task evaluation $[\![t]\!]_\sigma \in \sigma \to \sigma \times r$ is a new state along with the *file accesses* performed by this task. An instance of the domain of file accesses $r \in \mathit{FileAcc} = \mathcal{P}(\mathit{Path}) \times$

$$\iota \in INode = \{\iota_i \mid i \in \mathbb{Z}^*\} \cup \{\iota_r\}$$
$$r \in FileAcc = \mathcal{P}(Path) \times \mathcal{P}(Path)$$
$$\tau \in INodeT = (INode \times Filename) \hookrightarrow INode$$
$$\pi \in ProcFD = FileDesc \hookrightarrow INode$$
$$\phi \in ProcT = Proc \hookrightarrow ProcFD$$
$$\sigma \in State = ProcT \times INodeT$$

(a) Domains

$$\textsc{dir}(p) = \text{parent directory of path } p$$
$$\textsc{base}(p) = \text{base name of path } p$$
$$\textsc{join}(p_1, p_2) = \text{join file paths } p_1 p_2$$
$$P(\iota, \tau) = \text{file path which inode } \iota \text{ is accessed through}$$
$$I(p, \tau) = \text{inode of path } p$$
$$\textsc{key}(p, \tau) = (I(\textsc{dir}(p), \tau),\ \textsc{base}(p))$$

(b) Definitions.

$$r_{\downarrow c} = r_{\downarrow_1} \quad r_{\downarrow p} = r_{\downarrow_2}$$
$$[\![e]\!] \in \pi \times \tau \to Path$$
$$[\![p]\!]_{\pi,\tau} \Rightarrow p$$
$$[\![\texttt{fd}_f]\!]_{\pi,\tau} \Rightarrow P(\pi(f), \tau)$$
$$[\![p \texttt{ at fd}_f]\!]_{\pi,\tau} \Rightarrow \textsc{join}([\![\texttt{fd}_f]\!]_{\pi,\tau},\ p)$$
$$[\![o]\!] \in \pi \times \tau \times r \to \pi \times \tau \times r$$
$$[\![\texttt{fd}_f = e]\!]_{\pi,\tau,r} \Rightarrow (\pi[f \to \iota], \tau, r)\ ,\ \iota = I([\![e]\!]_{\pi,\tau},\ \tau)$$
$$[\![\texttt{del(fd}_f\texttt{)}]\!]_{\pi,\tau,r} \Rightarrow (\pi[f \to \bot], \tau, r)$$
$$[\![\texttt{consume}(e)]\!]_{\pi,\tau,r} \Rightarrow (\pi, \tau, (r_{\downarrow c} \cdot [\![e]\!]_\pi\ ,\ r_{\downarrow p}))$$
$$[\![\texttt{produce}(e)]\!]_{\pi,\tau,r} \Rightarrow (\pi, \tau, (r_{\downarrow c}\ ,\ r_{\downarrow p} \cdot [\![e]\!]_\pi))$$
$$[\![\texttt{mv}(e, p)]\!]_{\pi,\tau,r} \Rightarrow (\pi, \tau[k \to \iota], r)\ ,\ k = \textsc{key}(p, \tau) \text{ and } \iota = I([\![e]\!]_{\pi,\tau},\ \tau)$$
$$[\![o_1; o_2]\!]_{\pi,\tau,r} \Rightarrow [\![o_2]\!]_{\pi',\tau',r'} \quad (\pi', \tau', r') = [\![o_1]\!]_{\pi,\tau,r}$$
$$[\![s]\!] \in \sigma \times r \to \sigma \times r$$
$$[\![\texttt{sysOp in } z = s]\!]_{\phi,\tau,r} \Rightarrow (\phi[z \to \pi], \tau, r) \quad (\pi, \tau, r) = [\![s]\!]_{\phi(z),\tau,r}$$
$$[\![\texttt{newproc } z]\!]_{\phi,\tau,r} \Rightarrow (\phi[z \to \bot], \tau, r)$$
$$[\![\texttt{newproc } z_1 \texttt{ from } z_2]\!]_{\phi,\tau,r} \Rightarrow (\phi[z_1 \to \phi(z_2)], \tau, r)$$
$$[\![s_1; s_2]\!]_{\sigma,r} \Rightarrow [\![s_2]\!]_{\sigma',r'} \quad (\sigma', r') = [\![s_1]\!]_{\sigma,r}$$
$$[\![t]\!] \in \sigma \to \sigma \times r$$
$$[\![\texttt{task } \tau\ k_1 : k_2 \texttt{ after } d = s]\!]_\sigma \Rightarrow [\![s]\!]_{\sigma,\bot}$$

(c) Semantics

Figure 3.16: Semantic domains of FSMove (a), some auxiliary definitions (b), and the semantics of FSMove expressions, operations, statements, and tasks (c).

$\mathcal{P}(Path)$ is a pair that contains the set of files consumed and produced by the task. The first element of the pair is the set of consumed files, while the second one stands for the set of produced files. For convenience, we also define the projections $r_{\downarrow c}$ and $r_{\downarrow p}$ that give the set of files consumed, and produced by the task respectively. Statements, operations, and expressions are evaluated accordingly. Notably, operations and expressions are evaluated using (1) the file descriptor table (i.e., $\pi \in ProcFD$) of the process where they take place, and (2) the inode table $\tau \in INodeT$. As an example, after evaluating the following FSMove task on the state $\sigma = \bot$, we get a new state σ' that involves (1) a process table of the form $z_1 \to (1 \to 4, 2 \to 6)$, (2) an inode table of the form $((\iota_r, \texttt{"f1"}) \to 4, (4, \texttt{"f3"}) \to 5)$, while (3) the set of consumed files $r_{\downarrow c}$ is $\{\texttt{"/f1/f3"}\}$, and the set of produced files $r_{\downarrow p}$ is $\{\texttt{"/f2/f4"}\}$.

```
1  task τ ("/f1"): "/f2" after ⊥ =
2    newproc z1
3    sysOp in z1 =
4      fd1 = "/f1"
5      fd2 = "f3" at fd1
6      consume(fd2)
7      produce("f4" at "/f2")
```

```
1  task t1 ("/f1"): "/f2"
2    after ⊥ = s1
3  task t2 ("/f2"): ⊥
4    after t1 = s2
5  task t3 ("/f3", "/f4"): ("/f2", "/f5")
6    after ⊥ = s3
```

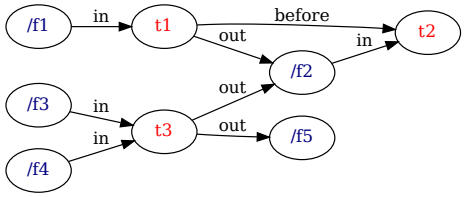


Figure 3.17: An example of an FSMoVe execution and its task graph.

3.3.2 Task Graph

We introduce the notion of *task graph*. The task graph is a component that stores the input files, output files, and the dependencies of every task as declared by the developers in build or configuration management scripts. The task graph is computed by traversing the specification of every task found in a FSMoVe program (Section 3.3.1), and collecting all input / output files and task dependencies. We later use the task graph for ensuring the correctness of the FSMoVe execution (Section 3.3.3).

We define the task graph as $G = (V, E)$. A node $v \in V$ in the task graph is either a task $t \in \mathit{Task}$ or a file $p \in \mathit{Path}$. The set of edges $E \subseteq V \times V \times L$, where $L = \{\texttt{in}, \texttt{out}, \texttt{before}\}$, determine the following relationships. Given a task graph G, the edge $v \xrightarrow{\texttt{in}}_G t$ indicates that the node v (it can be either a task t or a file path p) has been declared as an input of the task t. The edge $t \xrightarrow{\texttt{out}}_G v$ states that the task t produces node v. Finally, the edge $t_1 \xrightarrow{\texttt{before}}_G t_2$ shows a task dependency, i.e., the execution of t_1 precedes that of t_2.

Given an execution modeled as a FSMoVe program (Figure 3.14), we gradually compute the task graph by inspecting the specification of every task entry. The $\xrightarrow{l}$ edges, where $l \in \{\texttt{in}, \texttt{out}, \texttt{before}\}$, are constructed by examining the header part of a `task` construct. For instance, whenever we encounter a task entry of the form `task` t_1 (p_1): p_2 `after` t_2, we add the following edges to the task graph G: (1) an $p_1 \xrightarrow{\texttt{in}}_G t_1$ edge, (2) an $t_1 \xrightarrow{\texttt{out}}_G p_2$ edge, and (3) an $t_2 \xrightarrow{\texttt{before}}_G t_1$ edge. Notably, all these edges denote the intent of developers as specified in the underlying scripts.

A complete example is shown in Figure 3.17, where we have an execution in FSMoVe on the left, and its resulting task graph on the right. Red nodes denote tasks while blue nodes indicate files.

3.3.3 Correctness of FSMoVe Executions

Having proposed our FSMoVe model and the concept of task graph, we now formalize the property of correctness for executions modeled in FSMoVe. To do so, we exploit the task graph and define two relations that we use as a base for verifying correctness: (1) the *subsumption* relation, and (2) the *happens-before* relation.

Definition 3.3.1 *(Subsumption). Given a task graph G, we define the reflexive, binary relation $\sqsubseteq_G$ and its transitive closure $\sqsubseteq_G^+$ on two nodes $v_1, v_2 \in G$. The definition is shown in Figure 3.18.*

$$\textsc{self}\ \frac{v \in G}{v \sqsubseteq_G v} \qquad \textsc{top}\ \frac{v \in G}{v \sqsubseteq_G \top} \qquad \textsc{par-dir}\ \frac{p_1, p_2 \in \mathit{Path} \quad \textsc{isParentDir}(p_1, p_2)}{p_1 \sqsubseteq_G p_2}$$

$$\textsc{indirect}\ \frac{p_1, p_2 \in \mathit{Path} \quad t \in G \quad p_1 \sqsubseteq_G p_1' \quad p_1' \xrightarrow{\text{in}}_G t \quad t \xrightarrow{\text{out}}_G p_2}{p_1 \sqsubseteq_G p_2}$$

$$\textsc{task}\ \frac{t_1, t_2 \in \mathit{Task} \quad t_1 \xrightarrow{\text{in}}_G t_2}{t_2 \sqsubseteq_G t_1} \qquad \textsc{trans-clos}\ \frac{p_1 \sqsubseteq_G p_2}{p_1 \sqsubseteq_G^+ p_2} \qquad \textsc{trans-clos}\ \frac{p_1 \sqsubseteq_G^+ p_2 \quad p_2 \sqsubseteq_G^+ p_3}{p_1 \sqsubseteq_G^+ p_3}$$

$$\textsc{mul}\ \frac{m = (p_1, p_2 \dots) \quad \exists p' \in m.\ p \sqsubseteq_G^+ p'}{p \sqsubseteq_G^+ m}$$

Figure 3.18: Definition of the $\sqsubseteq_G$ relation through inference rules.

$$\textsc{dep}\ \frac{t_1, t_2 \in \mathit{Task} \quad t_1 \xrightarrow{\text{before}}_G t_2}{t_1 \prec_G t_2} \qquad \textsc{trans-clos}\ \frac{t_1 \prec_G t_2}{t_1 \prec_G^+ t_2} \qquad \textsc{trans-clos}\ \frac{t_1 \prec_G^+ t_2 \quad t_2 \prec_G^+ t_3}{t_1 \prec_G^+ t_3}$$

Figure 3.19: Definition of the $\prec_G$ relation through inference rules.

Notably, this relation is defined on both file paths and tasks. This relation is reflexive ([SELF]), and for every node $v \in G$, we have $v \sqsubseteq_G \top$. In case of file paths $p_1, p_2 \in \mathit{Path}$, the subsumption relation $p_1 \sqsubseteq_G p_2$ says that the file path p_1 is subsumed within the file path p_2. The relation $p_1 \sqsubseteq_G p_2$ holds when p_2 is the parent directory of p_1 ([PAR-DIR]), or when p_2 relies on p_1, i.e., there is at least one task in the task graph G that produces p_2 using p_1 ([INDIRECT]). For tasks $t_1, t_2 \in \mathit{Task}$, we say that task t_2 is subsumed in t_1, when there is an $\xrightarrow{\text{in}}$ edge from t_1 to t_2. As we will see later the subsumption relation is important for ensuring that a file access made while executing a build task matches the task's specification.

Definition 3.3.2 *(Happens-Before). Given a task graph G, we define the binary relation $\prec_G$ and its transitive closure $\prec_G^+$ on two tasks $t_1, t_2 \in \mathit{Task}$. The definition is shown in Figure 3.19.*

The happens-before relation $t_1 \prec_G t_2$ states that the task t_1 is executed before t_2. The definition of this relation consults the task graph G to identify tasks that are connected with each other through an $\xrightarrow{\text{before}}_G$ edge, which indicates a dependency between two tasks. Finally, the transitive closure of $\prec_G$ gives indirect task dependencies. The happens-before relation enables us to verify whether two dependent tasks are executed in the correct order.

3.3.3.1 Verifying Correctness of Tasks

Using the subsumption relation, we now formalize what the property of correctness means for an FSMOVE task.

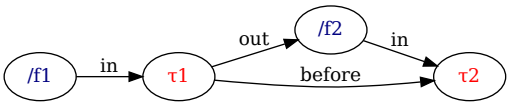
Definition 3.3.3 *(Missing Input). Given a task graph G and a state σ, a task $t \in Task = \texttt{task}\ \tau\ k_1 \texttt{:}\ k_2\ \texttt{after}\ d = s$ manifests a* missing input *on state σ, when*

- $(\sigma', r) = [\![t]\!]_\sigma$
- $\exists p \in r_{\downarrow_c}\ p \not\sqsubseteq_G^+ k_1$

In other words, to verify that a task does not contain a missing input issue, we first compute all file accesses made by the task on the given state σ, i.e., $[\![t]\!]_\sigma$. Then, we check that every file consumed by this task (i.e., $r_{\downarrow_c}$) matches the input files k_1 declared in the specification. To do so, we exploit the subsumption relation. In particular, when there exists a file path p consumed by this task for which $p \not\sqsubseteq_G k_1$, we say that the task has a missing input on the state σ. In practice, this means that although the task relies on p (as the definition of task consumes p), the execution engine does not trigger the execution of the task, whenever p is modified.

Example. Consider the following FSMove program and its task graph.

```
1  task τ1 ("/f1"): "/f2" after ⊥ =
2    newproc z1
3    sysOp in z1 =
4      consume "/f1/f3"
5      produce "/f2"
6  task τ2 ("/f2"): ⊥ after τ1 =
7    sysOp in z1 =
8      consume "/f2"
9      consume "/f1/f3"
10     consume "/f3"
```



When examining the task τ_1, we presume that it does not contain any missing input issues, as it only consumes the file `"/f1/f3"` (line 4) which is subsumed within the input file `/f1` declared at line 1 (recall the [PAR-DIR] rule from Figure 3.18). On the other hand, the task τ_2 consumes three files (lines 8–10). For the first access (line 8), the $\sqsubseteq_G^+$ relation holds, as the path consumed by τ_2 is the same with that declared in the specification. The subsumption relation also holds for the second access, as the file `f1/f3` is an input of the first task τ_1 whose output is used as an input for the target task τ_2. Therefore, changing this file will first trigger the execution of task τ_1. This will eventually cause the invocation of task τ_2, because the first task τ_1 updates the inputs of τ_2. This behavior is captured by the [INDIRECT] rule (Figure 3.18). Finally, the task τ_2 manifests a missing input for the third access (line 10), because we have `/f3` $\not\sqsubseteq_G^+$ `/f2`.

Definition 3.3.4 *(Missing Output). Given a task graph G and a state σ, a task $t \in Task = \texttt{task}\ \tau\ k_1 \texttt{:}\ k_2\ \texttt{after}\ d = s$ manifests a* missing output *on state σ, when*

- $(\sigma', r) = [\![t]\!]_\sigma$
- $\exists p \in r_{\downarrow_p}\ p \not\sqsubseteq_G^+ k_2$

The definition for missing outputs is conceptually similar to that for missing inputs. This time however, we check that for every file $p \in r_{\downarrow_p}$ produced by the examined task, the $p \sqsubseteq_G^+ k_2$

relation holds, where k_2 stands for the declared output files found in the specification of the task.

Given the above definitions, we now introduce the notion of correctness for a certain FSMOVE task.

Definition 3.3.5 *(Correctness of Task). Given a task graph G and a state σ, a task $t \in$ Task is correct on state σ, when it does not manifest a missing input or missing output on state σ.*

3.3.3.2 Verifying Correctness of Executions

Recall that an execution in FSMOVE is a sequence of tasks $b = \langle t^1, t^2 \dots \rangle$. An execution may manifest an ordering violation, when there are pairs of tasks that access a file p, at least one of them produces p, and there is no ordering constraint between these tasks (i.e., they can be executed in any order).

Definition 3.3.6 *(Ordering Violation). Given a task graph G and an initial state $\sigma^0 = \bot$, a build execution $b = \langle t^1, t^2 \dots t^n \rangle$ manifests an* ordering violation *on state σ_i, when $\exists j$ with $1 \leq j < i < n$ such that*

- $(\sigma^{i+1}, r^{i+1}) = [\![t^{i+1}]\!]_{\sigma^i}$ *and* $(\sigma^j, r^j) = [\![t^j]\!]_{\sigma^{j-1}}$
- $|r^{i+1}_{\downarrow_p} \cap (r^j_{\downarrow_c} \cup r^j_{\downarrow_p})| \geq 1 \vee |r^j_{\downarrow_p} \cap (r^{i+1}_{\downarrow_c} \cup r^{i+1}_{\downarrow_p})| \geq 1$
- $t^{i+1} \not\prec^+_G t^j \wedge t^j \not\prec^+_G t^{i+1}$

Contrary to missing inputs and outputs, the definition for ordering violations checks whether two tasks with a conflicting file access are executed in the right order. To achieve this, we use the happens-before relation $\prec^+_G$. For example, consider a task t_1 that creates a file p. When the same file is consumed by a task t_2, the $t_1 \prec^+_G t_2$ must hold. Otherwise, the build system is free to execute t_2 before t_1. Therefore, t_2 may access a file that does not exist, resulting in an execution failure.

Now, we formally define the case when an execution manifests an issue related to missing notifiers.

Definition 3.3.7 *(Missing Notifier). Given a task graph G and an initial state $\sigma^0 = \bot$, a build execution $b = \langle t^1, t^2 \dots t^n \rangle$ manifests a* missing notifier *on state σ_i, when $\exists j$ with $1 \leq j < i < n$ such that*

- $(\sigma^{i+1}, r^{i+1}) = [\![t^{i+1}]\!]_{\sigma^i}$ *and* $(\sigma^j, r^j) = [\![t^j]\!]_{\sigma^{j-1}}$
- $\exists p \in$ *Path*. $p \in r^{i+1}_{\downarrow_c} \wedge p \in r^j_{\downarrow_p}$
- *isService*(t^{i+1})
- $t^{i+1} \not\sqsupseteq^+_G t^j$

```
1  + write(1, "#FSMoVe#: Begin target");
2  + write(1, "#FSMoVe#: target input /source");
3  + write(1, "#FSMoVe#: target output /target");
4  open("/source", O_RDONLY) = 3;
5  open("/target", O_WRONLY|O_CREAT) = 4;
6  read(3, "content");
7  write(4, "content");
8  close(4);
9  close(3);
10 + write(1, "#FSMoVe#: End target");
```

Figure 3.20: Example of the native function calls observed, while executing the build scripts of Figure 3.15a. The highlighted function calls are those inserted by our instrumentation.

The above definition identifies pairs of tasks where the execution of the first element should trigger the execution of the second one. In the context of configuration management (e.g., Puppet), such relationships involve `service` resources. Definition 3.3.7 looks for a task $t_1 \in \mathit{Task}$ that produces a particular resource p, after evaluating this task on a state σ. This task must have notification relation with another service-oriented task t_2 consuming the resource p as derived from the evaluation of t_2 on a subsequent state σ'. If such a notification relation does not exist, i.e., the subsumption relation does not hold between t_2 and t_1 (i.e., $t_2 \not\sqsubseteq_G t_1$), we say that task t_2 contains a missing notifier bug.

Definition 3.3.8 *(Correctness of Execution). Given a task graph G and an initial state $\sigma^0 = \bot$, an* FSMoVe *execution $b = \langle t^1, t^2 \dots t^n \rangle$ is correct, when*

- *the task t^i is correct on state σ^{i-1}, and when $(\sigma^i, r^i) = [\![t^i]\!]_{\sigma^{i-1}}$ the task t^{i+1} is also correct on state σ^i for $1 \leq i < n$.*
- *the execution does not manifest an ordering violation on state σ^i for $1 \leq i < n$.*
- *the execution does not manifest a missing notifier on state σ^i for $1 \leq i < n$.*

Definition 3.3.8 summarizes our approach for verifying executions modeled in FSMove. We begin with examining and evaluating tasks in the order they appear in an FSMove program according to the semantics of Figure 3.16. The initial state is $\sigma_0 = \bot$. Evaluating a task gives us a new state, and the set of files consumed and produced by the task. We then verify that the task is correct, that is, it does not contain any missing inputs or outputs, while we also check that the task does not conflict with any previous task based on the Definitions 3.3.6 and 3.3.7 used for checking for ordering violations and missing notifiers respectively. Finally, we use the fresh state to evaluate the next task and perform the same verification task.

3.3.4 Generating FSMoVe programs

Recall from Figure 3.13 that in order to model an execution in FSMove, our dynamic approach takes as input an *instrumented* script and monitors its execution. The goal of the instrumentation applied to the input scripts is to provide the execution boundaries of every task, and other

information coming from high-level specifications written in the corresponding DSLs (i.e., declared input / output files and dependencies). To do so, we place instrumentation points before and after the execution of each task (e.g., build task, Puppet resource), and augment their execution by calling special native functions. These functions take a string argument that either contains the information originated from the original definitions, or indicate when the execution of a task begins or ends. Then, our dynamic analysis identifies these calls, extracts their arguments to synthesize the specification of each FSMove task, and map the intermediate file operations to the corresponding task definition. Through monitoring these special native functions calls and all the other (low-level) file system operations (e.g., a call to `open`) that take place at runtime, we are able to build FSMove programs.

An example of native function calls inserted by our instrumentation are writes to the standard output. These calls are triggered by inserting simple `print` statements as part of the instrumentation. Consider again the build scripts of Figure 3.15a. When monitoring their base execution (no instrumentation is added), we observe the file system operations at lines 4–9 (see Figure 3.20) that reflect file copying. When instrumenting these scripts, we augment their execution by adding the native function calls at lines 1–3, before the execution of the task `target`, along with the function call at line 10 after file copying. These calls enables us to identify the points where the task `target` begins and ends (lines 1, 10), along with its input / output files (lines 2, 3). Our dynamic analysis detects and examines these calls, and finally produces the FSMove representation shown in Figure 3.15b. Note that without this instrumentation, we are unable to map the intermediate file system operations (lines 4–9) to the task they come from, and to build the specification of the currently executed FSMove task.

Based on this instrumentation, we extract task specifications while executing the provided scripts, and *not* through statically analyzing them. In particular, our approach leverages the following insight: to perform a sound execution, every build and configuration management system is aware of *all* the dependencies and input / output files of each task at runtime. For example, a build system can recognize all task dependencies to schedule the execution of a task in the correct order. Similarly, in an incremental build, a build system is aware of all declared file inputs to determine which tasks must be executed in response to some identified file updates. Our approach benefits from dynamically extracting this information from the execution engine of the underlying systems for two reasons. First, we do not have to perform static analysis (a challenging task as we discussed in 2.4.3) to extract this information from the given scripts. Second, we can recognize all dependencies and input / output files, including the ones computed dynamically and not explicitly mentioned in the scripts under test. As we show in Chapter 4, the full details regarding the instrumentation of build and configuration management scripts are implementation-specific and depend on the underlying system. In the same chapter we also discuss that implementing the instrumentation part requires little development effort.

Despite its simplicity, FSMove is expressive and allows us to model several OS features that are related to or affect the file system, including operations on file paths, symbolic links, working directories, process cloning, operations on file descriptors, and more. Below, we discuss how we express such operations in terms of FSMove.

Operations on paths: A system operation that works on paths is translated to ei-

```
binary: $(glob dir/*.c)
  cc -o binary dir/*.c
```

(a) glob in Make.

```
$source = glob(["dir/*.c"])
exec{"binary":
  command => "cc -o binary
      $source"
}
```

(b) glob in Puppet.

```
task binary (type: Exec) {
  inputs.files fileTree(dir: "dir", include: "*.c")
  commandLine "cc -o binary dir/*.c"
}
```

(c) glob in Gradle.

Figure 3.21: Examples of glob operations.

ther consume or produce operations, depending on its effect on the file system. For example, when a task creates a new directory through the mkdir("/dir") system call, we emit a produce("/dir") operation. Another example is when the script creates a hard link to an existing file by invoking the link("/source", "/target") system call. In this case, we yield two FSMove operations: consume("/source"), and produce("/target").

Similarly, we treat symbolic links as operations on file paths. We first track symbolic links, and the files they reference by monitoring the symlink-family system calls. Then, when we encounter an access to a symbolic link, we emit two consume constructs. The first consume statement concerns the symbolic link, while the second one is associated with the file pointed by the symbolic link.

Operations on file descriptors: When the system process creates a new file descriptor, we use the $\texttt{fd}_f = \ldots$ operation. For example, when creating a new file descriptor through opening a file open("/file") = 3, we emit $\texttt{fd}_3 =$ "/file". We do the same, when copying an existing file descriptor to a new one through the dup-family system calls. For instance, dup2(3, 4) turns into $\texttt{fd}_4 = \texttt{fd}_3$. Finally, closing a file descriptor leads to del operations.

Note that we do not need to model pipes, as file descriptors are tracked across system processes by modeling the clone and dup-family system calls.

Working directory: Each system process operates on a specific directory. In FSMove, we use a special file descriptor variable, namely $\texttt{fd}_0$ that points to the working directory of the current process. Whenever the working directory of a process changes (through the chdir system call), we emit a $\texttt{fd}_0 = \ldots$ operation to model this effect.

Relative paths: Some file system operations operate on relative paths. For example, the call to mkdir("dir") creates the directory dir inside the current working directory. We handle relative paths through the p at e expression. Specifically, we model the above example as produce("dir" at $\texttt{fd}_0$).

Forking processes: When a system process creates a new one (e.g., by calling the clone system call), we generate a newproc z_2 from z_1 statement, where z_2 refers to the id of the new process, while z_1 is the parent process. Finally, we model the main process that governs the execution using a newproc z statement.

Operations on directories: Beyond modeling working directories, our approach does not

make any other special treatment on directories. Operations on directories, such as indexing or globbing, are irrelevant and they do not affect the effectiveness of our approach.

To better illustrate this, consider the three equivalent Make, Puppet, and Gradle tasks shown in Figure 3.21. All tasks use system-specific functions to determine their input files through a `glob` pattern. In particular, the Make task (Figure 3.21a) uses the function `$(glob dir/*.c)` (line 1), the Puppet task (Figure 3.21b) invokes the function `glob(["dir/*.c"])` (line 1), while the Gradle task (Figure 3.21c) employs the function `fileTree` (line 2). By the time our approach runs the instrumented code to dump the declared inputs of each task to the standard output, the functions `glob` and `fileTree` have *already* been evaluated, and have resulted in a list of file paths that match the `glob` pattern `dir/*.c`. As a result, our approach is able to print all the *concrete* input files to the standard output, as we did in the example of Figure 3.20 (line 2). In a similar manner, the `glob` pattern passed as input to a shell command (e.g., `cc -o binary *.c`—line 2 in Figure 3.21a or line 3 in Figure 3.21c) also leads to a list of matching files which we capture and model through their individual accesses (i.e., file opens) while monitoring the execution (i.e., there is no a Unix system call operating on `glob` patterns).

3.3.5 Analyzing FSMoVe Programs & Detecting Bugs

After modeling an execution in an FSMoVe representation, our method performs a linear pass over the representation and produces two types of output. First, it generates the corresponding task graph, and second, it computes all file accesses that take place in every FSMoVe task based on the semantics presented in Figure 3.16.

In the final step of our method (bug detection), we verify the correctness of an FSMoVe execution based on the task graph and the file accesses computed in the analysis step. In particular, we examine the file accesses of every task t, and we proceed as follows. If a file access p is of type "consumed" ("produced"), and the subsumption relation (Section 3.3.3.1) between p and the file inputs (outputs) of t does not hold, we report a missing input (output) on p. For ordering violations, we check whether p was accessed elsewhere (say another task t') in the given FSMoVe program. If this is the case, we verify whether the execution order between t and t' is deterministic using the happens-before relation. If the happens-before relation between these tasks is undefined, we report an ordering violation. To identify missing notifiers, we proceed in a similar manner as ordering violations. In this case though, (1) we check that the task that consumes the file resource p corresponds to a `service` resource, and (2) we compare t and t' against the subsumption relation. Our bug detection approach eventually reports all file accesses that violate the correctness of the given script according to the definitions of Section 3.3.3.

The bugs related to parallelism manifest themselves non-deterministically: they appear depending on the execution schedule of the underlying execution engine. Also, the dependency bugs associated with incrementality do not appear in full builds. However, our technique is capable of detecting subtle and future latent bugs, because it does not require a build or a configuration management script to crash and then reason about the root cause of the failure.

3.3.6 False Negatives and False Positives

We discuss potential false positive and negative errors focusing on specific examples.

False Negatives. Monitoring and reasoning about a single execution (as our approach does) produces false negatives when incorrectly specified file accesses are not observed in the monitored script. For example, such erroneous file accesses may depend on the environment (e.g., OS, environment variables, parameters, and other external factors) where the build or configuration management execution takes place. If these condition criteria are not met under the monitored execution, our approach will be unable to identify the bug. As an example, consider the following Make script:

```
1 ifeq ($(LANG),en_US)
2   options = --input file_en_US.in
3 else
4   options = --input file.in
5 endif
6 target: file.in
7   cmd $(options) --output $@
```

This script defines a task (lines 6, 7) that executes a command (`cmd`). The input of this command depends on the current locale settings, i.e. `–input file_en_US.in` in the case of English (lines 1, 2), and `–input file.in` in any other case (lines 3, 4). When we run this build script in an environment where the locale settings are not set to English, a false negative occurs. Even though the script is incorrect, our approach misses the bug because it is unable to infer that in addition to `file.in` (line 6), the build task can also depend on the file `file_en_US.in`.

False negatives may also occur in the case of file system operations not modelled in FSMove. For example, our approach does not model the case when a process passes a file descriptor to another process via sockets (through an `SCM_RIGHTS` message).[1] In practice, it is highly unlikely for different tasks to communicate with each other in such a way. That said, we leave supporting this case as future work. Note that we can support the aforementioned operation without extending our FSMove model. Specifically, we can express the operation in terms of a $\texttt{fd}_f = p$ statement in the process that receives the message (i.e., file descriptor). In this scenario, f corresponds to the file descriptor that was passed via a socket message, while p is the file path pointed by the file descriptor f defined in the process that sent the message.

Similarly, not modelling the data blocks for read / write operations (e.g., `read` system call) may lead to an ordering violation issue not captured by our approach. Consider the case where a task must read from an existing file (not produced by the main process), after the contents of this file get modified by another task. If this ordering constraint is not specified by the developer, our approach will miss the issue because it does not track the contents read from and written to the file. Therefore, it cannot capture the fact that the first task may read a different value than the one written by the second one. In the context of the studied systems though (i.e., builds and IaC software), file modifications are not as common as file creations.

False Positives. We can identify the missing inputs / outputs of tasks without reporting false positives. This is justified by two reasons. First, by determining the exact execution

[1] https://www.man7.org/linux/man-pages/man7/unix.7.html

boundaries of every task, our approach correlates the files that are read and written with the corresponding tasks. When we detect an access on a file named `file.txt` during the execution of a specific task, this task indeed accesses `file.txt`. Second, the observed file accesses can be reliably verified against the specifications written by programers. Thanks to the instrumentation performed on the given scripts, we know *all* task inputs and outputs that are declared by developers. Hence, our approach never reports false positives caused due to incomplete knowledge of the declared dependencies.

However, we may report false alarms when dealing with ordering violations. Specifically, a false alarm occurs when there are two independent tasks that conflict with each other (i.e., they access the same file), but their execution order does not affect the correctness and the output of the script under test. Consider the next Gradle script.

```
1  task A {
2    f = file("file.txt")
3    doFirst { f.text = "value" }
4  }
5  task B {
6    f = file("file.txt")
7    doFirst { f.text = "value" }
8  }
```

There are two tasks (lines 1, 5) that produce the same file with the same contents (lines 3, 7). In this case, there is not an ordering violation. Any execution order of tasks `A` and `B` has the same effect: at the end of the build, we always end up with `file.txt` containing the text `"value"`. Although such benign cases were not observed in practice and usually indicate a flaw in the build scripts (e.g., unnecessary computation), someone can identify them by also checking the contents written to every file.

As another example of false positive, consider the following Puppet program.

```
1  $file = "/file"
2  file { $file:
3    owner => "root",
4    group => "root",
5    mode => "0600",
6  }
7  exec { "Generate secret file":
8    unless => "/bin/egrep '^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$'
         '${file}' >/dev/null",
9    command => "/bin/cp /proc/sys/kernel/random/uuid '${file}'",
10 }
```

In the above program, the developers use two different tasks to partially configure a certain file. On the one hand they use `file` to set the permissions and ownership of the file, and on the other they use `exec` to initialize its contents. In this case the execution order, in which Puppet processes these tasks does not matter. For example, Puppet can first use `exec` to create a file with the desired contents, and then it can apply `file` to set the appropriate file's attributes. Again, as we will see in Chapter 5, such behaviors (e.g., configuring a file through the combination of two tasks) is not particularly common.

Chapter 4

Implementation

This thesis is accompanied by three command-line tools that implement the techniques presented in Chapter 3. In this chapter, we provide the implementation details of these tools.

4.1 Hephaestus: A Framework for Testing Compilers' Type Checkers

To find compiler typing bugs, we have implemented the techniques of Section 3.1 as a command-line tool named Hephaestus. [1] Hephaestus contains roughly 15k lines of Python code, which breaks as follows: the code related to our IR is about 5k LoC, our program generator contains around 3k LoC, our analysis for building type graphs is 1k LoC, while our mutations include roughly 400 LoC.

Hephaestus provides a rich command-line interface, which among other things, allows users to (1) affect the characteristics of the generated programs (e.g., the size of programs, disabling or enabling certain features), (2) choose the compiler under test, or (3) disable or enable the implemented mutations. The complete command-line interface of Hephaestus is shown in Appendix B.

All mutations, analyses, and translators are implemented through the visitor design pattern [Palsberg and Jay, 1998]. In particular, each node in Hephaestus's IR contains the following two methods:

```
1  def accept(self, visitor):
2    return visitor.visit(self)
3
4  def children(self):
5    # The implementation is specific to each type of node
```

The `accept` method accepts a visitor and returns the result of visiting the current node `self` using the given visitor. The method `children` gives the children of the current node. For example, the children of a node that corresponds to a variable declaration of the form `var` $x : t = e$ is the expression corresponding to the variable initialization, that is, e.

[1] In Greek mythology, Hephaestus was the smithing god.

```
1  class BaseVisitor(object):
2    def visit(self, node):
3      visitors = {
4        ast.VariableDeclaration: visit_var_declaration,
5        ast.StringConstant: visit_string_constant,
6        ...
7      }
8      visitor = visitors.get(type(node))
9      if visitor is not None:
10       return visitor(node)
11     raise Exception("visitor for node {} was not found".format(node))
12
13   @abstractmethod
14   def visit_var_declaration(self, node):
15     pass
16
17   @abstractmethod
18   def visit_string_constant(self, node):
19     pass
20
21   ...
```

Figure 4.1: A sketch of the implementation of the base visitor class.

The implementation of a visitor (e.g., a mutation, a translator) is then responsible for implementing a `visit` method for every type of node in the IR. Each implementation visits nodes in a post-traversal order, meaning that it first visits children, and then the parent. For example, the implementation of each `visit` method looks like the following:

```
def visit(self, node):
  children_res = []
  for c in node.children():
    children_res.append(node.accept(self))
  # Now do something with the result of children 'children_res'
```

Since Python does not support method overloading for distinguishing every `visit` method based on the type of its parameter, the base visitor class (namely `BaseVisitor`) contains a `visit` method that *manually* dispatches the given node to the appropriate visitor method. As illustrated in Figure 4.1, the `visit` method of class `BaseVisitor` selects and invokes another visitor method (prefixed with "`visit_`"—see lines 3–6) based on the type of the given node (lines 3–10). For example, the visitor of a variable declaration node is given by the implementation of a method named `visit_var_declaration`.

Regarding the performance of Hephaestus, we observed that most of the testing time is spent on compiling the generated test programs. To mitigate this bottleneck, instead of generating and compiling one program at a time, Hephaestus generates and compiles programs in batches, where each batch contains a user-specified number of programs. All programs of each batch are then passed as input to the compiler under test. Note that to avoid conflicting declarations (e.g., two programs define classes of the same name), every program in a batch is placed in a dedicated package. To do so, the corresponding translator puts a `package` statement

at the beginning of each source file. Compiling programs in batches significantly boosts the performance of testing, as we avoid bootstrapping a JVM per generated program.

For even better throughput, Hephaestus generates and compiles programs using multiple system processes via the `multiprocessing` module of Python. In particular, Hephaestus creates a pool of workers using the `multiprocessing.Pool` class. The number of workers is given by the user using the `-workers` command-line option. Based on this pool of workers, the generation of every program is done in an *asynchronous* manner. This means that the execution of Hephaestus does not stop until a new program is generated. Instead, the execution proceeds with the next instructions and another worker (process) generates the program in background. Once Hephaestus generates all programs included in a batch, it invokes the compiler under test. Compilation is again done asynchronously via the same pool of workers.

When the compiler under test terminates its execution, Hephaestus sends the result of compilation back to the main process, which in turn, checks the validity of the compilation result based on the given oracle. For example, when the expected behavior of the compiler is to accept the generated program, but the compiler returns a non-zero exit code, Hephaestus analyzes the compiler output to determine whether the compiler under test crashed or produced an unexpected compile-time error. To do so, Hephaestus uses a regular expression to capture strings found in the compiler output that correspond to error messages generated by the compiler. For example, the regular expression used for identifying *all* compile-time errors produced by `javac` is given below:

```
ERROR_REGEX = re.compile(
  r"([a-zA-Z0-9\/_]+.java):(\d+:[ ]+error:[ ]+.*)(.*?(?=\n{1,}))')")
```

For every detected compile-time error, Hephaestus extracts the name of the file where the error resides, along with the error message itself. For example, consider a compiler output that contains only the following two error messages:

```
program5/Main.java:7: error: incompatible types: double cannot be converted to T
program25/Main.java:15: error: incompatible types: B<CAP#1> cannot be converted to B<?
    super T>
```

From the information found in the above strings, we know which of the programs included in our batch violate our test oracle: (1) the program that resides in file `program5/Main.java` and (2) the program of file `program25/Main.java`.

When the output of the compiler under test does not include any compile-time errors (e.g., a type mismatch error), Hephaestus uses another regular expression to see if the execution of the compiler produces a crash. If this is the case, Hephaestus marks all programs in the batch for further inspection, as it cannot identify which of them triggers the compiler crash.

Generalizability: Currently, our implementation, generates programs written in three popular languages: Java, Kotlin, and Groovy. However, our approach is not limited to JVM languages. Our techniques also apply to statically-typed, object-oriented languages that support type inference or parametric polymorphism/generics, such as Scala, C#, TypeScript, Swift, and more. Supporting a new target language requires little engineering effort. Specifically, two components are necessary: (1) a translator to convert a program written in IR into a concrete program written in the target language (existing translators contain roughly 800 LoC), and (2)

a regular expression that distinguishes compiler crashes from compiler diagnostic messages. For testing language-specific features (e.g., Scala implicits, pattern matching), HEPHAESTUS' IR must be extended as well.

Although our initial focus is to test the type checkers of statically-typed languages, HEPHAESTUS's program generator can also be used to produce programs written in dynamic languages (e.g., Python, JavaScript) by implementing the corresponding translators.

Necessity of program generation: A reader may wonder whether our program generator is actually necessary or not. Actually, our program generator is not necessary for using our mutations. That is, one can apply the mutations directly to existing code (e.g., compiler test suites). However, this would require a parser implementation for each target language because the mutations operate on our IR. Furthermore, as we will observe in our bug-finding results (Section 5.1.2, Figure 5.4), our program generator (1) is very efficient in discovering typing bugs on its own, and (2) improves the effectiveness of mutations by providing them with good seeds.

4.2 Cynthia: A Differential Testing Engine for ORM Implementations

We have implemented the data-oriented testing approach described in Section 3.2 as a command-line tool called CYNTHIA,[2] which consists of around 8.6k lines of Scala code.

The interface of CYNTHIA takes as input the names of the ORMs to test along with a set of DBMSs on which CYNTHIA runs the ORM code. Beyond that, CYNTHIA provides a rich set of command-line options and sub-commands used for a plethora of use cases: from testing ORMs, to debugging and inspecting CYNTHIA results. A detailed description of CYNTHIA's CLI can be found in Appendix C.

CYNTHIA's execution consists of one or more *testing sessions*. The precise number of testing sessions is given by the user. In every testing session, CYNTHIA processes a user-specified number of AQL queries associated with a (randomly-generated) database schemas. These AQL queries are either generated from scratch by CYNTHIA, or given by the user (see below about the "replay" mode). Then, CYNTHIA translates every AQL query into multiple ORM queries; we have one ORM query for every specified DBMS. Finally, CYNTHIA runs the derived ORM queries and compare their results, as explained in Section 3.2.4.

For efficiency, CYNTHIA processes testing sessions and ORM queries in parallel using Scala futures [Haller et al., 2020]. Specifically, CYNTHIA executes every testing session asynchronously by creating a new future. Every testing session retrieves a number of AQL queries from disk or generates them on-the-fly, and then processes them (i.e., translating them into ORM queries, running ORM queries) asynchronously, again, by using Scala futures. Notably, for memory efficiency, testing sessions derive AQL queries in a lazy fashion. This means that CYNTHIA does not load all the AQL queries of a testing session into the memory (as a testing session can handle a tremendous number of AQL queries). Instead, CYNTHIA generates or retrieves one AQL query at a time, and then creates a future to (1) map the query into ORM code,

[2] In Greek mythology, Cynthia was the epithet of Artemis, the goddess of the hunt.

and (2) run the resulting ORM code on top of all the underlying database engines. All database interactions are also done in parallel. For example, a query run on a PostgreSQL database is executed in parallel with a query run on an SQLite database.

During the generation process of AQL queries, CYNTHIA internally uses a singleton object named `RUtils` that is responsible for producing random constants (e.g., an integer, a word, a boolean) or performing other `random`-related operations (e.g., picking a random element from a list). Interestingly, CYNTHIA's `RUtils` object produces valid words by randomly selecting elements from an English dictionary. Optionally, CYNTHIA may also receive a random seed (i.e., a number) from the user to make the testing procedure deterministic. This is achieved by configuring the `RUtils` object using the given seed number. To prevent non-determinism when dealing with parallelism that stem from the use of Scala futures, each spawned thread has its own copy of the `RUtils` object. To do so, CYNTHIA makes the internal state of `RUtils` *thread-local* using the `java.lang.ThreadLocal` API.[3]

CYNTHIA also provides a *replay* mode, which is used to replay a testing session (i.e., repeat the execution of existing AQL queries) for either debugging purposes or experimenting with different settings (e.g., running existing queries on different DBMSs, translating existing queries into different ORM code). For example, consider the following two `replay` invocations of CYNTHIA:

```
# Invocation 1
cynthia replay \
  --orms django,peewee \
  --backends sqlite,postgres \
  --all

# Invocation 2
cynthia replay \
  --schema Cucumbers \
  --orms django,pewee \
  --backends mysql \
  --all
```

The first invocation replays all previously-created testing sessions by testing the Django and peewee ORMs on top of the SQLite and PostgreSQL databases. The second invocation replays only one testing session: that whose name is `Cucumbers`. This time, CYNTHIA tests the Django and peewee ORMs by running their code on a MySQL database. In both situations, CYNTHIA extracts the information associated with previously-executed testing sessions by examining a directory named `.cynthia`, which is located in the current working directory. This directory is used for storing all (intermediate) results stemming from a CYNTHIA run, including SQL scripts for setting databases, AQL queries, ORM code. We refer the reader to the Appendix C for more details about the structure of the `.cynthia` directory.

To generate database records, our tool uses the Z3 theorem prover [de Moura and Bjørner, 2008], configured with a user-specified timeout. In particular, CYNTHIA takes an AQL query and translates it into SMT formulae (Section 3.2.3) using the `com.microsoft.z3` Maven package,[4]

[3] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ThreadLocal.html
[4] https://mvnrepository.com/artifact/com.microsoft.z3

which provides a Java API for interacting with the Z3 solver. CYNTHIA then converts a model generated by the solver into executable SQL `INSERT` statements.

Regarding the implementation effort of ORM translators, each translator consists of roughly 300–400 lines of Scala code. Every translator traverses the AST of AQL queries and emits code that uses the API of the corresponding ORM. To do so, CYNTHIA adopts the visitor design pattern in a similar way to our HEPHAESTUS tool (Section 4.1). Adding a new CYNTHIA translator is guided by extending and implementing an abstract Scala class.

4.3 FSMoVe: A Tool for Detecting Bugs in System Configuration and Build Scripts

In this section, we discuss the implementation of a command-line tool that relies on the FSMOVE model introduced in Section 3.3. Our implementation is written in the OCaml programming language and consists of approximately 6.5k lines of code. To trace system operations, our implementation employs `strace` [McDougall et al., 2006], but this could be accomplished by using either other system call tracing utilities (e.g., `DTrace` [Rodriguez, 1986])) or dynamic binary instrumentation [Bruening et al., 2012; Nethercote and Seward, 2007].

Our tool parses the `strace` output and translates it into a FSMOVE representation. The implementation supports two modes: *offline*, and *online* (the reader is referred to Appendix D for more details about the command-line interface of FSMOVE). In the offline mode (which is primarily used for debugging purposes), our tool does not monitor script execution. Instead, it expects a file containing the `strace` output obtained from previous runs.

When in online mode, the tool generates and analyzes a FSMOVE program, while monitoring a build command or a configuration management command through `strace`. To do so, it creates two processes. The first process runs `strace` on the given command, while the second reads the `strace` output produced by the first process and runs the FSMOVE generation and analysis steps in a streaming fashion. Communication is done through pipes which allows processes to run concurrently. This avoids the I/O overhead of storing system call trace in a temporary file. Specifically, in Unix, there is a virtual device directory named `/dev/fd`, which contains virtual files that correspond to the file descriptors of each process. FSMOVE leverages this virtual device directory as follows. The process that executes the given command using `strace` writes the system call trace to file `/dev/fd/N`, where `N` stands for the file descriptor that represents the write endpoint of the process. Then, the second process parses and analyzes the trace output by reading the contents of `/dev/fd/N`.

Communicating via pipes eliminates the observable time spent on the analysis phase, because running a build or a Puppet script is much slower than the analysis of the corresponding FSMOVE programs. Therefore, in a multicore architecture, our tool exploits a spare core to perform the analysis, while the input script is running.

We have implemented our method with efficiency in mind. Our tool is able to handle GB-sized traces with reasonable time and space requirements (see Section 5.3.5). This was made possible through a number of optimizations, such as the use of streams to process and analyze

traces, a reversed inode table to lookup paths based on their inodes, function memoization, and tail recursion. For example, FSMOVE processes every system call trace in a lazy manner by taking advantage of the following data type and its operations:

```
1  type 'a stream =
2    | Stream of 'a * (unit -> 'a stream)
3    | Empty'
4
5  let next stream =
6    match stream with
7    | Stream (_, thunk) -> thunk ()
8    | Empty -> raise Empty_stream (* next is not defined on empty streams *)
9
10 let peek stream =
11   match stream with
12   | Stream (v, _) -> Some v
13   | Empty -> None
```

The data type `a' stream` represents either an empty stream (line 3) or a tuple consisting of the head element of a stream (`'a`) and a thunk (i.e., `unit -> 'a stream`) which gives the tail of the current stream when called. The function `next` (lines 5–8) returns the tail of the given stream (if stream is not empty), while `peek` (lines 10–13) gives the head of the input stream.

Using the above data type and operations, FSMOVE implements the following functions

```
1  val parse_trace : Unix.file_descr -> line stream
2
3  val analyze_program : line stream -> (task_graph * file_accesses)
```

The `parse_trace` function receives a file descriptor corresponding to the virtual file `/dev/fd/N` holding the execution trace, and produces a stream of lines, where each line represents an FSMOVE construct (see Figure 3.14). The `analyze_program` function receives a stream of lines, and produces an output that represents the set of file accesses of every FSMOVE task, along with the task graph (Figure 3.13). Every time `parse_trace` reads and parses a trace element, it creates a stream of the form:

```
Stream(parsed_line, () -> read_next_trace input)
```

This stream is then passed as input to `analyze_program`, which (1) gets the head element of the stream via the `peek` function, (2) analyzes that element, and (3) recursively analyzes the tail of the input stream by calling the `next` function. Calling `next` triggers the execution of the thunk (i.e., `() -> read_next_trace input`) associated with the stream. Notably, this thunk reads and parses the next element of the system call trace.

As discussed in Section 3.3.4, our FSMOVE approach requires monitoring the execution of an *instrumented* script. To instrument Gradle scripts, we have implemented a Gradle plugin written in Kotlin that hooks *before* and *after* the execution of every task as shown in Figure 4.2a (lines 1, 14 – irrelevant code is omitted). The plugin utilizes the Gradle API [Gradle Inc., 2020b] to print the following elements: (1) declared inputs / outputs of every task (lines 3–5, 6–8), (2) declared dependencies of every task (lines 9–11), and (3) execution boundaries of every task (lines 12, 16). This output is identified by our dynamic analysis and converted to FSMOVE tasks

```
1  // Runs BEFORE the execution of every task.
2  fun processTaskBegin(task: Task) {
3    task.inputs.files.forEach { input ->
4    println("#FSMoVe#: ${task.name} input ${input}")
5   }
6   task.outputs.files.forEach { output ->
7    println("#FSMoVe#: ${task.name} output ${output")
8   }
9   task.getTaskDependencies().forEach { d ->
10   println("#FSMoVe#: ${task.name} after ${d.name}")
11  }
12  println("#FSMoVe#: Begin ${task.name}")
13 }
14 // Runs AFTER the execution of every task.
15 fun processTaskEnd(task: Task) {
16  println("#FSMoVe#: End ${task.name}")
17 }
```

(a) Fragment of the instrumentation applied to Gradle builds.

```
1  target=$1
2  prereqs=$2
3  taskName=$(pwd):$target
4  echo "#FSMoVe#: Begin $taskName
       "
5  echo "#FSMoVe#: $taskName input
       $prereqs"
6  shift 2
7  /bin/bash "$@"
8  echo "#FSMoVe#: End $taskName"
```

(b) The `fsmake-shell` that instruments every Make rule.

Figure 4.2: The instrumentation implemented for Gradle and Make builds.

as explained in 3.3.4. To apply our plugin to a Gradle project, we modify Gradle scripts by inserting *only* six lines of code, as shown below:

```
1 buildscript {
2   dependencies {
3     classpath files ("/path/to/plugin/jar")
4   }
5 }
6 apply plugin: "our plugin id"
```

For instrumenting Make scripts, we created a shell script (`fsmake-shell`) that *wraps* the execution of every Make rule (Figure 4.2b). As with Gradle, this script prints the execution boundaries (lines 4, 8) and prerequisites of each task (lines 5). To achieve this, we override Make's built-in variable `$SHELL` to point to our script. After printing the necessary information, our script invokes the underlying shell to eventually execute the requested Make command (line 7). To handle Make dependencies generated at build time (e.g., through `gcc -MD`), we refine the task graph computed during the analysis phase by adding missing edges. To do so, we exploit information stemming from the Make database by running `make -pn` after each build. Note that we do not need to make any changes in the source code of Make scripts to enable tracing; we simply build projects by running

```
make "$@" -- SHELL='fsmake-shell '\''$@'\'' '\''$+'\''' '''
```

Finally, as to Puppet instrumentation, we simply run a Puppet script in debug mode by providing the Puppet execution engine with the `–evaltrace` and `–debug` command-line options. Thanks to the above command-line options, Puppet prints to the standard output an informative message when the application of a Puppet resource begins. For example, Figure 4.3 shows the trace produced by running the Puppet script of Figure 2.27 in debug mode. Observe the calls of `write` at lines 1, 6–7, 12 that correspond to messages printed to the standard output by

```
1  103 write(1, "Info: /Stage[main]/Exec[Initialize MySQL DB]: Starting to evaluate the
       resource", 80) = 80
2  103 execve("/usr/sbin/mysqld", ["/usr/sbin/mysqld", "--initialize"], ...) = 0
3  650 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
       child_tidptr=0x7f70159c39d0) = 660
4  660 open("/etc/mysql/my.cnf", O_RDONLY) = 3
5  660 read(3, "default content..."..., 44) = 44
6  103 write(1, "Info: /Stage[main]/Exec[Initialize MySQL DB]: Evaluated in 1.85 seconds",
        72) = 72
7  103 write(1, "Info: /Stage[main]/File[/etc/mysql/my.cnf]: Starting to evaluate the
       resource", 78) = 78
8  103 open("/etc/mysql/my.cnf20190128-32-15kba2r", O_RDWR|O_CREAT, 0600) = 7
9  103 write(7, "db settings..."..., 48) = 48
10 103 close(7) = 0
11 103 rename("/etc/mysql/my.cnf20190128-32-15kba2r", "/etc/mysql/my.cnf") = 0
12 103 write(1, "Info: /Stage[main]/File[/etc/mysql/my.cnf]: Evaluated in 0.06 seconds",
       70) = 70
```

Figure 4.3: An example of trace produced when running the Puppet script of Figure 2.27 in debug mode.

Puppet engine for debugging purposes. These messages indicate the points where the execution of each Puppet resource starts and ends respectively. Our framework exploits these points to classify system calls according to the Puppet resource they come from. For constructing the corresponding task graph, our implementation derives the compiled catalog (which is in a JSON format) that stems from the given Puppet script. Then, our implementation parses the computed catalog and examines the parameters of every Puppet resource. Specifically, given the parameters of a resource p, we create edges as follows.

- p has the parameter `"before":v`. This indicates that the resource p is applied before every resource included in the value of `"before"`. In this case, we add a $\overset{\text{before}}{\rightarrow}$ edge from p to every element of the list `v`.
- p has the parameter `"require":v`. This indicates that Puppet processes p after every element included in the value of `"require"`. Thus, we create a $\overset{\text{before}}{\rightarrow}$ edge from every element of `v` to p.
- p has the parameter `"notify":v`. The same as the `"before"` parameter, but this time, we also create an $\overset{\text{in}}{\rightarrow}$ edge from p to every element of the list `v`.
- `p` has the parameter `"subscribe":v`. The same as the `"require"` parameter, but this time, we also create an $\overset{\text{in}}{\rightarrow}$ edge that originates from every element of `v` to p.

Extending FSMoVe: Applying our method to a new build tool, IaC system, or programming language requires little development effort. Our Gradle plugin contains 90 lines of Kotlin code, `fsmake-shell` consists of *only 8 lines of shell code*, while our Puppet catalog analyzer includes 500 lines of OCaml code. To support a new system or technology, we need to implement an instrumentation that provides (1) the execution boundaries, (2) the declared inputs / outputs,

and (3) the declared dependencies of each task as we did for Gradle and Make. For example, to support the Scala's sbt build system, we need to implement a simple sbt plugin similar to that implemented for Gradle. This plugin will eventually provide us with hooks before and after the execution of every sbt task so that we can print their dependency information to the standard output.

Limitations: Currently, our tool can trace executions only in a Linux environment. However, extending our implementation to support monitoring in other platforms is straightforward. Also, `strace` introduces a $2\times$ times slowdown on script executions on average (see Section 5.3.5). Employing a tracing utility that runs in the kernel space may reduce the overhead [Celik et al., 2017]. Furthermore, non-deterministic executions (i.e., touching different files on different days) may lead to false negatives when a faulty file access does not happen the time when observing the given execution. However, unlike other approaches [Licker and Rice, 2019], our tool can cope with non-determinism occurred in subsequent executions (e.g., temporary files generated with random names), because it requires a single execution for performing the verification.

Chapter 5

Evaluation

Having introduced the techniques proposed in this thesis (Chapter 3) and some of their key implementation details (Chapter 4), we now evaluate them by answering a set of research questions. Our evaluation is multifaceted: each proposed technique and tool is evaluated in terms of several aspects, including bug-finding capability, importance and characteristics of bugs discovered, runtime performance, code coverage improvement, and if possible, comparison with state-of-the-art. In the following sections, we provide the experimental setup and the empirical findings that come from the evaluation of every bug-finding tool included in this thesis, that is Hephaestus (Section 5.1), cynthia (Section 5.2), and fsmove (Section 5.3).

5.1 RQ1: Evaluation of Hephaestus

The evaluation of Hephaestus is based on the following research questions:

RQ1.1 Is Hephaestus effective in finding typing bugs in JVM compilers? (Section 5.1.2)

RQ1.2 What are the characteristics of the discovered bugs and the bug-revealing test cases? (Section 5.1.3)

RQ1.3 Are the type erasure and type overwriting mutations effective in detecting type inference and soundness bugs respectively? (Section 5.1.4)

RQ1.4 Can Hephaestus improve code coverage? (Section 5.1.5)

To answer these questions, we used Hephaestus between February 2021 and mid-November 2021 to systematically test the selected JVM compilers. During this period, we ran Hephaestus for three months of CPU time, in total.
Result summary: Our key experimental results are

RQ1.1 Hephaestus *has found many bugs.* Within nine months of testing, Hephaestus has detected 170 bugs, of which 148 are confirmed, and 104 have been already fixed by developers. Interestingly, Hephaestus was able to find bugs in *all* the examined compilers: 126 bugs in `groovyc`, 33 bugs in `kotlinc`, and 11 bugs in `javac`.

RQ1.2 HEPHAESTUS *finds typing bugs.* HEPHAESTUS found 147 typing bugs, 2 parser/lexer bugs, and 7 back-end bugs. Most of these bugs are defects in the implementation of parametric polymorphism and type inference.

RQ1.3 *The type erasure and type overwriting mutations are effective in revealing type inference and soundness bugs.* TEM has found 56 type inference-related bugs, while TOM discovered 25 bugs. Notably, almost half of the found bugs (i.e., 48%) were triggered by our mutation techniques, meaning that the program generator could not catch these bugs by itself. Moreover, our mutations can exercise deep compiler code associated with type inference and other type-related operations, e.g., TEM has covered up to 5,431 more branches, and invoked up to 217 more functions, when compared with our program generator.

RQ1.4 Similarly to prior work [Yang et al., 2011; Livinskii et al., 2020], the incremental code coverage improvement due to HEPHAESTUS is small.

5.1.1 Experimental Setup

Hardware and compiler version: We performed all experiments in commodity servers (16 cores and 32 GB of RAM per machine) running Ubuntu 16.04 (x86_64). For our testing campaign, we used one server to test each compiler daily since February 2021. Note that our testing efforts were incremental, i.e. we concurrently developed HEPHAESTUS and tested the compilers. Hence, we have run HEPHAESTUS in its full capabilities only for one month. To avoid reporting previously known bugs, our efforts have focused on testing the *latest development* version of each compiler.

Baseline: There is no relevant baseline to which we could compare HEPHAESTUS. The fuzzer presented by Dewey et al. [2015], which detects bugs in the type-checker of Rust, is the closest related work. Still, their tool is language-specific and probably outdated. Stepanov et al. [2021] have developed a tool focusing on back-end crashes in the Kotlin compiler. However, their tool is not publicly available. Similarly, AFL [M. Zalewski, 2013] and the AFL compiler fuzzer [Groce et al., 2022] can only detect crashes. Therefore, these tools do not target bugs similar to those triggered by HEPHAESTUS.

Settings of test program generation: Producing very large programs can decrease throughput, as compilers require more time to process input programs and HEPHAESTUS's internal algorithms (e.g., type inference analysis) take longer to proceed. Based on our exploratory experiments, generating programs with up to ten top-level declarations, and generating expressions with depth up to seven give us a good balance between bug-finding results and performance. Other settings considered during our testing campaign include the maximum number of type parameters per parameterized class/function (three), the maximum number of local variable declarations (three), and the maximum number of parameters per method (two). Using the above settings, HEPHAESTUS results in programs consisting of 500–1000 LoC.

Test case reduction: HEPHAESTUS produces programs that trigger compiler bugs with three different manifestations (Section 2.2.3): unexpected-compile time error (UCTE), unex-

Table 5.1: Status of the reported bugs in `groovyc`, `kotlinc`, and `javac`

Status	groovyc	kotlinc	javac	Total
Reported	0	4	0	4
Confirmed	26	14	3	44
Fixed	93	10	2	104
Duplicate	4	3	1	8
Won't fix	3	2	5	10
Total	**126**	**33**	**11**	**170**

pected runtime behavior (URB), and compiler crashes. Unlike prior work [Yang et al., 2011; Le et al., 2014; Zhang et al., 2017; Livinskii et al., 2020; Stepanov et al., 2021], in most cases, test programs generated by HEPHAESTUS are easy to reduce. For UCTE, the compilers emit informative diagnostic messages that help us locate the expression that is responsible for the bug, and reduce the test case effortlessly. URB errors are an outcome of the type overwriting mutation. Henceforth, HEPHAESTUS logs the mutated program points; thus, we know precisely what line and instruction introduces the error. Nevertheless, minimizing programs that cause compiler crashes could benefit from an automated program reducer, such as C-Reduce [Regehr et al., 2012].

Interaction with compiler developers: We observed that some recurring bugs slowed down our testing efforts. Specifically, sometimes, HEPHAESTUS may produce programs that uncover the same bug. Consequently, other deeper bugs may be shadowed by a more common and shallow error. One of the reasons we designed our approach to be configurable is to mitigate this issue, even though disabling some specific features reduces the search space of HEPHAESTUS. Subsequently, it was important to interact with responsive development teams, as we could use their feedback to improve HEPHAESTUS. Groovy developers responded to most of our bug reports soon after reporting them, and typically patched easy-to-fix bugs within a week. Kotlin developers were also very responsive. Despite Kotlin developers being more interested in compiler crashes (as they fixed them immediately), they also answered other bug reports within a few days. For the OpenJDK's Java compiler, bug reports were verified within a week by developers. Unfortunately, OpenJDK's issue tracker is not open to the public. Although we tried to contact OpenJDK developers through email, we could not get any details beyond what is visible on their Jira deployment [OpenJDK, 2021]. Furthermore, we could not interact directly with the bug tracker and comment on the reports. Therefore, we focused our testing efforts on Groovy and Kotlin compilers.

5.1.2 RQ1.1: Bug-Finding Results

Table 5.1 summarizes the bugs we identified during our testing campaign. Overall, we reported 170 bugs. The developers confirmed most of them (148/170) as previously unknown, real bugs, while they have already fixed 104 bugs. This highlights the correctness and importance of the reported issues. As shown in Section 2.2.5, the relatively high number of unfixed

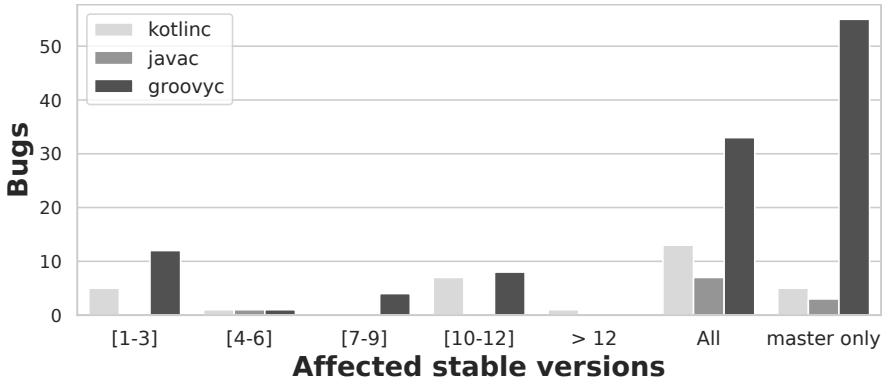


Figure 5.1: Number of bugs along with the number of stable versions they affect.

bugs could be attributed to the fact that some of the submitted bugs required much time to be resolved, as they are challenging and need careful examination. Notably, one compiler developer commented on our bug reports: *"The generics bugs are tough and so it was like working on a difficult crossword or Sudoku puzzle every day."*

Before submitting a new bug report, we always performed two steps. First, we waited for developers to fix existing bugs that may had the same root cause as the bug we wanted to report. Second, we searched in the issue trackers to find potential duplicate bugs. Overall, 8 out of 170 reported bugs were marked as duplicates. Specifically, two of them have already been opened by other users, and the other two had the same root causes with bugs we have already submitted.

Finally, only ten bugs were marked by developers as "not an issue" or "won't fix". Most of these "won't fix" issues are associated with cases where either the corresponding type inference engines are *underimplemented*, or there are decidability issues in the underlying type systems [Grigore, 2017; Pierce, 1992; Mackay et al., 2020]. For example, our five Java bugs marked as "not an issue" are programs where the inference engine of *javac* could not gather all the constraints it needed to get to a "solution". We discuss one such example in Section 5.1.6 (Figure 5.2f).

Importance of bug-finding results: In general, compiler developers welcomed our testing efforts and bug reports. A developer mentioned that: "*Thanks for your high-quality bug reports. I have been finding them quite complete in terms of recreating the issue. And the inclusion of variations that work as expected gives a nice basis of comparison when investigating the root cause*". Furthermore, we identified issues in fundamental compiler components. For example, seven out of 33 bugs in `kotlinc` were classified as *"major"* by developers. A `groovyc` developer commented on our reports: "*Static compilation and static type-checking is, for me, one of the most important features that must work with most other features of the language*". All the above demonstrate the practical implications of our testing efforts.

Affected compiler versions: We also ran the test cases that accompany our bug reports on all stable compiler versions. Figure 5.1 presents how many stable compiler versions are affected by the discovered bugs. It is clear that Hephaestus is able to find both *long-standing* and *regression* bugs. In particular, 33 `groovyc` and 13 `kotlinc` bugs occur in *all* stable compiler versions, while there is also a non-trivial number of bugs that affect numerous versions (i.e.,

Table 5.2: Number of bugs with unexpected compile-time error (UCTE), unexpected runtime behavior (URB), and crash symptom

Symptom	groovyc	kotlinc	javac	Total
UCTE	91	18	7	116
URB	20	3	0	23
Crash	15	12	4	31

10–12 affected versions). Such long-standing issues remain unnoticed for years. Also observe that a large portion of `groovyc` bugs (56/126—50%) are triggered *only* in the master branch of the compiler. These issues indicate that a feature that worked properly in previous versions, is broken in the current implementation `groovyc`. Regression bugs are often introduced by fixes of other bugs; their discovery reduces the friction and risk of development work.

5.1.3 RQ1.2: Bug and Test Case Characteristics

Overall, developers classified 161 out of our 170 (95%) discovered bugs, as bugs whose root cause reside in compilers' static typing procedures. We also identified seven back-end bugs. All of them were assertion failures or other runtime exceptions (e.g., `NullPointerException`) that were observed during code generation or optimizations, e.g., mishandling of bounded type parameters in code generation, erroneous optimizations corrupted the stack. Finally, Hephaestus detected two parser/lexer bugs.

Hephaestus detects typing bugs with three kinds of bug symptoms (Section 2.2.3): (1) errors that manifest when the compiler incorrectly rejects a well-formed program (*Unexpected Compile-Time Error – UCTE*), (2) *Unexpected Runtime Behavior (URB)* bugs[1] that occur when the compiler mistakenly accepts an ill-formed program, and (3) errors where the compiler terminates its execution abnormally (*Internal Compiler Error – Crash*).

Table 5.2 characterizes the bugs by their symptoms. Most of the discovered bugs (116/170) result in a UCTE, followed by 31 crashes, and 23 URBs. UCTE errors are triggered by well-formed programs produced by either our generator or the type erasure mutation. URB errors are an outcome of the type overwriting mutation that yields ill-typed programs. Finally, 26 crashes are caused by well-formed test cases, while four crashes are triggered by wrongly-typed code. The above results indicate that our approach enables the discovery of bugs with diverse manifestations (thus addressing "Challenge 1", Section 2.2.7).

We also identified what language features are involved in every minimized bug-revealing program that accompanies our bug reports. Table 5.3 lists the frequency of the top-10 most prevalent features supported by our generator and mutators. Features related to parametric polymorphism (e.g., parameterized class, bounded polymorphism) are in the list of features with the most bug-revealing capability. In total, 116/170 bugs are caused by programs containing at least one such feature. This confirms a comment by a compiler developer who wrote: "*generics are the feature with the most latent concerns*". Type inference is another category of features

[1]We followed the same terminology as Section 2.2.3, although it may be a bit misleading. Actually, our tool detects URB bugs at compile-time and not at runtime.

Table 5.3: Top-10 language features that appear in the minimized test cases of our reported bugs.

Feature	Category	Occ
Parameterized class	Parametric polymorphism	108
Parameterized type	Parametric polymorphism	86
Bounded type parameter	Parametric polymorphism	57
Type argument inference	Type inference	39
Lambda	Functional programming	38
Variable type inference	Type inference	25
Conditional	Standard language features	35
Inheritance	OOP features	33
Subtyping	Type system-related features	34
Function type	Functional programming	27
Parameterized function	Parametric polymorphism	28

Table 5.4: Bugs revealed by the generator, the type erasure mutation (TEM), the type overwriting mutation (TOM), and their combination (TEM & TOM)

Component	groovyc	kotlinc	javac	Total
Generator	63	17	7	87
TEM	41	12	3	56
TOM	21	3	1	25
TEM & TOM	1	1	0	2

that is hard to get right, as type inference features appear in 67 test cases. We further observed that some features are often combined with other individual features. For instance, in 47% of test cases that use conditionals, type inference features are also included. These results (1) validate our design decision to focus our efforts on parametric polymorphism and type inference (Section 3.1), and (2) are consistent with our bug study presented in Section 2.2.6 where we first pointed out the impact of specific language features on triggering typing bugs.

5.1.4 RQ1.3: Effectiveness of Mutations

The purpose of *type erasure* (TEM) and *type overwriting* (TOM) mutations is to test compiler components that the generator is unable to examine. It is worth mentioning that first, we implemented the generator, then TEM, and finally TOM. Specifically, we have been using our program generator since February 2021, whereas we finished implementing TEM in May 2021 and TOM in August 2021. Finally, we need to mention that beyond a few small-scale experiments, we have not tested the two mutations in combination.

Figure 5.4 shows the number of bugs triggered by the generator, the mutators, and their combination. Our mutations led to the identification of 83 out of 170 bugs, about half of the total discovered bugs. Our generator fails to detect these 83 bugs, as they are all related to either type

Table 5.5: Coverage increase by type erasure (TEM) and type overwriting (TOM) mutations.

Compiler		Line Coverage	Function Coverage	Branch Coverage
groovyc	Generator	42.68 %	41.77 %	42.07 %
	TEM change	+167 (0.46 %)	+27 (0.37 %)	+752 (0.45 %)
	TOM change	+99 (0.27 %)	+10 (0.14 %)	+447 (0.27 %)
	TEM `stc.*`	+106 (4.25%)	+13 (3.6%)	+531 (4.58%)
kotlinc	Generator	30.92 %	30.60 %	30.32 %
	TEM change	+787 (0.46 %)	+217 (0.39 %)	+5,431 (0.46 %)
	TOM change	+572 (0.33 %)	+166 (0.30 %)	+4,171 (0.35 %)
	TEM `resolve.calls.inference.*`	+238 (17.8%)	+63 (14.9%)	+1,865 (20.1%)
	TEM `resolve.*`	+572 (3.93%)	+135 (3.3%)	+4,086 (4.2%)
	TEM `types.*`	+147 (4.5%)	+69 (6.5%)	+957 (4.3%) %
javac	Generator	36.99 %	39.68 %	34.56 %
	TEM change	+396 (0.68 %)	+87 (0.81 %)	+2,150 (0.62 %)
	TOM change	+362 (0.62 %)	+79 (0.74 %)	+1,990 (0.57 %)
	TEM `comp.Resolve`	+100 (14.1%)	+27 (17.2%)	+613 (15.7%)
	TEM `comp.*`	+204 (2.67%)	+47 (3.6%)	+1,200 (3.1%)
	TEM `code.Types`	+113 (8.1%)	+23 (7.7%)	+ 558 (7.5%)
	TEM `code.*`	+131 (3.3%)	31 (3.2%)	636 (3.3%)

inference issues or other issues triggered by wrongly-typed code. TEM is an effective approach, able to identify 56 type inference bugs. This suggests that beyond compiler optimizations, type inference is another compiler procedure that can cause problems and deserves the attention of researchers. Finally, TOM either by itself or in combination with TEM has uncovered 27 bugs, of which 23 bugs are soundness issues. Detecting soundness bugs is of particular importance because such bugs can lead to unexpected runtime errors and security issues. Note that, even though soundness bugs may stem from either deep issues in the language or type system design (Section 2.2.5), all the discovered soundness bugs point to implementation-related errors.

We also conducted an experiment to estimate the impact of mutations on code coverage. To do so, (1) we instrumented each compiler using the JaCoCo code coverage library [EclEmma, 2021], (2) we generated 10k random programs via Hephaestus, (3) for each generated program, we produced two mutants using TEM and TOM respectively, and (4) we measured the code coverage increase that comes from compiling each mutant.

Figure 5.5 shows the results of this experiment. In all compilers, TEM and TOM increase line, function, and branch coverage when compared to our generator. In all cases, TEM is more effective in exercising new code than TOM. Also, `kotlinc` testing exhibits the most noticeable increase in terms of absolute numbers. For example, TEM covers 787 (0.46%) additional lines of code, triggers 5,431 (0.46%) additional branches, and calls 217 (0.39%) more functions.

At a first glance, the percentage increase may seem low (<1%). However, we should clarify that the goal of our mutations is to exercise the inference engines and other type-related operations, and *not* explore the entire compiler codebase. To validate this we further investigated the results. Indeed, when examining `kotlinc` results, we observe that TEM mostly exercises code in `resolve.*` and `types.*` packages, e.g., 204 / 217 (94%) of the additionally invoked functions belong to one of these packages. Specifically, these packages contain code responsible for

Table 5.6: Coverage on compilers' test suites plus 10K randomly-generated programs.

Compiler		Line Coverage	Function Coverage	Branch Coverage
groovyc	test suite	82.00 %	71.77 %	78.38 %
	test suite & random	82.06 %	71.79 %	78.44 %
	% change	+0.06 %	+0.02 %	+0.05 %
kotlinc	test suite	80.80 %	72.99 %	74.08 %
	test suite & random	80.83 %	73.05 %	74.11 %
	% change	+0.03 %	+0.06 %	+0.04 %
javac	test suite	83.76 %	83.95 %	83.90 %
	test suite & random	83.94 %	83.99 %	84.12 %
	% change	+0.18 %	+0.03 %	+0.22 %

inferring types and resolving method calls by building and solving a type constraint problem (e.g., see `resolve.calls.inference` package). In `groovyc`, TEM mostly covers code in the package responsible for static typing (namely, `stc.*`). Beyond type inference, this component performs many more operations. Thus, a 4% increase is significant. Finally, in `javac`, TEM exercises much code in the `code.*` and `comp.*` packages, which among other things, contain the implementation of (1) `javac`'s name resolution algorithm (`comp.Resolve`), and (2) type-related operations, such as type variable substitution (`code.Types`). Similarly, TOM mainly exercises code in the aforementioned packages.

The above results clearly suggest that our mutations can effectively find bugs through increased coverage of relevant compiler procedures, such type inference.

5.1.5 RQ1.4: Code Coverage

To answer this research question, we employed the JaCoCo code coverage tool. Specifically, we measured for each compiler the code coverage of its test suite, plus 10K programs produced by Hephaestus. Figure 5.6 summarizes our results. We observe that in all cases, the code coverage improvement is negligible. Nevertheless, Hephaestus is still able to trigger numerous bugs in all studied compilers. For example, although the line coverage improvement on `groovyc` is only +0.06 %, Hephaestus was able to find 126 `groovyc` bugs. Thus, we find that traditional code coverage metrics are too *shallow* to capture the efficacy of our approach (as also observed in testing optimizing compilers [Yang et al., 2011; Livinskii et al., 2020]).

5.1.6 Examples of Reduced, Bug-Triggering Programs

We discuss a selection of bugs discovered by Hephaestus. For space reasons, we present the manually minimized test cases.

Figure 5.2a: While type-checking the variable declaration on line 7, `groovyc` checks whether the call of the parameterized method `foo` returns a subtype of `C<String>`. The problem here is that due to a bug in its type inference algorithm, `groovyc` fails to infer the correct

```
class A {
 static <T> T foo(C<T> t) { ... }
}
class C<T> {}
class B {
 void test() {
  C<String> x = A.foo(new C<>())
 }
}
```

(a) GROOVY-10324: A bug in the inference engine of `groovyc` that leads to an UCTE.

```
class A<T: B<out Number>>(val x: T) {
 fun test() {
  val y: Int = x.m<C<out Number>>()
 }
}
class B<T> {
 fun <X: C<T>> m(): Int = 1
}
class C<T>
```

(b) KT-49101: A crash found in `kotlinc` due to a bug in type constructor projection.

```
class A<T> {
 T p
}

void test() {
 var x = new A<String>()
 var y = x.p
 x = null // changes inferred type
}
```

(c) GROOVY-10308: A bug in the flow typing algorithm of `groovyc` that causes an UCTE.

```
interface R<T>
interface W
interface J
open class A
open class B: A(), R<W>
open class E: A(), R<J>
open class C {
 open fun foo(): A = B()
}
class D: C() {
 override fun foo() = if (true) B()
    else E()
}
```

(d) KT-44082: `kotlinc` mistakenly approximates rejects this program.

```
class A{}
class B extends A{
 void m() {}
}
class Foo<T extends A> {
 T foo(T x) {
  // does not catch the error;
  x = new A();
 }
}
void test() {
 new Foo<B>().foo(new B()).m()
}
```

(e) GROOVY-10127: `groovyc` does not catch the error at line 8. This results in a URB.

```
class A<T extends Double,
        K extends T> {

 public T test() {
  T foo = (T) null;

  final var v = ((true) ? foo :
   (K) null);

  return v;
 }

}
```

(f) JDK-8269348: `javac` infers the type of `v` as `double`. This leads to an UCTE.

```
public class Test {
  void test() {
    def closure = {
      new B<>(new A<Long>());
    }
    A<Long> x = closure().f
  }
}
class A<T> {}
class B<T> {
  T f;
  B(T f) { this.f = f; }
}
```

(g) GROOVY-10080: This well-typed program is rejected by the Groovy compiler.

```
fun <T1: Number> foo(x: T1) {}

fun <T2: String> bar(): T2 {
  return "" as T2
}

fun test() {
 foo(bar())
}
```

(h) KT-48765: This ill-typed program is compiled by the Kotlin compiler.

Figure 5.2: Sample test programs that trigger typing bugs.

type for instantiating type variable `T` of function `foo`. Consequently, `groovyc` infers the return type of `foo` as `Object` instead of `C<String>`. This bug was found by TEM.

Figure 5.2b: This program triggers a bug in the Kotlin compiler that leads to a compiler crash. The program defines a parameterized class `B` that contains a parameterized function `m`, which in turn declares a bounded type parameter `X`. When calling method `m` at line 3, we instantiate it with `C<out number>` as the type argument (note that `out Number` is the equivalent of `? extends Number` in the Java world). The compiler then tries to compute the captured type for `X` but it crashes due to a missing condition in the implementation of type capturing. This bug was found by TOM.

Figure 5.2c: Groovy supports flow typing as an extension of type inference. The idea behind flow typing is that the compiler infers types of variables based on the flow of a program. In this program, the type of variable `x` is `A<String>` at line 6 and `Object` at line 8. Nevertheless, when `groovyc` type checks line 7, it erroneously uses `Object` as the declared type of `x`, thereby rejecting the program. This bug was found by TEM.

Figure 5.2d: In this example, the inferred type of the expression `if (true) B() else E()` is the intersection type `A & R<out Any>`. Since `kotlinc` uses intersection types only internally, the intersection type is transformed into a type that is representable in the program. In this context, `kotlinc` first approximates this intersection type to type `Any` (`Any` is the counterpart of `Object`), and then checks it against the return type of the overridden method, which is `A`. This makes the compiler reject that program, as `Any` is not compatible with `A`. Notably, in the discussion of this bug report, the product owner of Kotlin suggests that the compiler *"should have checked that* `A & R<out Any>`*, is a subtype of* `A`*, and then have computed the approximation of their intersection to derive* `A` *as the return type of the overriding method* `foo`*"*. This bug was found by TEM.

Figure 5.2e: `groovyc` erroneously accepts the program of Figure 5.2e, and produces a binary that breaks the type safety of the language. In particular, the compiler should have reported a compile-time error at line 8, which would indicate that `A` cannot be converted to type `T` (as `A` is not a subtype of `T`). At runtime, this incorrect compilation leads to a `MissingMethodExcepiton`, when executing the call at line 12. This bug was found by TOM.

Figure 5.2f: This program presents a "wont'fix" `javac` issue. Although the least upper bound of the conditional (line 7) is type `T`, the compiler infers the type of local variable `v` as type `double`. This in turn causes a type mismatch as a `double` cannot be converted to type `T` (see line 7). A Java developer commented that type inference is not possible in this case, as the target variable `v` does not contain all required constraints to compute an "optimal" solution. Using `T extends Double` (line 1) is the cause of this issue, as using `T extends Number` or any other type leads to a successful compilation. Beyond that, replacing the expression at line 7 with `(true) ? (T) null : (K) null` results in a correct compilation. All the above suggest that this is a broader issue in `javac`'s type inference algorithm design and implementation; this issue was found by TEM.

Figure 5.2g: This bug had affected `groovyc` since version 2.0.0. For almost a decade (from December 2011 until May 2021 when we reported it), this bug had slipped the thorough testing efforts applied by the Groovy development team. Notably, this long-latent issue was resolved

within days after reporting it.

The program declares two parameterized classes, namely `A` and `B`. Class `B` defines a field whose type is given by the type parameter `T`. On line 3, the code declares a lambda that returns an object of type `B<A<Long>>`. Although the type argument of `B` is omitted on line 3 (via the diamond operator `<>`), the compiler should be able to infer the corresponding type parameter from the type of the constructor's argument, which is `A<Long>`. However, a type inference bug causes the compiler to report a type mismatch on line 4. `groovyc` incorrectly infers the type of `closure().f` as `Object` instead of `B<A<Long>>`. Surprisingly, replacing `A<Long>` with `Long` at line 3 successfully compiles the program. This bug was found by TEM.

Figure 5.2h: The development and the stable versions of `kotlinc` fail to detect a type error in this ill-typed program. This bug is a regression introduced by a major refactoring in the type inference algorithm of Kotlin, shipped with version 1.4, which appeared in August 2020. The bug remained undetected until we reported it in September 2021, and was classified as *"major"* by developers.

The program defines two parameterized functions: `foo` and `bar`. The first function declares a type parameter `T1` bounded by `Number`, while the second one introduces `T2` bounded by `String`. When calling `bar` at line 4, `kotlinc` instantiates it as `Number => Number`, because the return value of `bar` flows to a parameter whose type is bounded by `Number`. However, this type substitution is not valid, as `T2` cannot be a `Number`. Hence, instead of accepting the program, `kotlinc` should have raised a type error of the form: *" type parameter bound for T2 is not satisfied: inferred type Number is not a subtype of String"*.

All the bug examples presented in this section demonstrate that both well-typed and ill-typed programs can uncover typing bugs. Furthermore, the bug-revealing programs combine multiple language features, e.g., mix of parametric polymorphism, lambdas, type inference, etc. Finally, both examples highlight that the process of static typing is hard to get right.

5.2 RQ2: Evaluation of Cynthia

In this section, we evaluate CYNTHIA by answering the following research questions.

RQ2.1 Is CYNTHIA effective in finding new bugs in established ORM systems? (Section 5.2.2)

RQ2.2 What are the characteristics of the bugs discovered by CYNTHIA? (Section 5.2.3)

RQ2.3 Is solver-based approach effective in generating appropriate data for differential testing? (Section 5.2.4)

Result summary: Our key experimental results are:

RQ2.1 CYNTHIA *is effective in finding bugs in all the examined ORM systems.* CYNTHIA has found 28 bugs, 20 of which, have been fixed by developers.

RQ2.2 *Most of the bugs detected by* CYNTHIA *are logic errors.* 17/28 bugs are DBMS-independent, meaning that they are triggered irrespective of the underlying database engine.

Table 5.7: The ORM systems examined in our evaluation.

ORM	Language	LoC(k)	Stars(k)	Used By(k)
ActiveRecord (Rails)	Ruby	49.2	46.2	1400
Django	Python	37.7	51.3	466
Sequelize	JavaScript	25.3	22.6	211
SQLAlchemy	Python	150	2.6	182
peewee	Python	7.6	7.7	10

RQ2.3 *Solver-based data generation improves the effectiveness of differential testing.* Compared to a naive approach, solver-based data generation reduces the number of cases where ORMs return empty results by 77%, on average. In terms of bug-finding results, differential testing underpinned by a naive data generation method fails to reproduce three ORM bugs.

5.2.1 Experimental Setup

Target ORM systems: We applied CYNTHIA to the five ORM systems listed in Table 5.7. We selected these ORMs based on the following criteria:

- *Usage*: the ORM should be established and widely-used.
- *High-level Logic*: the ORM should expose a high-level API that abstracts SQL-specific details.
- *Automation*: the ORM must provide tools for easy setup and utilities for generating model classes (recall Section 3.2.1).

According to the Github's statistics, all ORMs incorporated in our evaluation are used by millions of applications. For example, ActiveRecord, which is part of the Rails web framework, is employed by more than 1400k Github repositories. Further, many popular applications and services rely on them. For example, "Nova", OpenStack's cloud computing service, uses SQLAlchemy for interacting with the database. Finally, exposing high-level APIs from programmers can be prone to bugs / errors [Atlidakis et al., 2016]. Notably, Django, which provides the most expressive API, has the most bugs as we will see later.

DBMS: We ran the ORM queries on four DBMSs: SQLite, MySQL, PostgreSQL, and Microsoft's SQL Server (MSSQL). The first three DBMSs are extensively used by the open-source community and are supported by all the examined ORMs. Although MSSQL is supported by a subset of ORMs (i.e., Django, SQLAlchemy, Sequelize), we selected it because is one of the most popular proprietary DBMSs.

Cynthia configuration: We ran CYNTHIA on a regular basis, and tested the "master" version of the selected ORMs. In each run, CYNTHIA generated five random schemas. Each schema contained at least five tables, and each table included at least seven columns. After setting up the databases, CYNTHIA spawned a testing session, and processed each testing session separately until a specific timeout was reached (eight hours). For every query, Z3 produced 5

Table 5.8: Bugs detected by CYNTHIA.

ORM	Total	Fixed	Confirmed	Unconfirmed
Django	10	6	3	1
SQLAlchemy	8	8	0	0
Sequelize	5	2	1	2
peewee	4	4	0	0
ActiveRecord	1	0	1	0
Total	**28**	**20**	**5**	**3**

Table 5.9: The types of the detected bugs and the DBMSs where the bugs manifest appear. The column "All DBMS" shows the number of bugs that happen regardless of the underlying database engine.

Type	#Bugs	All DBMS	SQLite	MySQL	PostgreSQL	MSSQL
Logic Error	12	11	0	0	0	1
Invalid SQL	11	3	1	3	2	3
Crash	5	3	0	0	2	0
Total	**28**	**17**	**1**	**3**	**4**	**4**

records, while we set the solver timeout to 5 seconds. After each run, we manually inspected the reported mismatches for new bugs, and report them to the developers.

5.2.2 RQ2.1: Bug-Finding Results

CYNTHIA found 28 bugs in total, out of which, 20 were fixed by the developers, 5 were confirmed but are not yet fixed, 3 are still unconfirmed, while one confirmed bug in Django was previously known and marked as duplicate. Table 5.8 summarizes the bug detection results. Django is the system where we detected the most bugs (10). We attribute this to the fact that Django provides an API at a higher-level of abstraction compared to the other ORM systems. Interestingly, Django APIs hide every SQL-specific detail from the programmer, therefore translating a Django API call into a corresponding SQL construct becomes more challenging [Atlidakis et al., 2016]. Django is followed by SQLAlchemy (8), Sequelize (5), peewee (4), and finally ActiveRecord (1).

Importance of bug-finding results: 71% (20 / 28) of the reported bugs have already been fixed by the developers demonstrating the correctness and importance of the reported issues. We were particularly impressed by the prompt fixes of SQLAlchemy and peewee developers: they fixed most of the bugs within six hours after our bug report. Furthermore, three Django bugs were marked as release blockers by the corresponding developers. The above demonstrate that our bug reports were important and improved the reliability of the corresponding projects.

Regression bugs: Running CYNTHIA on the master version of ORMs enabled us to find a couple of interesting regression bugs. Regression bugs indicate that a feature that worked properly in previous versions, is broken in the current implementation. These bugs were of

paramount importance for the developers. For example, Django developers marked our regression bugs as release blockers. Also, SQLAlchemy developers commented: *"it's very useful if you are in fact alpha testing it."* (i.e., master branch). We also noticed that some bugs that were allegedly fixed were triggered again by new queries. This observation was confirmed by the developers, who, indeed reopened and fixed old bugs reported by us.

Remark on ORMs: Although it is the de-facto framework for Python, the Django ORM is the system where our approach detected the most bugs. One may wonder why we detected so many bugs in Django, while we uncovered only one bug in ActiveRecord. The reason is that Django is a more high-level ORM than ActiveRecord: it hides *every* single SQL-detail via its API. On the other hand, ActiveRecord's API provides some functionalities that are closer to SQL. For example, ActiveRecord supports arithmetic operations and aliasing by writing plain SQL. Thus, ActiveRecord does not employ any sophisticated translation mechanism and in many cases, the input of the programmer is passed directly to the SQL code.

Remark on DBMSs: CYNTHIA identified four PostgreSQL- and MSSQL-related bugs. These ORM bugs are triggered only when the DBMS is switched to PostgreSQL or MSSQL. On the other hand, only one ORM bug is related to SQLite. This happens because PostgreSQL and MSSQL are much stricter than SQLite (and even MySQL). For example, unlike MySQL and SQLite, PostgreSQL has a strict type system, and comes with many restrictions that ORMs need to take into account when producing SQL code. Also, we note that during our testing efforts, we discovered one bug in SQLite. The bug was already known and fixed in a later version of SQLite though. This implies that with some tuning, our approach may be also useful for testing DBMSs.

5.2.3 RQ2.2: Characteristics of Discovered Bugs

Recall from Section 2.3 that ORM bugs span three categories. The first category (*invalid SQL*) contains cases where an ORM yielded either a grammatically or semantically invalid SQL query. The second category (*internal ORM errors — or crashes*) contains cases where an ORM crashed unexpectedly, without even producing an SQL query. The third category (*logic errors*) contains cases where an ORM produced a grammatically and semantically valid SQL query, but this query did not fetch the right data from the database. Most of the discovered bugs (12) were logic ones (Table 5.9). Unlike differential testing, a naive fuzzing technique is unable to identify such bugs due to the test oracle problem. In a significant number of cases (11), the ORM generated an invalid SQL query, while the remaining cases (5) are related to crashes.

Table 5.9 also presents how many bugs are DBMS-dependent. Almost all logic errors (11 / 12) are DBMS-independent, i.e., they appear regardless of the underlying DBMS. By contrast, the majority of "Invalid SQL" bugs are DBMS-dependent. For example, two instances of "Invalid SQL" bugs happen when the code operates on PostgreSQL. This means the corresponding ORM failed to produce a legal PostgreSQL SQL query. Overall, 17 / 28 of the reported bugs are DBMS-independent. Yet, there is a large number of DBMS-dependent bugs (11 / 28). This validates our intuition to test ORMs across multiple database engines.

Based on the feature that ORMs fail to handle correctly, we further classify the discovered

```
1 Comment.new(:rating => 4)
2 Comment.new(:rating => 4)
3 # It incorrectly applies
      AVG to duplicate
      records.
4 Comment.select("comments.
      rating").distinct.
      average("comments.
      rating")
```

(a) A bug in ActiveRecord associated with DISTINCT.

```
1 // WHERE Comment.text
      LIKE '%_%'
2 Comment.findAll({
3  where: {
4   text: {[Op.substring]:
        "_"}
5  })
6 })
```

(b) A buggy Sequelize query associated with incorrect string comparison.

```
1 cons = ExpressionWrapper(
2   Value(3),...)
3 # GROUP BY Comment.text, 3
4 Comment.objects\
5   .annotate(cons=cons)\
6   .values("cons", "text")\
7   .annotate(sum=Sum("rating
      "))
```

(c) A buggy Django query associated with GROUP BY.

Figure 5.3: A collection of bugs discovered by CYNTHIA.

bugs into six categories.

Expression-related bugs. Expression-related bugs are the most common ones (7/28). This category involves cases where ORMs fail to produce an SQL expression that respects the original ORM query. As an example of this category, consider the peewee bug (Figure 2.24) discussed in Section 2.3. In this bug, peewee produces an SQL expression (i.e., $1 + col * 1 + col$) that is *not* equivalent with the high-level peewee expression written by the programmer (see Figure 2.24, lines 1, 2).

Distinct-related bugs. DISTINCT is a keyword in SQL that when present, it removes all duplicate records from the result set. ORM systems expose this functionality through a simple method call (typically called `distinct()`). Although the use of this feature looks simple, we detected six bugs related to this functionality. Figure 5.3a shows a buggy query in ActiveRecord associated with DISTINCT. The intended functionality of this query is to fetch all the records of the table "Comments", remove the duplicates, and then apply AVG to a column named "rating". However, ActiveRecord produces an SQL query that ignores the call of `distinct`, and therefore, it applies AVG to the entire set of records.

Combined-query-related bugs. SQL supports the combination of individual queries using the UNION and INTERSECT keywords. ORM systems support this feature by implementing the `union` and `intersect` methods. Five bugs discovered by CYNTHIA are associated with this functionality of ORMs. An example of this category of bugs has been already discussed in Section 2.3 (Recall Figure 2.22). In particular, Django is unable to produce a valid sequence of UNION operations when using MySQL as the database engine.

String-comparison-related bugs. String comparisons in SQL are typically done via the LIKE operator (or ILIKE for case-insensitive comparisons). These operators expect a pattern which SQL matches the value of a string against. There are two characters (namely '%' and '_') that have special semantics when used as part of a LIKE pattern. For example, '%' is a wildcard character that matches any sequence of characters. ORMs typically abstract LIKE with high-level methods, such as `contains()`. ORMs must escape the aforementioned characters when passed as an argument to these methods. We found four cases where ORMs fail to escape these characters leading to wrong string comparisons in the SQL part.

Consider Figure 5.3b that presents a bug in Sequelize. The Sequelize query shown in this figure attempts to fetch the records of "Comments" where the column "text" contains the char-

acter "_". Sequelize produces the SQL condition shown on line 1. Although the character "_" has a special meaning (it matches every single character), Sequelize does not escape it. As a result, the generated SQL query incorrectly retrieves all records of the table.

Aliasing-related bugs. SQL allows column aliasing through the `AS` construct. ORM systems also support aliasing. CYNTHIA uncovered four bugs where the corresponding ORMs either do not construct the alias correctly, or do not make a reference to a legal alias.

As an example of an aliasing-related bug, consider the SQLAlchemy query: `session.query( Model.column.label("exists"))`. When running this query on SQLite, SQLAlchemy generates the following SQL code: `SELECT "model"."column" AS exists FROM model`. Unfortunately, this SQL query is invalid because "`exists`" is a reserved keyword in SQLite. As a result, the execution of this query throws an *"sqlite3.OperationalError: near "exists": syntax error"* message. To fix this bug, the developers of SQLAlchemy wrapped the reserved word with quotes (i.e., `AS "exists"`).

Group-by-related bugs. The `GROUP BY` clause is used when selecting or referencing a table's column together with aggregated data. Many ORMs hide the `GROUP BY` clause from the programmer and handle this SQL feature behind the scenes. In this context, when a table's column and a aggregate function appear in the `SELECT` clause, ORMs emit a `GROUP BY` clause containing all columns that are not part of an aggregate function. `GROUP BY` comes with some caveats that ORMs need to consider in order to properly handle this SQL feature. We ran into three bugs caused by incorrect handling of the `GROUP BY` functionality.

Consider the Django query shown in Figure 5.3c. Django builds three expressions: the integer constant 3 (lines 1, 5), a reference to the column "text", and an aggregate function `SUM` applied to the column "rating". Django places all non-aggregate expressions on `GROUP BY` as shown on line 3. Integer constants have special semantics when they are part of `GROUP BY`. For example, `GROUP BY 3` means to group by the third expression of the `SELECT` clause of the query (i.e., `SUM("rating")`). This makes the generated SQL query invalid, leading to *"ProgrammingError: aggregate functions are not allowed in GROUP BY"*. The developers fixed this by ignoring constant expressions from the set of grouping fields.

5.2.4 RQ2.3: Effectiveness of Solver-Based Data Generation

For effectively identifying mismatches between the outputs of ORMs, it is important that ORMs return non-empty results for the given queries. Empty results indicate that the corresponding query was *unsatisfied* with respect to the data inserted to the database. Empty results can potentially hide logic errors that otherwise would be uncovered if the corresponding ORMs could get some data from the database and we were able to notice differences in their results.

To demonstrate the effectiveness of our solver-based data generation approach and its suitability for differential testing, we compare it against a simplistic approach that populates the database with random records *a-priori* [Rigger and Su, 2020c,a], i.e., it inserts data while setting up the tables, without considering the constraints of the generated queries.

We used CYNTHIA to spawn 20 testing sessions. For each testing session, we generated 100 queries and compared the results of ORMs as usual. At the end of each testing session, we

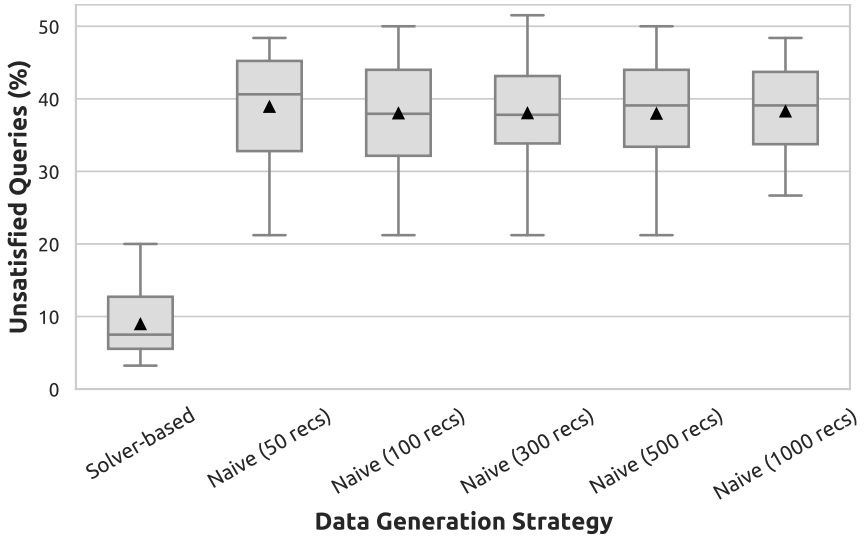


Figure 5.4: Percentage of the unsatisfied queries per data generation strategy using a sample of 20 testing sessions.

measured in how many queries the ORMs returned empty results. We then *replayed* each testing session, using a naive data generation strategy, and tried out different settings: generating 50 random records, 100, 300, 500 and finally 1000.

Figure 5.4 illustrates the comparison results. The y-axis shows the percentage of the unsatisfied queries. Every box plot contains the observations taken from the 20 testing sessions, along with the median (horizontal line), the mean (black triangle), and the maximum and minimum values. The solver-based approach leads to significantly fewer unsatisfied queries (median: 7.5%, mean: 8.9%) than the naive approach (mean and median values are roughly 38% for all the different settings). The reason why there is still a number of unsatisfied queries even with the solver-based approach is because either the corresponding AQL query was unsatisfiable or the solver timed out. Regarding the naive data generation, increasing the number of the records inserted to the database does not improve the effectiveness of this method at all, i.e., generating 50 records is almost identical to generating 1000 records.

We also tried to reproduce the discovered bugs using the naive data generation strategy. This strategy missed 3 out of the 12 logic errors previously detected by CYNTHIA, because it failed to generate appropriate data for the database. In these cases, the differential testing was meaningless, as the ORMs returned empty results. We did not consider the rest categories (e.g., invalid SQL), as in these cases the corresponding ORMs produce an error message regardless of the data stored in the database. Overall, our findings suggest that it is the quality of the inserted data that matters, and not the quantity: it is better to produce 5 targeted records than 1000 random records.

5.3 RQ3: Evaluation of FSMoVe

We evaluate the FSMOVE framework by answering five research questions:

RQ3.1 What is the effectiveness of FSMOVE in locating bugs in build and Puppet scripts? (Sec-

tion 5.3.2)

RQ3.2 What is the perception of developers regarding the detected bugs? (Section 5.3.3)

RQ3.3 What are the main bug patterns? (Section 5.3.4)

RQ3.4 What is the performance of FSMOVE? (Section 5.3.5)

RQ3.5 How does FSMOVE perform with regards to other tools, i.e., mkcheck? (Section 5.3.6)

Result summary: Our key experimental results are:

RQ3.1 FSMOVE *is effective in finding issues in both Puppet and build scripts.* FSMOVE has found plenty of dependency bugs in scripts associated with all the examined systems (Puppet, Gradle and Make). In summary, it has found 92, 237, and 15,754 issues in Puppet, Gradle, and Make scripts respectively.

RQ3.2 FSMOVE *reports bugs that are worth fixing.* Based on bug reports produced by FSMOVE, we submitted patches to the development teams. More than half of the submitted patches have been accepted: 25 out 33 (76%) for Puppet scripts, and 47 out of 71 (66%) for build scripts. These patches resolved 300 dependency bugs in total.

RQ3.3 FSMOVE *finds dependency bugs with diverse symptoms, implications, and root causes.* Most of the discovered bugs cause runtime failures, inconsistent states, and creation of stale artifacts.

RQ3.4 FSMOVE *imposes a* $2\times$ *slowdown on the execution of the given scripts because of system call tracing.* The bug detection phase employed by FSMOVE is every efficient: it takes < 1 second, on average.

RQ3.5 FSMOVE *outperforms state-of-the-art (mkcheck) both in terms of bug-finding capability and running times.* In particular, FSMOVE is $74\times$ faster than mkcheck, on average.

5.3.1 Experimental Setup

Selecting Puppet scripts: We collected a large number of Puppet modules taken from Forge and Github. We were particularly interested in non-deprecated modules that support Debian Stretch, because Debian is one of the most popular Linux distributions [Adams et al., 2016]. We inspected the top-1000 modules returned by Forge that satisfied our search criteria. We automatically ran every module separately through the `include <module-name>` statement.[2] We monitored the Puppet process and collected the system call trace of every program via `strace`. Through this process, we successfully applied 354 Puppet modules in total. The remaining modules failed because they required extra arguments or further setup before their invocation. For example, many of the failing modules required multiple pre-installed packages, or in other cases we needed to infer specific values for the modules' arguments including IPs of DNS servers and

[2] `include` applies all the resources defined in the module using the default settings.

URLs of specific upstream directories. Note that the failing modules and the successful ones were pretty similar in terms code size, popularity, and age in Puppet Forge. Finally, for every Puppet module that succeeded, we logged the reports generated by FSMOVE.

Selecting build scripts: We applied our approach to a large set of Gradle and Make projects. To identify interesting Gradle projects, we employed the Github API[3] to search for popular Java, Kotlin, and Groovy repositories that use Gradle. We selected 200 projects for each language (i.e., 600 projects in total) ordered by the number of stars. For every project, we performed the following steps. First, we instrumented the Gradle scripts as described in Section 4.3. Then, we ran the instrumented Gradle scripts through the `gradle build` command, which is the de-facto command for building Gradle projects. Note that this command executes the compilation, assembling and testing tasks as well as other user-defined tasks. For efficiency, we ran our tool in online mode (Section 4.3). In the end, we successfully analyzed and generated reports for 312 projects. The build of the remaining projects failed because it required human intervention, e.g., to set up a specific environment for the build. This was also observed in prior work [Hassan et al., 2017].

For diversity, we discovered Make projects from two sources. First, we used the Github API to collect popular C/C++ projects. Second, to ensure the buildability of the examined projects, we also employed the Ultimate Debian Database (UUD) [Debian, 2020b] to identify widely-used Debian packages based on the *"vote"* metric, which indicates the number of people who regularly use a specific package [Avery Pennarun and Reinholdtsen, 2020]. The build workflow of Debian packages utilizes the `sbuild` utility [Debian, 2020a], which automates the build process of Debian binary packages by creating the necessary build environment (e.g., it installs all build dependencies in an isolated environment) for a particular architecture (e.g., x86-64). `sbuild` allows us to hook over the build phase of its process. In this manner we can monitor each build and perform our own analysis. We built every Debian package using our Make wrapper (Section 4.3) instead of the default Make command. In total, we examined 300 Make projects coming from the Github and Debian ecosystems.

Popularity of the selected benchmarks: Overall, the list of the selected Gradle, Make, and Puppet projects contains popular and well-established ones. For example, our list of benchmarks includes the SQLite database, the Spring framework, which involve complex build scripts. Similarly, we examined well-established Puppet modules developed by popular organizations, such as Puppet Inc., and Vox Pupuli.[4]. As an example, Table 5.11 summarizes the characteristics of the selected projects related to Gradle and Make.

Hardware & execution environment: We ran every Gradle and Make build in sequential mode as in the work of Licker and Rice [2019] to make fair comparisons against mkcheck (Section 5.3.6). However, our approach is able to support parallel executions by tracking the thread (and its descendants) where every task is running. Finally, to spawn a clean environment efficiently, we ran FSMOVE on Docker containers inside (1) a host machine with an Intel i7 3.6GHz processor with 8 cores and 16GB of RAM for testing build scripts, and (2) a machine

[3] https://developer.github.com/v3/

[4] Vox Pupuli is a big community that is currently managing and maintaining more than one hundrend Puppet modules. https://voxpupuli.org/

with an Intel i7 2.2GHz processor with 12 logical cores and 16GB of RAM for testing Puppet scripts.

5.3.2 RQ3.1: Bug-Finding Results

Puppet results: The FSMOVE framework detected 92 previously unknown bugs in 33 Puppet modules. Table 5.10 presents the analysis results for each module. To the best of our knowledge, this is the first study that led to the disclosure of such a large number of issues in Puppet repositories. Our tool marks 70 out of 92 bugs as missing ordering relationships (column MOR). Thus, ordering violations are the most prevalent issue in the inspected manifests. The remaining bugs are related to missing notifiers (column MN).

Build system results: Table 5.11 summarizes our bug-finding results for the inspected build scripts. Our method identified problematic builds in 73 out of 312 Gradle projects. There are 157 issues related to incremental builds from which 122 bugs are missing inputs appearing in 58 projects, while the remaining issues (35) are associated with missing outputs found in 21 projects. Faulty parallel builds are also common in Gradle projects, as we uncovered 80 ordering violations in 25 Gradle repositories. Furthermore, our tool detected issues in 251 Make projects; it discovered 15,740 Make target rules with missing inputs. Most of them involved missing header dependencies concerning object files. It also reported 14 ordering violations that may lead to race conditions in 5 projects.

The large number of missing inputs (15,740) found in Make projects is justified by the complex structure of the examined builds. Most of the projects use configuration scripts (e.g., GNU autotools, etc.) that auto-generate complicated build rules with many recursive Make definitions. We also observed a common programming pattern that may have caused many missing inputs: developers often declare some header files in a Make variable, and use this variable as prerequisite of different build rules. When developers miss out to specify a header file in this variable, then all the rules that use this header file suffer from a missing input issue.

Regarding missing outputs, we did not detect this kind of bug in Make projects, as it is only relevant to Gradle (Section 2.4.1.2). Specifically, we modeled every Make rule as a FSMOVE task that was producing $\top$ (any file). Based on the [TOP] rule of Figure 3.18, the subsumption relation always holds when verifying a task whose declared output is $\top$. Therefore, we never reported missing outputs in tasks producing $\top$ because there was no violation of the subsumption property (Definition 3.3.4).

Verifying discovered bugs: To verify that the issues detected by our tool are indeed bugs, we worked as follows. For Puppet modules, we repeatedly applied every manifest in a clean container until Puppet applied resources in the wrong order, leading to a failure or an inconsistent state. Only few trials were needed (1–3) for that. For Gradle projects, we examined each bug report, and tried to reproduce it. Specifically, we automatically verified each issue related to incremental builds by checking that re-running Gradle does not trigger the execution of tasks marked with missing inputs/outputs by our tool, even after updating the contents of their dependent files. We followed the same automated approach for the verification of the reported Make bugs associated with incremental builds. For ordering violations, we manually

Table 5.10: Bugs found in Puppet modules. Each table entry consists of the name of the module, the number of bugs detected by FSMOVE and a check mark indicating whether our fixes were accepted by the module's developers.

		Number of Bugs			Fix
#	**Module**	Total	MOR	MN	**Accepted**
1	puppet-proxysql	10	10	0	✔
2	istlab-stereo	9	9	0	✔
3	olivierHa-influxdb	7	6	1	-
4	hetzner-filebeats	6	5	1	✔
5	geoffwilliams-auditd	5	5	0	✔
6	coreyh-metricbeat	4	4	0	✔
7	coreyh-packetbeat	4	4	0	✔
8	norisnetwork-packetbeat	4	4	0	✔
9	Slashbunny-phpfpm	4	2	2	✔
10	nogueirawash-mysqlserver	3	3	0	-
11	cirrax-dovecot	3	1	2	✔
12	nextrevision-flowtools	3	1	2	✔
13	deric-zookeeper	3	0	3	✔
14	albatrossflavour-os_patching	2	2	0	✔
15	hardening-os_hardening	2	2	0	✔
16	vpgrp-influxdbrelay	2	2	0	✔
17	puppet-collectd	2	1	1	✔
18	sgnl05-sssd	2	1	1	✔
19	jgazeley-freeradius	2	0	2	✔
20	saz-ntp	2	0	2	-
21	walkamongus-codedeploy	1	1	0	✔
22	spynappels-support_sysstat	1	1	0	-
23	roshan-mysqlzrm	1	1	0	-
24	puppetfinland-nano	1	1	0	✔
25	noerdisch-codeception	1	1	0	-
26	baldurmen-plymouth	1	1	0	✔
27	saz-locales	1	1	0	-
28	alertlogic-al_agents	1	1	0	-
29	puppet-telegraf	1	0	1	✔
30	puppetlabs-apache	1	0	1	✔
31	example42-apache	1	0	1	✔
32	alexharvey-disable_transparent_hugepage	1	0	1	✔
33	camptocamp-ssh	1	0	1	✔
	Total	**92**	**70**	**22**	**63/92**

Table 5.11: Bugs detected by our approach. Each table entry indicates the number of buggy projects/the total number of the examined projects (Projects), the average LoC (Avg. LoC), and the average lines of build scripts (Avg. BLoC). The columns MIN, MOUT, and OV indicate the number of projects where missing inputs, missing outputs, and ordering violations appear respectively.

	Project Characteristics			Bug types		
Build System	**Projects**	**Avg. LoC**	**Avg. BLoC**	**MIN**	**MOUT**	**OV**
Gradle	73/312	35,536	589	58	21	25
Make	251/300	74,414	838	249	-	5

verified that executing conflicting build tasks in the erroneous order can affect the outcome of a build, e.g., causing build failures, or producing build targets with incorrect contents.

For Make projects, we also analyzed every build with mkcheck, which is the state-of-the-art tool for Make-based builds [Licker and Rice, 2019]. We then cross-checked the results generated by our tool with those produced by mkcheck (see Section 5.3.6).

5.3.3 RQ3.2: Importance of Discovered Bugs

Based on the bug reports generated by FSMOVE, we prepared and submitted fixes to the developers to examine the importance of the discovered issues. Regarding Puppet bugs, we submitted patches to the development teams of all the 33 Puppet modules where FSMOVE identified an issue. For build scripts, we provided fixes for 71 Make and Gradle projects that we chose while we were examining their bug detection results, and in turn, we submitted patches to the upstream developers. Patch generation was done manually, and substantial additional work was required to propose a suitable fix that would satisfy the development standards of each project. This was mostly because of the complex structure of the buggy projects, and the peculiar semantics of each system's DSL. We leave repairing build scripts and Puppet scripts through automated means as future work.

Puppet results: As shown in Table 5.10, the development teams of 25 projects confirmed and fixed 63/92 Puppet issues in total. The developers welcomed our patches. Notably, in some projects (e.g., `deric-zookeeper`, `Slashbunny-phpfpm`, `cirrax-dovecot`, and more), the developers provided instant bug-fix releases after the approval of our patches. Remarkably, FSMOVE found bugs in modules that are widely used by the Puppet community e.g., `puppetlabs-apache` (> 9000k downloads), and `deric-zookeeper` (> 4500k downloads).

Build system results: Table 5.12 enumerates the bugs that are confirmed and fixed. Notably, 237 issues found in 47 out of 71 projects were fixed, while most of the remaining patches are in a pending state. In a small number of projects, the corresponding developers rejected our patches, but they confirmed and finally fixed the reported bugs on their own.

The list of projects where our patches were accepted contains popular projects, such as `tinyrenderer` (∼8k stars), `caffeine` (> 7k stars), `aeron` (∼5k stars), `Cello` (∼5k stars), and more. The list also includes projects that are maintained and developed by well-established organizations, such as `conductor` (developed by Netflix), `tsar` (developed by the Alibaba Group).

All the results above clearly indicate that the bugs we identified do matter to the community.

Table 5.12: Issues confirmed and fixed by the developers of the examined build scripts.

Project	Build System	Total	MIN	MOUT	OV
junit-reporter	Gradle	10	2	3	5
aeron	Gradle	7	2	2	3
muwire	Gradle	6	0	0	6
xtext-gradle-plugin	Gradle	4	0	0	4
rundeck	Gradle	3	1	1	1
apina	Gradle	3	0	0	3
caffeine	Gradle	2	0	0	2
conductor	Gradle	2	0	1	1
RxAndroidBle	Gradle	2	0	0	2
jmonkeyengine	Gradle	1	0	1	0
tsar	Make	31	31	-	0
Cello	Make	27	27	-	0
webdis	Make	27	27	-	0
density	Make	17	17	-	0
VRP-Tabu	Make	12	12	-	0
pspg	Make	9	9	-	0
janet	Make	8	8	-	0
parcellite	Make	8	8	-	0
reptyr	Make	5	5	-	0
hdparam	Make	4	4	-	0
Others[1]	Make,Gradle	49	40	0	9
	Total	**237**	**193**	**8**	**36**

[1] **Others:** gps-sdr-sim, cscout, proxychains, gradle-scripts, p2rank, fzy, groocss, ShellExec, anna, fulibGradle, nf-tower, alfresco-gradle-sdk, GradleRIO, helios, kscript, coveralls-gradle-plugin, gradle-test-logger-plugin, joystick, cqmetrics, JenkinsPipelineUnit, swagger-gradle-plugin, gradle-swagger-generator-plugin, gradle-sora, tinyrenderer, libcs50, LuaJIT, greatest

5.3.4 RQ3.3: Characteristics of Discovered Bugs

Below, we categorize and discuss some of the bugs identified by FSMove. Most represent previously unknown to us bug patterns, which we learned through our tool. Notably, these detected bugs involve a variety of implications, including crashes, inconsistent states, data loss. We provide a separate discussion for Puppet- and build-related bugs.

5.3.4.1 Characteristics of Puppet Bugs

We have observed three types of ordering violations that are commonly found in Puppet scripts, namely, *generate-use violation*, *configure-use violation*, and *API misuse*.

Generate-use violation: The use of a resource must always succeed its creation. Many modules fail to preserve that ordering relationship. Consequently, the execution of Puppet may complete with failures, when resources are applied in an erroneous order. We observed this violation in 16 Puppet modules such as `alertlogic-al_agents`, `hardening-os_hardening`, and more.

Figure 5.5 shows a fragment from `alertlogic-al_agents` [Alert Logic Inc., 2019]. The code first fetches a `.deb` package (a Debian archive) using the `wget` command (lines 2–4). The package is stored in the path specified by the `$package_path` variable whose value is

```
1 $package_path = "/tmp/al-agent"
2 exec {"download":
3   command => "/usr/bin/wget -O ${package_path} ${pkg_url}",
4 }
5 package {"al-agent":
6   ensure => "installed",
7   provider => "dpkg",
8   source => $package_path,
9 }
```

Figure 5.5: An ordering violation between package and exec.

/tmp/al-agent. Then, the code installs the Debian archive on the system (lines 5–9) through the dpkg package management system. It is easy to see that the package depends on the exec, because it requires $package_path (the .deb file) to exist in the system (line 8) so that it can install the package successfully. Otherwise, when Puppet processes package before exec the application of the catalog fails with the following error: *"dpkg: error: cannot access archive "/tmp/al-agent": No such file or directory"*.

Configure-use violation: The configuration of a file must precede its use. For example, when a service starts, all the files consumed by that service have to be properly configured. This category differs from the previous one because when a Puppet resource attempts to use the file, the latter exists in the system. However, this is not in the expected state (e.g., the file does not have the right contents, permissions, etc). This error pattern appears in five modules, including saz-ntp, vpgrp-influxdbrelay, and jgazeley-freeradius.

Figure 2.27 illustrates a program with an issue related to this category. When the shell script is invoked, the configuration file is guaranteed to be there because package creates it during installation. However, it is possible that exec does not read the desired contents of the /etc/mysql/my.cnf file specified by content => "user db settings.." (line 4), because there is a missing ordering relationship between file and exec. Note that this category—unlike the previous one—may lead to errors that are difficult to debug as the application of the Puppet catalog does not produce any error messages.

API misuse: Many Puppet modules expose an API that other modules rely on to build their functionality. These APIs may establish some constraints that the dependent modules need to respect to achieve the intended functionality. As with traditional software [Amann et al., 2018; Kechagia et al., 2019], failing to do so can have a negative impact on the reliability of applications. In particular, in Puppet, API misuses can lead to missing dependencies and race conditions. Eight modules (such as puppet-proxysql) do not properly use the API of their dependencies, causing the ordering violations reported by FSMove.

For example, the puppetlabs-apt module provides an interface for managing apt[5] sources and keys. The API of this module includes—among other things—the apt::source resource (Figure 5.6) and the apt::update class. The former is used for adding new repositories to the list of apt sources, while the latter retrieves all the essential information about the newly-added repositories by executing the apt update command. The puppet-proxysql module employs the apt::source resource to add the http://repo.proxysql.com/ repository from which it

[5] https://salsa.debian.org/apt-team/apt

```
1  # module puppetlabs-apt
2  define apt::source(String $loc) {
3    include ::apt::update
4    file { "/etc/apt/sources.list.d/${title}.list":
5      ensure => "file",
6      content => "$loc$",
7      notify => Class["::apt::update"]
8    }
9  }
10 # module puppet-proxysql
11 apt::source {"proxysql": loc => "http://repo.proxysql.com/" }
12 package {"proxysql":
13   ensure => "latest",
14   require => Apt::Source["proxysql-source"]
15 }
```

Figure 5.6: Misuse of the `puppetlabs-apt`'s API.

installs `proxysql` via the `package` resource (Figure 5.6, lines 11–15). The documentation of the `puppetlabs-apt`'s API [Puppet, 2019a] states: *"If you are adding a new source and trying to install packages from the new source on the same Puppet run, your package resource should depend on Class['apt::update'], as well as depending on the Apt::Source resource"*. However, the developers of `puppet-proxysql` consider only the `Apt::Source` dependency in their code, i.e., they omit the `Class[apt::update]` dependency. As a result, the code may crash with an *"Unable to locate package proxysql"* message, when Puppet tries to install `proxysql`, before invoking the `apt update` command first. The developers of `puppet-proxysql` immediately confirmed and fixed this fault.

We have identified three different categories of issues related to notifiers, i.e., *configuration files*, *log files*, and *packages*.

Configuration files: A configuration file must always send notifications to a service, so that any change to that file triggers the restart of the corresponding service. Although this is a standard pattern, we observed that in four modules (shown in rows 12, 19, 20, 32 of Table 5.10) this is not the case.

Log files: Typically, services log various events in dedicated files. For instance, the log file of an Apache server records every incoming HTTP request. Log files are essential for debugging and monitoring purposes [Spinellis, 2016, Item 56]. When a service starts, it opens a corresponding log file, which remains open, while the service is up, to write any events that take place.

We discovered issues related to logging in two popular Puppet modules: `puppetlabs-apache` and `deric-zookeeper`. These modules declare the log files for the `apache` and `zookeeper` services in their manifests. However, the log files do not have a notifier for their associated services. This may lead to data loss. Consider the case where the log file of a service is removed or renamed. When we remove or rename an open file, the underlying system call (`unlink` or `rename`) only changes the file entry, not the `inode`. This means that although the filename disappears from the file system and Puppet creates a new one, the service still uses a file descriptor that points to the `inode` of the original file. The issue is that

after removal, the `inode` typically becomes an *orphan* (i.e., it is not linked with any file), which means that it is no longer accessible through a file path. Therefore, in the case of a missing notifier, the log history of the upcoming events is lost because the service writes to an orphan `inode`. To fix that issue, the log file should notify the service so that the service opens the newly-created log file. The developers of both projects confirmed this kind of bug.

Packages: When Puppet applies a `package` resource, the service that depends on that package should restart. This ensures that a service gets all the necessary updates, including, security patches, new features, and more. FSMove identified this kind of issue in twelve modules, including example42-apache, and puppet-telegraf. Specifically, the `package` resources that are responsible for installing Apache, and telegraf do not notify the running instances whenever there is a new version of those packages.

5.3.4.2 Characteristics of Bugs in Build Scripts

When we manually examined the build-related issues generated by FSMove, we recognized five bug patterns, which result in build failures, time-consuming builds, or erroneous build outcomes. We identified three kinds of bugs related to incremental builds caused by missing inputs or outputs, namely, *test resources*, *stale artifacts*, and *time-consuming tasks*.

Test resources: To ensure the correctness of their programs, developers typically specify dedicated build rules for performing different forms of testing (e.g., unit and functional testing) during build. Running tests is a time-consuming task [Gligoric et al., 2015], so the build rules associated with tests are triggered only when there are updates to any of the source files that tests rely on. As with source files, changing any of the resources used by tests (e.g., test data or additional helper scripts) must re-run tests to make sure that the change does not break anything. Not running tests is a missed opportunity to identify potential issues and may lead to late identification of bugs.

For example, the Gradle project kscript contains a test suite of Kotlin files included in the `test/resources` directory. The tests of `kscript` contains test assertions that rely on the state and contents of the files included in this test suite. However, the developers failed to declare the test suite directory as input of the Gradle task `test`. Our tool detected this bug, and we reported to the developers who fixed it. We can find this type of issue in other projects as well, such as fulibGradle, alfresco-gradle-sdk, groocss.

Stale artifacts: As already discussed, the main goal of a build is to construct artifacts, such as executables, libraries, documentation accompanying software, and more. The build process must re-generate these artifacts, when any of the files used for their construction is updated since the last build. Failing to do so can lead to stale artifacts, which in turn, can either harm the reliability of applications, (e.g., cause runtime errors), or generate wrong build outputs.

This pattern is particularly common in Make builds where developers do not enumerate the dependencies of object files correctly. As an example of stale artifacts, recall the build script of Figure 2.28. This example demonstrates that even best practices for tracking dependencies automatically (e.g., through `gcc -MD`) are not sufficient for ensuring the correctness of builds. This type of issue also appears in Gradle projects, such as jmonkeyengine, goomph.

Time-consuming tasks: The purpose of incremental builds is to reduce build time by

```
1 LIBS := $(addprefix build/lib/, $(LIB_BASE) $(LIB_VERSION))
2 $(LIBS): $(SRC)
3   $(CC) $(CFLAGS) -o $(LIB_VERSION) $(SRC)
4   $(CC) $(CFLAGS) -c -o $(LIB_OBJ) $(SRC)
5   rm -f $(LIB_OBJ)
6   ln -sf $(LIB_VERSION) $(LIB_BASE)
7   mv $(LIB_VERSION) $(LIB_BASE) build/lib
```

Figure 5.7: A Make script with conflicting producers.

running only the build tasks needed to achieve a specific goal. This boosts productivity as it enables developers to get feedback and respond to changes of their codebase much earlier. To avoid unnecessary computation, it is important that time consuming build tasks are incremental.

We identified an instance of this pattern in the Gradle project `junit-reporter`. This project contains some JavaScript source files used to visualize JUnit reports in HTML format. Developers define a Gradle task to bundle JavaScript source files into a single file (`site.js`) that is finally incorporated in the HTML page of test reports. However, this task was not declared as incremental. As a result, Gradle was bundling JavaScript files at every build, causing the subsequent Gradle tasks that were dependent on `site.js` to be executed, as Gradle was creating a newer version of `site.js` each time. Our tool marked `site.js` as a missing output of the bundler task. Based on this, we sent a patch to the upstream developers who integrated it in their codebase. Fixing this issue made builds eight times faster.

Below we discuss two categories of bugs related to parallel builds.

Conflicting producers: We have identified issues associated with tasks that produce the same file or write to the same output directory. Parallel execution of such build tasks is harmful, because race conditions may emerge, when two tasks affecting the same state (i.e., files) run concurrently.

Figure 2.26 demonstrates an example of conflicting producers. Recall that the Gradle tasks `shadowJar` and `jar` produce a JAR file with the same name, but with different content. In addition, there is no dependency between these tasks. Hence, the content of the generated JAR is not deterministic and depends on the order in which Gradle schedules tasks' execution.

Figure 5.7 shows another example coming from the libcs50 project. This Make script defines a rule (line 2) that creates two libraries inside the `build/lib` directory (see variable `$(LIBS)`). The code first compiles the source file into the corresponding object file (line 3) from which a shared library, namely `$(LIB_BASE)`, is constructed (line 4). Then, it creates a symbolic link (`$(LIB_VERSION)`) pointing to the newly-created library (line 6), and finally moves these files to the `/build/lib` directory (line 7). The official documentation of GNU Make states that such a rule definition is incorrect [GNU Make, 2020b], and can result in race conditions. In particular, the rule at line 2 is executed twice (one for every target defined in the `$(LIBS)` variable). Consequently, the parallel build might crash with the error *"mv: cannot stat 'libcs50.so.10.1.0': No such file or directory"*, as every rule execution races against each other. Specifically, when the second rule invocation attempts to move the libraries, they may have already been moved to `/build/lib` by the first rule. The developers of `libcs50` immediately fixed this problem. We

```
1  sourceSets {
2    main {
3      srcDir "build/generated-sources/"
4    }
5  }
6  task generateNodes(type: JavaExec) {
7    main = "com.github.benmanes.caffeine.cache.NodeFactoryGenerator"
8    args "build/generated-sources/"
9    outputs.dir "build/generated-sources/"
10 }
11 apply plugin: "com.bmuschko.nexus"
```

Figure 5.8: A Gradle script manifesting an ordering violation.

can also find this pattern in muwire, RxAndroidBle, nf-tower, and others.

Generated source files and resources: Many projects generate part of their source code or resources at build time. These automatically generated source files and resources are then compiled or used later by other build tasks to form the final artifacts of the build process, e.g., binaries. Developers must be careful enough to preserve the correct execution order between the build tasks that are responsible for generating and using these source files and resources. Ordering violations (e.g., compiling code when source files are missing) are the root cause for build failures, or subtle errors detected at a later stage of software lifecycle.

Figure 5.8 presents a code fragment taken from the popular caffeine project. The code specifies that the source files of the project are stored in the `build/generated-sources` directory (line 3). These source files are generated automatically by the Gradle task `generateNodes`. To do so, this task runs the class `NodeFactoryGenerator` with `"build/generated-sources"` as an argument (lines 7–9). Then, this code applies the plugin `"com.bmuschko.nexus"` used for uploading the sources JAR file to a remote repository. To assemble a JAR file containing the source files of the application, this plugin adds the `sourcesJar` task to the project. The problem with this code is that no dependency is declared between the tasks `generateNodes` and `sourcesJar`. Thus, the build process uploads empty artifacts to the remote repository, when Gradle executes `sourcesJar` before `generateNodes`. The developers of `caffeine` confirmed and fixed this ordering issue. We also encounter this bug pattern in jmonkeyengine, aeron, conductor.

5.3.5 RQ3.4: Performance

To measure the performance of FSMOVE we recorded the time spent at each step (recall Figure 3.13). The generation step, which is responsible for executing and monitoring our instrumented scripts, dominates the execution time of our method. In particular, this step slows down both builds and Puppet applications by a factor of around two for the 90th percentile of the examined projects. This is consistent with the recent literature [Licker and Rice, 2019], as the main overhead of this phase stems from the system call tracing utility (i.e., `strace`).

The analysis of FSMOVE programs takes around 2, 2.47 and 5.1 seconds on average for Puppet, Make, and Gradle scripts respectively, and is linear to the size of programs, as shown in

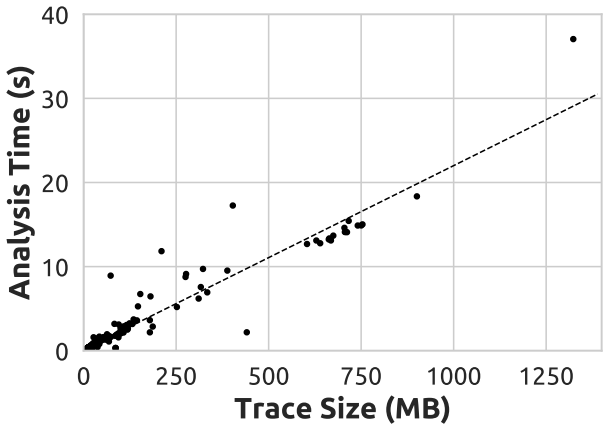


Figure 5.9: The FSMOVE analysis time as a function of the trace size. Each spot shows the average time spent on FSMOVE analysis for a given trace obtained by the execution of a Puppet module.

Table 5.13: Time spent on analyzing builds and detecting bugs by our approach vs. mkcheck (in seconds).

	Median (seconds)		Average (seconds)	
Phase	**FSMoVe**	**mkcheck**	**FSMoVe**	**mkcheck**
Build	4.68	4.91	21.64	22.2
Bug detection	0.01	182.62	0.44	3,368
Overall	4.69	186.75	22.08	3,390

Figure 5.9, which presents the performance results for the examined Puppet benchmarks. This analysis phase is efficient enough to analyze GBs of programs in a reasonable time (e.g., 6.9GB in less than 3 minutes). In online mode, though, the observable time spent on the analysis step is eliminated, as the overall time is bounded to the time needed for a build or a Puppet program execution. As explained in 4.3, this is because the processing of FSMOVE programs is faster than the build itself, and thus we take advantage of multicore architectures. Finally, the bug detection step employed by FSMOVE is pretty fast; it takes less than a second, on average.

5.3.6 RQ3.5: Comparison with State-Of-The-Art

Unfortunately, a side-comparison between FSMOVE and the state-of-the-art tools for Puppet was not feasible or fair. Citac [Hanappi et al., 2016] works on an instrumented version of Puppet 3.7. However, Puppet 3.x reached its end of life, and the vast majority of modules no longer support Puppet 3.x. Furthermore, as already discussed, Rehearsal [Shambaugh et al., 2016] is unable to analyze modules that use the `exec` resource. More than a half of the inspected modules contain `exec`, let alone their dependencies. Also, Rehearsal cannot statically analyze modules whose source code consists of multiple files. Finally, in most cases, Rehearsal's analysis aborts prematurely due to some internal bugs, e.g., its custom parser fails to parse complex Puppet constructs. For these reasons, Rehearsal cannot analyze any of the modules listed in Table 5.10.

The remaining section presents the details of comparing FSMOVE with mkcheck [Licker and

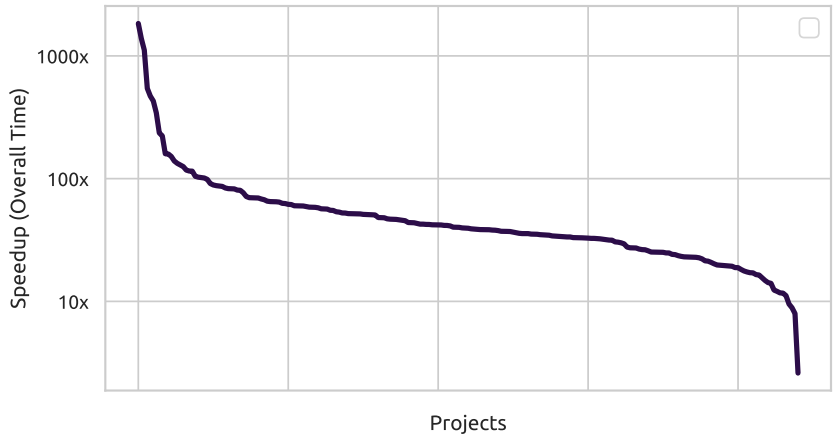


Figure 5.10: The speedup for every Make project by our approach over mkcheck (ordered by y-axis). This plot considers the overall times (build time + bug detection time) spent by our approach and mkcheck.

Rice, 2019], a state-of-the-art tool for detecting issues in parallel and incremental builds. As a first step, we built and analyzed a number of Gradle projects with mkcheck. After finding that mkcheck produces an overwhelming number of false positives (this is not surprising because mkcheck is unable to deal with JVM-based builds as explained in Section 2.4.3), we focused on performing comparisons only for Make projects.

We applied mkcheck to the 300 Make projects of our experimental setup and we recorded the bug reports and the time spent at each phase (i.e., build time and bug detection time). Note that build time includes the time needed for building the project as well as the time taken for generating and analyzing the build trace.

In terms of bug detection results, mkcheck produced false positives in three cases due to the granularity of processes. That is, two build tasks were merged into a single task because they were governed by the same system process, leading to imprecision. False positives are also observed in the initial work of Licker and Rice [2019]. On the contrary, our approach did not generate false positives as it can reliably determine all file accesses of each task as explained in Section 3.3.4.

Regarding efficiency, we compared our approach with mkcheck when (1) building a project, and (2) detecting bugs. Starting with build times, our tool and mkcheck spend almost the same amount of time for building and monitoring a project. The minimum speedup is -1.25×, the median is 1.04×, the average is 1.19×, while the maximum speedup is 9×. Although mkcheck uses `ptrace` for tracking system operations, which is 20% times faster than `strace` Licker and Rice [2019], our approach benefits from running the generation and analysis steps concurrently. Moving to bug detection times, we observe that our approach outperfoms mkcheck in terms of efficiency: the minimum speedup of our bug detection algorithm is 83×, the median is 25,431×, the average is 56,863×, and the maximum speedup is 2,708,917×. This huge speedup is explained by the fact that mkcheck needs to trigger multiple incremental builds (one for every source file) for verifying correctness. Therefore, incremental builds dominate the time needed for detecting bugs in mkcheck. Notably, in projects consisting of a large number of source files,

mkcheck required days to detect bugs (e.g., it spent 3.3 days for analyzing ghostscript). Contrariwise, our bugs detection algorithm performs no incremental builds; it just iterates over the file accesses computed by the analysis of the FSMOVE program, and checks whether the expected subsumption and happens-before relations hold (Section 3.3.5). Finally, when considering the overall time (i.e., build + bug detection time), our approach is 74× faster than mkcheck, on average. The median speedup is 39×, while the minimum and maximum speedup is 2.6× and 1,837× respectively (see Figure 5.10).

Furthermore, we provide some absolute performance times on Table 5.13. Overall, our method required 22 seconds (on average) for analyzing builds and detecting bugs (see phase "Overall" on Table 5.13), while mkcheck spent 3,390 seconds on average for performing the same tasks. The median time is 4.69 and 186.75 seconds for our tool and mkcheck respectively. Notably, mkcheck spent more than ten minutes for reporting bugs in the 29% of the inspected Make projects.

Our findings, indicate that our method is superior to the state-of-the-art in terms of both bug detection and performance. Moreover, due to its effectiveness and efficiency, we argue that our method can be used in practice as part of the software testing pipeline.

Chapter 6

Related Work

In this chapter, we cover work related to the methods presented in this thesis, and we explain how our techniques and tools differ from prior work.

6.1 Related Work on Compiler Reliability

In this section, we present the existing work on enhancing the reliability of compilers. As an effort to better understand them, in Section 2.2, we conducted a bug study of previously-reported compiler typing bugs. We now report previous empirical studies on compiler defects (Section 6.1.1), and then conclude with a summary of all the exciting work on compiler testing that relates to HEPHAESTUS (Section 6.1.2).

6.1.1 Understanding Compiler Defects

Understanding compiler bugs: The closest bug study to our work is that conducted by Sun et al. [2016c]. The authors collected and automatically analyzed 52,732 bugs and 31,399 revisions from the GCC and LLVM compilers. Their study focused on the following aspects: (1) location of bugs, (2) size of test cases and bug fixes, (3) lifetime of bugs, and (4) priorities of bugs. Some of their key findings are: (1) C++ is the most buggy component of the examined compilers, (2) GCC and LLVM bugs are typically triggered by small test cases (3) most of the bug fixes are local and (4) developers need a couple of months to resolve the reported bugs. Zhou et al. [2021] recently repeated the study of Sun et al. [2016c], but this time, the researchers gave emphasis to optimization bugs in GCC and LLVM. Some of their results (e.g., size of test cases, duration of bugs, locality of bugs) are consistent with the findings of Sun et al. [2016c]. In a different spirit, Marcozzi et al. [2019] tried to measure the effect of compiler bugs found by fuzzing tools on real-world application code. According to their results, most of the fuzzer-found bugs indeed affect the final executables produced by compilers, but they semantically change only a small portion of the code (typically involving a small number of functions).

Our bug study is complementary to these previous studies. It provides the first insights into understanding the nature of typing-related bugs, a category of bugs that is currently overlooked.

The findings of our bug study can be combined with the findings of the other studies in order to design techniques for a more rigorous testing of compilers.

Other bug studies: Here we briefly present recent studies that are closely related to the study of Section 2.2. Trying to investigate the characteristics of distributed concurrency bugs, Leesatapornwongsa et al. [2016] manually analyzed 104 non-deterministic concurrency bugs from four distributed systems used in a production environment. Their analysis consisted of several aspects, including bug symptoms and fixing. Their findings contribute to the better understanding of distributed bugs and facilitate the design of future verification and testing tools. Other studies on concurrency bugs in server-side JavaScript [Wang et al., 2017; Davis et al., 2017] showed that such bugs are mainly caused by atomicity and ordering violations, and beyond shared memory, a significant number of bugs is triggered by races in external resources, such as files or databases. Bagherzadeh et al. [2020] focused on actor-based concurrency bugs. They constructed a dataset consisting of 186 concurrency bugs found in Akka coming from Stack Overflow questions, and GitHub projects. For each bug in the dataset, they identified its symptom, root cause, and the Akka APIs that the buggy program uses. Their results showed that crashes is the most prevalent category of symptoms, while Akka concurrency bugs are mainly caused by logic faults.

Numerical bugs form another category of bugs that has been examined by previous empirical studies. Di Franco et al. [2017] selected 269 numerical bugs from five popular numerical libraries (i.e., NumPy, SciPy, LAPACK, GNU Scientific Library, and Elemental), and classified them into four categories based on their patterns and root causes. One of this study's takeaways is that some of the numerical bugs can be detected and fixed by adopting rule-based approaches, as many of them follow specific patterns, e.g., some accuracy bugs can be fixed by simply re-ordering arithmetic expressions. In a subsequent study, Dutta et al. [2018] characterized inference-related bugs by manually analyzing 118 commits from three probabilistic programming systems. Their categorization involves accuracy bugs, bugs associated with the handling of special numerical values (e.g., `NaN`), and other correctness issues. Based on their findings, they also proposed a differential testing approach for finding such bugs.

Jin et al. [2012] performed one of the first bug studies for performance bugs. They selected 109 real-word performance bugs from well-established systems (e.g., GCC, MySQL), and showed how these bugs are introduced and fixed. They designed a bug-finding tool that was able to detect 332 performance issues in MySQL, Apache and Mozilla.

6.1.2 Compiler Testing

Testing compilers through randomized testing is a research topic that has received much attention recently (see the survey of Chen et al. [2020] for more details).

Program Generators for Compiler Testing: Csmith [Yang et al., 2011] is a program generator for C programs, that has found hundreds of bugs in GCC and Clang. Csmith generates programs that are free from *undefined behavior*. To do so, it employs various strategies and static analyses to prevent the generation of programs with errors such as out-of-bounds array accesses, use of uninitialized variables, and others. Relying on Csmith, several other program

generators have emerged for (1) testing other compilers (e.g., OpenCL) [Lidbury et al., 2015a], or link-time optimizers [Le et al., 2015b], and (2) generating more expressive programs [Even-Mendoza et al., 2020].

Epiphron [Sun et al., 2016a] is a program generator for C that aims to uncover defects in the error reporting mechanisms of compilers. Unlike Csmith, Epiphron does not necessarily generate programs that are free from undefined behavior. Targeting optimizations bugs, Orange [Nagai et al., 2014] creates programs that involve longer and more complex arithmetic expressions, such as floating-point arithmetics. Orange uses a set of heuristics to eliminate rounding errors in floating-point arithmetics, or arithmetic overflows. YARPGen Livinskii et al. [2020] is a program generator for C/C++ programs that comes with a set of generation policies aiming to trigger certain optimizations (e.g., global value numbering, constant folding). Similarly to YARPGen, in Section 3.1.4 we have designed approaches (i.e., TEM and TOM) for testing specific components (e.g., inference engines, soundness checks) in the typing procedures of compilers.

Most of the aforementioned program generators focus on the detection of crashes or miscompilations caused by optimization bugs. Finding miscompilations requires *differential testing* [McKeeman, 1998]. Contrary to this work, Hephaestus does not involve differential testing and focuses on compiler typing bugs where test cases act as their own oracle.

Dewey et al. [2014, 2015] have introduced a *Constraint Logic Programming (CLP)* approach for synthesizing programs for JavaScript and Rust. The idea of the CLP-based program generation is to encode all syntactic and semantic rules (e.g., type system) of the language to logic predicates, and then use a constraint solver to generate test programs. Like Hephaestus, their fuzzing approach finds precision and soundness bugs. However, as stated by the authors, one of the fundamental shortcomings of CLP-based program generation and encoding typing rules into logic predicates, is poor performance. Moreover, the techniques behind Hephaestus are (1) adaptable (already applied to three languages), (2) more effective (it identified more bugs than the fuzzer of Dewey et al. [2015]), and (3) the first to validate type inference algorithms.

Transformation-Based Compiler Testing: *Equivalence Modulo Input (EMI)* [Le et al., 2014, 2015a; Sun et al., 2016b] is an effective metamorphic testing [Chen et al., 1998] approach for finding bugs in optimizing compilers. EMI transforms a given program in a way that does not change its output under the same input. This is achieved by deleting dead statements [Le et al., 2014], inserting code in dead regions [Le et al., 2015a], or even updating live parts [Sun et al., 2016b]. EMI testing has been also ported to testing OpenCL compilers [Lidbury et al., 2015a], and simulation software [Chowdhury et al., 2020].

GLFuzz and spirv-fuzz [Donaldson et al., 2017, 2021] repeatedly apply a set of semantics-preserving transformations (e.g., dead code injection) to an initial corpus of programs for finding bugs in graphics shader compilers. classfuzz [Chen et al., 2016] and classming [Chen et al., 2019] employ a set of transformations on existing Java bytecode programs to test JVM implementations through differential testing. They both leverage *Markov Chain Monte Carlo sampling* [Chib and Greenberg, 1995] to guide either mutator or mutant selection. However, unlike classming, classfuzz may lead to semantically invalid programs. Other transformation-based approaches include the AFL-based (American Fuzzy Lop [M. Zalewski, 2013]) method intro-

duced in the work of Wang et al. [2019], namely Superion. Although Superion is effective in testing lexers and parsers, it struggles to construct semantically-valid inputs that get past the very front-end of a compiler and exercise other compiler components, such as code generation or typing procedures. Given a specific program structure, *Skeletal Program Enumeration (SPE)* [Zhang et al., 2017] enumerates all variant programs that expose different variable usage patterns.

SPE is complementary to the mutations of Hephaestus. For example, instead of removing the maximal set of types, our type erasure mutation could employ SPE to enumerate all variant programs that manifest different patterns of omitted type information. Similarly, we could combine SPE with fault-injecting mutations (e.g., TOM), to identify what program points are promising to inject the error.

Inspired by SPE, Stepanov et al. [2021] have designed *Type-Centric Enumeration (TCE)*. TCE produces variants by assigning different values to variables or call arguments, while preserving the same type information as the original program. Unlike Hephaestus, TCE is effective in primarily finding crashes caused by back-end bugs. Another similar approach to TCE is *generative type-aware mutation* [Park et al., 2021], which has been recently used for testing SMT solvers. Like TCE, generative type-aware mutation replaces an expression of an SMT formula with a newly-generated expression of the same type. A variant of this is *type-aware operator mutation* [Winterer et al., 2020a], which substitutes an SMT operator with another compatible operator. Instead of replacing expressions and operators, our type overwriting mutation replaces types. Also, the existing approaches (e.g., TCE) respect the semantics of the input program, while TOM is the first to adopt a fault-injecting approach, as an effort to find soundness bugs.

6.2 Related Work on the Reliability of Data-Centric Software

We now discuss two research areas related to the differential testing approach presented in Section 3.2: (1) Quality in ORM-based application code, and (2) testing of DBMSs. Even though these systems are extensively used, to the best of our knowledge, our thesis includes the first testing effort targeting bugs in ORM systems.

Quality in ORM-based applications.: A number of tools and studies have been proposed to improve the quality of ORM-based applications. Chen et al. [2014] introduced a static analysis framework for identifying ORM queries in Java applications that degrade the response times of database engines. Their approach first explores the paths of the program to identify database accesses, and then detects performance anti-patterns through a rule-based approach. Furthermore, their technique provides an assessment mechanism for prioritizing the fixes of the detected performance issues. Subsequent work [Davar and Handoko, 2014; Singh et al., 2016] focused on fixing performance issues through automated means. In particular, Singh et al. [2016] introduced a genetic algorithm for tuning the configuration of ORM systems to achieve better performance. Davar and Handoko [2014] proposed a refactoring framework by applying

a set of known transformation rules to inefficient ORM-based code. A later study [Chen et al., 2016] pointed out that there are several maintenance issues in ORM-based applications. Unlike this prior work that finds issues in the ORM-based applications, CYNTHIA is the first to find issues in the ORM implementations.

Testing of database engines: The work of Slutz [1998] is the first to uncover bugs in DBMSs using a differential testing approach. To safely compare results, his method generates random queries on a small subset of the SQL language that is common across DBMSs. Over the past decade, there have been numerous approaches for generating (targeted) SQL queries in order to effectively test DBMSs [Bruno et al., 2006; Bati et al., 2007; Mishra et al., 2008; Abdul Khalek and Khurshid, 2010]. The most recent approaches are SQLsmith [Seltenreich, 2020] and SQLfuzz [Jung et al., 2019], two SQL query generators that respectively target crashes and regression bugs in popular DBMSs. The approach behind CYNTHIA differs from all these query generators because it produces queries in a higher-level query language (AQL) and adopts differential testing to detect logic errors beyond crashes or regression bugs. Abdul Khalek et al. [2008]; Abdul Khalek and Khurshid [2010] followed a solver-based approach for testing DBMSs. Their work employs a relational constraint solver to generate valid database records with respect to a given SQL query and database schema. Besides populating the database, their method also determines the expected results of an SQL query and the authors use this oracle to find bugs. CYNTHIA also uses an SMT solver to populate the database, but we specify the test oracle by adopting a differential testing approach.

More recently, Rigger and Su [2020c] proposed the *Pivoted Query Synthesis* (PQS) technique for testing database engines. PQS generates SQL queries so that they fetch a specific record from the database. In this way, PQS forms the test oracle: failing to fetch the expected record reveals a potential bug in DBMS. Unlike this work, our approach adopts differential testing for determining the oracle. Also, beyond reasoning about a single record, our approach is able to detect bugs involving operations on result sets (e.g., aggregate functions, sorting, distinct). In an attempt to find optimization bugs in database systems, their subsequent work introduced a metamorphic testing technique called *Non-Optimizing Reference Engine Construction* (NoREC) [Rigger and Su, 2020a]. At a high-level, NoREC applies a semantics-preserving transformation to a given SQL query in way that the various optimizations performed by the DBMS are disabled. Finally, NoREC compares the results of the original and the resulting queries for mismatches. In their most recent work, they propose *Ternary Logic Partitioning* (TLP) [Rigger and Su, 2020b]. Given an SQL query, TLP derives multiple queries that compute a partial result of the initial query, and then combines the results of each individual query using a `UNION` operation. If the result of the combined query does not match that of the initial one, then a bug is found. TLP is suitable for testing the implementation of the `WHERE`, `HAVING`, `DISTINCT` clauses, or aggregate functions.

All these previous approaches are tailored to testing DBMSs, i.e., they aim to find DBMS-specific bugs (e.g., optimization bugs, bugs associated with the evaluation of `WHERE` clauses). ORM systems differ from database engines, and suffer from other types of bugs.

6.3 Related Work on Detecting Dependency Bugs in File System Resources

In this section, we provide details about the existing work related to the FSMove model presented in Section 3.3. We group this related work into three categories: *quality and reliability in Infrastructure as Code (IaC)*, *quality and reliability in builds*, and *trace analysis*.

6.3.1 Quality and Reliability in Infrastructure as Code (IaC)

With the proliferation of the IaC process, there have been numerous attempts to identify defects and quality concerns in configuration code.

A number of studies focus on maintainability issues. Sharma et al. [2016] design and implement a code-smell detection scheme for Puppet, which searches for issues related to naming conventions, code design, indentation, etc. Their findings suggest that such anti-patterns—as in the traditional programs—exist in many IaC repositories. van der Bent et al. [2018] introduce a quality model for Puppet programs which is empirically evaluated by interviewing practitioners from industry. Schwarz et al. [2018] do similar work focusing on Chef recipes. Endeavors have recently moved to the identification of security issues. Rahman et al. [2019] define and classify security smells into seven categories (such as hard-coded passwords, use of weak cryptographic algorithms), and then build a tool for statically detecting these smells in Puppet repositories.

Other studies attempt to extract error patterns and source code properties from the analysis of defective IaC programs. Rahman and Williams [2018] employ machine learning and text processing techniques to identify properties that buggy Puppet programs hold. Then, they build a prediction model for asserting whether IaC scripts manifest bugs or not. Chen et al. [2018] identify error patterns in Puppet manifests by following a different approach. First, they inspect the code changes from repositories' commits. Second, they construct an unsupervised learning model to detect error patterns based on the clustering of the proposed fixes. Their approach is based on the assumption that similar faults are fixed with similar patches [Hanam et al., 2016].

There are few automated techniques proposed for improving the reliability of configuration management programs. Rehearsal [Shambaugh et al., 2016] statically verifies that a given Puppet configuration is deterministic and idempotent. Rehearsal models a given Puppet manifest in a small language called FS and then it constructs logical formulas based on the semantics of each language's primitive. Then, an SMT solver decides whether the initial program is non-deterministic or not. Compared to FSMove, Rehearsal is less effective and practical. Specifically, Rehearsal employs a form of static analysis that can only handle a subset of Puppet programs. For example, the analysis does not support `exec` resources because it is unable to reason about the file system resources that shell commands process. Unlike Rehearsal, FSMove works by reasoning actual system calls rather than Puppet manifests; thus, it can effectively determine which files are affected by a particular Puppet run and how.

Other advances [Hummer et al., 2013; Hanappi et al., 2016] adopt a model-based testing approach for checking whether configuration scripts meet certain properties. Hummer et

al. [Hummer et al., 2013] focus on testing the idempotence of Chef scripts. Their proposed framework generates multiple test cases that explore different task schedules. By tracking the changes in the system trigged by a Chef script, they determine if idempotence holds for the given program. Hanappi et al. [Hanappi et al., 2016] extend the work of Hummer et al. and introduce Citac; a framework that can be applied to Puppet manifests to examine the convergence of programs. Convergence states that the system reaches a desired state even at the presence of failed Puppet resources. They formally express the properties of idempotence and convergence, and through test case generation, they verify if the provided manifests violate those properties. Contrary to Citac, we adopt a more lightweight approach applying manifests only once. Finally, neither Rehearsal nor Citac detect issues involving missing notifiers.

Tortoise [Weiss et al., 2017] employs a semi-automated approach for repairing buggy Puppet scripts. In particular, it monitors the fixes made by administrators using the command-line, and follows a program-synthesis-based technique to generate a patch for the corresponding Puppet manifests. This patch reflects all the file system changes made by administrators through the execution of shell commands.

6.3.2 Quality and Reliability in Builds

Testing and debugging builds: Testing build scripts is an emerging research area. mkcheck [Licker and Rice, 2019], and BEE [Bezemer et al., 2017] are two tools that also detect missing inputs, but they are tailored for Make-based builds. As we pointed out in Section 2.4.3, these tools have two important limitations that concern: (1) low precision when applied to JVM build tools, and (2) efficiency & applicability. As we explained earlier, our FSMove model tackles both limitations.

Beyond testing, a number of studies have been developed to identify the root causes of problematic builds, and suggest fixes for them. Al-Kofahi et al. [2014] have designed a tool that given a failed build, it identifies the faulty Make rules that caused the build crash. Their approach performs an instrumentation on a Make build that tracks the execution trace of each Make rule, and records the crash point of the build. Based on a probabilistic model, they assign different scores to every rule, indicating the probability that the rule caused the crash. Subsequent work [Ren et al., 2018; Ren et al., 2019] have focused on locating faults of unreproducible builds. Reproducibility is a property that ensures that a build is deterministic and always results in bitwise-identical targets given the same sources and build environment. The initial work of Ren et al. [2018] analyzes the logs from an unreproducible build, and proposes a ranked list of problematic source files that might contain the fault. Recently, the authors extended their approach [Ren et al., 2019] to locate the specific command that is responsible for the unreproducible build. To do so, they employed a backtracking analysis on system call trace stemming from build execution. Contrary to these approaches, FSMove does not require a build failure, but it is capable of detecting latent future faults.

Scott et al. [2017] develop `detmake`, a system that uncovers failures in Make builds by enforcing a deterministic execution of parallel builds. To do so, the technique behind `detmake` provides arbitrary system operations (such as file system operations) with deterministic se-

mantics and side-effects by intercepting system calls and libraries at runtime. Running a Make build through `detmake` always results in a build failure when the corresponding build scripts contain ordering violations. The developers then need to manually find and fix the root cause of the failure to run their builds successfully. `detmake` is not fully compatible with GNU Make as the technique behind this tool comes with several restrictions for the supported builds. Unlike `detmake`, FSMove focuses on fault localization: we report the *exact* build rule (i.e., task) whose definition is incorrect, and can potentially lead to issues. Further, the method behind FSMove is also capable of identifying *subtle* errors that lead to wrong build outcomes rather than failures. Finally, the work of Scott et al. [2017] does not reason about issues related to incremental builds.

In an attempt to repair build failures, Hassan and Wang [2018] have introduced a technique for automatically recommending patches that in turn, are based on patterns extracted from historical fixes of failed Gradle builds.

Understanding and refactoring builds: There are plenty of tools developed over the past decade to assist developers in understanding and refactoring builds. *Makao* [Adams et al., 2007] is a Make-related framework used for visualizing build dependencies. By extracting knowledge from such dependencies through filtering and querying, Makao provides support for refactoring build scripts via an aspect-oriented approach. *SYMake* [Tamrawi et al., 2012], evaluates Makefiles and produces (1) a symbolic dependency graph, and (2) a symbolic execution trace. Then, it applies different algorithms to the results to detect a number of code smells (e.g., cyclic dependencies), and perform refactoring on Make scripts (e.g., target renaming). Metamorphosis [Gligoric et al., 2014a] is a tool used to migrate existing build scripts to CloudMake Christakis et al. [2014]. As a starting point, Metamorphosis analyzes the execution trace of a given build and then automatically synthesizes an initial CloudMake script that reflects the behavior of the original script. Then, it optimizes the build script synthesized by the previous step by applying a sequence of transformations and choosing the best possible ones based on a fitness function. Vakilian et al. [2015] propose a new refactoring method, target decomposition, for dealing with underutilized targets; a build-related code smell that causes slower builds, larger binaries, and less modular code.

Formally-verified build systems: CloudMake [Christakis et al., 2014] is a modern build system developed by Microsoft. It supports both incremental and parallel builds, and exposes a functional interface in JavaScript. Developers can invoke external tools (e.g., compilers, linkers, other arbitrary programs), and define their own build tasks using the `exec` primitive. When calling an external tool via `exec`, developers need to explicitly enumerate the dependencies of this tool. Then, CloudMake internally uses these static dependencies to (1) form a parallel schedule, and (2) verify that each tool accesses only the files provided by the developer.

Christakis et al. [2014] introduce a language and a semantics for builds, and use them to formally verify some of the core algorithms of CloudMake, namely parallel and cached builds (but not incremental builds). Although CloudMake tackles similar issues, FSMove follows a different approach. New build systems face adoption issues [McIntosh et al., 2015; Gligoric et al., 2014a], and migrating existing build scripts to a new (formally-verified) build tool may require significant engineering effort [McIntosh et al., 2015; Gligoric et al., 2014a]. Therefore, instead of designing a new build system, our thesis introduces a generally-applicable method that detects

issues in builds written in existing systems without requiring extensive modifications to the underlying build scripts. To demonstrate applicability, we apply our FSMove method to two build systems with diverse build specification languages and designs that hold a significant stake in build technology [McIntosh et al., 2015; Hassan et al., 2017].

6.3.3 Trace Analysis

The FSMove tool relies on system call tracing for capturing the effects of every task on the file system. Analysis of (system call) traces has been widely used in the past, especially in the context of dependency inference [Gligoric et al., 2014a; van der Burg et al., 2014; Licker and Rice, 2019; Ammons, 2006; Ren et al., 2019], refactoring and testing [Gligoric et al., 2014b], boosting performance of builds [Ammons, 2006; Coetzee et al., 2011], or detecting license inconsistencies in open-source projects [van der Burg et al., 2014]. Our work differs from the previous approaches as the proposed FSMove model, and in turn, its practical realization captures both the dynamic behavior and the static specification of high-level programming constructs (i.e., build tasks, Puppet resources). This enables us to verify—contrary to existing approaches—the execution of a task with regards to its specification, while monitoring the execution, making our method more precise, efficient, and generally-applicable.

Mutlu et al. [2015] collect execution traces from JavaScript applications. Their traces do not track system calls, but they capture memory and storage (e.g., cookies) accesses in the context of the browser. They split traces into blocks, where each block describes the execution of an asynchronous callback (e.g., AJAX handler). As the execution of each handler is partially ordered, they apply a simple data-flow analysis over traces to join the states coming from different handlers. In this manner, they effectively detect data races by identifying handler pairs where the merges of their corresponding states result in different values of the same variable. In our work, we also separate the trace sequences into blocks. However, we are interested in file system operations instead of reads and writes to memory locations. Also, we apply a different method for discovering execution blocks that might lead to harmful scenarios.

Chapter 7

Conclusion and Future Work

The focus of this thesis is to improve the effectiveness of software testing by studying different forms of abstractions in the context of three important classes of bugs, namely, compiler typing bugs, bugs in data-centric software, and dependency bugs in file system resources.

7.1 Summary

Designing proper abstractions and bug-finding methods for the discovery of specific types of bugs is not an easy task. To this end, we first presented a method for collecting and analyzing existing bugs found in complicated software systems, such as compilers. The purpose of this bug analysis is to get insights about the characteristics and manifestations of highly frequent and reoccurring bugs. To do so, our method involves two main steps: (1) the collection of *fixed* bugs taken from the issue tracker of an examined software system, and (2) the manual analysis of the collected bugs by taking into account important features, including bug symptoms, bug causes, bug fixes, and test case characteristics.

The realization of our bug collection and assessment method was in the context of 320 fixed typing bugs found in compilers of four popular JVM languages that is, Java, Scala, Kotlin, and Groovy. Our bug analysis provides an excellent source of findings and implications for compiler developers and automated compiler bug detection.

Particularly, we found that unlike optimization bugs, compiler typing bugs have plenty of manifestations that span from compilation to the runtime. Correctness issues found in the core components of compiler typing processes, such as type system implementations, type inference and resolution engines, or the semantic validation of declarations, are responsible for the majority of the inspected bugs. Moreover, although front-end bugs are typically fixed without requiring extensive modifications in compilers' code base, compiler developers need a few months to resolve a bug, as they carefully assess the impact of each fix to prevent regression bugs. We also found that a non-trivial number of typing bugs is caused by non-compilable test cases, whereas loops and arithmetic expressions are hardly seen in the bug-revealing programs. These observations conflict with the intuition behind the existing approaches for finding compiler optimization bugs.

We also discussed several implications of our study's findings. Future testing techniques

should consider diverse test oracles, as compiler typing bugs affect both compilation and runtime in various ways. Another interesting future challenge is the generation of subtly invalid programs that are more likely to trigger typing-related bugs and, most notably, soundness issues. Furthermore, the existing program generators for C++ could benefit from the results of our study: their generation strategies could be adapted to include features (e.g., type inference, lambdas, overloading) that can potentially uncover inference or resolution bugs in the C++ compilers.

Based on the findings and observations of this bug study, we next presented a systematic and extensible approach for finding typing bugs in compilers (Section 3.1). To do so, we introduced a program generator that relies on an abstract input. This abstract input corresponds to programs (written in an intermediate language) that exercise type-intensive operations in way that the generated programs are promising for triggering compiler typing bugs. Based on this generator, we have designed two novel transformation-based approaches for uncovering type inference and soundness compiler bugs. Within nine months of testing, our implementation, HEPHAESTUS, has found 170 bugs (148 confirmed and 104 fixed) in more than one compilers: `javac`, `groovyc`, and `kotlinc`.

Next, to improve the reliability of data-centric software, and in particular that of Object-Relational Mapping (ORM) systems, we introduced a data-oriented differential testing approach (Section 3.2). A fundamental requirement for differential testing is that the implementations under test must be equivalent. By introducing an appropriate layer of abstraction that hides the implementation differences, we showed that differential testing can be also applicable in systems with (seemingly) dissimilar interfaces, such as ORMs. Another benefit of our abstractions was the ability to test the resulting ORM queries on various databases, such as MySQL, SQLite. This allowed us to find ORM bugs whose occurrence was dependent on running a buggy ORM query on a specific database management system. Further, we addressed an ORM-specific challenge: the generation of data that are guaranteed to produce non-trivial results in response to given queries. To do so, we employed an SMT solver to synthesize targeted records, dependant on the constraints of the generated inputs. Our findings showed that when compared to other simplistic data generation strategies, the solver-based approach enhances the bug detection capability. We demonstrated the importance and practicality of our data-oriented approach by systematically testing five popular open-source ORM systems using our implementation called CYNTHIA. CYNTHIA discovered 28 bugs, most of which have been fixed by the developers.

Finally, for dealing with dependency bugs in file system resources, this thesis proposed and developed a generic and practical approach based on a model known as FSMOVE (Section 3.3). The proposed model incorporates the static specification (declared dependencies) and the dynamic behavior (effects on the file system) of arbitrary executions. We then formally defined four different types of dependency bugs concerning file system resources, and presented a testing approach for exploiting FSMOVE and detecting such dependency bugs in practice. Combining static and dynamic information in a single representation made our method efficient and applicable, as our verification approach neither involves static analysis or re-execution of the scripts under test. The effectiveness and applicability of our FSMOVE method is exemplified by uncovering issues in hundreds of configuration management and build scripts, including Make,

Gradle, and Puppet programs. Notably, our approach tackled the limitations of existing work, and it is the first to deal with tools that use only one system process to govern the execution. We demonstrated the importance of the discovered dependency bugs by providing patches to numerous projects. Thanks to FSMove, the developers of 72 open-source projects confirmed and fixed 300 issues, in total. Moreover, a comparison between FSMove and a state-of-the-art Make-based tool showed that our approach is more effective and significantly more efficient (74× faster on average).

7.2 Thesis Takeaway

The main takeaway that stems from this thesis is the following:

> *Introducing appropriate abstractions, such as a test input abstraction or an execution abstraction, can improve the effectiveness of software testing in exposing bugs (including subtle and latent defects) in software that manifests complex functionality. These abstractions should be carefully crafted so that they hide unnecessary details, facilitate reasoning for bug identification, and promote applicability.*

Detection of latent and subtle bugs: Our abstraction-based techniques allow us to discover dozens of subtle and latent bugs that are not obvious for the programmers.

Specifically, our test input abstraction enables CYNTHIA to detect ORM bugs that manifest only when ORM code runs on top of a specific database engine. Detecting such bugs is of high importance, because developers typically use different database engines in a testing vs. a production environment. To better illustrate this, imagine an ORM-backed application that uses a lightweight SQLite database during testing, and a PostgreSQL server for production. If the application relies on a ORM system that contains a PostgreSQL-dependent bug, the unexpected behaviors that stem from this bug slip through the testing efforts of the application, as the bug is triggered only when a PostgreSQL database is enabled.

The abstractions of FSMove makes it possible to detect various dependency bugs that come from seemingly benign, Puppet and build executions. For example, FSMove does not require the build or the configuration management script to crash in order to reason about the root cause of the failure. Instead, FSMove is still capable of reporting dependency bugs even if (1) it monitors an execution where operations (randomly) take place in the right order, or (2) they appear in subsequent executions (e.g., detecting bugs related to incremental builds by monitoring a full build).

Finally, the abstractions of HEPHAESTUS lead to the discovery of soundness bugs where the compiler silently accepts a non-trivially type incorrect program. In addition to that, HEPHAESTUS detects cases where the compiler rejects a well-typed program, while it is not obvious for the programmer to determine whether this is due to a fault in their input program, or a bug in the compiler. Interestingly, some of the discovered bugs found during our evaluation (Section 5.1) were associated with deep flaws in language designs (e.g., see KT-44082).

Ignoring irrelevant details: Our abstractions are designed to hide details that are irrelevant to the problem these abstractions cope with. Intuitively, this makes our abstraction-

based approaches simpler. For example, HEPHAESTUS's IR does not support features that do not frequently lead to typing bugs (i.e., arithmetic expressions, complex control flow, access modifiers). CYNTHIA's abstractions omit column constraints, indexes, and other metadata of relational database schemas that do not affect the translation mechanism of ORMs. Our FSMOVE model does not capture aspects of the system (sockets, memory, pipes) not associated with dependency bugs in file system resources.

Facilitating reasoning: Our abstractions provide us with a platform for reasoning about certain properties of the system under test and its input. The abstraction of type graph along with its operations and properties allow HEPHAESTUS to model the problem of removing program type information and injecting a subtle type error into a program as graph reachability problems. The simplicity of AQL enables the modeling of the data generation problem as a constraint problem. In a similar way, the model of FSMOVE incorporates the static specification *and* the dynamic behavior of tasks. This leads to the identification of dependency bugs without (1) requiring static analysis on configuration management and build scripts, which are written in different DSLs, or (2) further script executions (e.g., incremental builds for deciding build correctness as prior work does [Licker and Rice, 2019]).

Applicability / generalizability: Finally, it is more than evident that our abstractions promote the applicability of the proposed underlying methods. Our in-depth evaluation demonstrates that all the techniques of this thesis can be applied to software systems that potentially involve dissimilar interfaces, implementations, and semantics. Later, in Section 7.4, we explain how our work can be extended to detect the classes of bugs described in this thesis in programs and systems other than those that are currently supported by our tools.

7.3 Implications on the Software Industry

The evaluation results of our methods are very promising. Overall, using the bug-finding tools provided by this thesis, we were able to find thousands of bugs in complex and well-established software. *most* importantly, we were able communicate these bugs to the corresponding development teams by the submission of *more than 300 bug reports and pull requests* (see Appendix E for more information). Through our bug reports and patches, the development teams were able to *fix 424 bugs*, and in turn, *improve the reliability of their software*. Notably, developers acknowledged (see the acknowledgement in Groovy's release notes [King, 2022]) and encouraged our testing efforts through the swift fix of our bug reports and the introduction of supportive comments, such as *"Thank you for the fix!"*, *"Thank you for pointing this out!"*, and more.

We argue that our tools can be integrated into the software testing pipelines of the corresponding development teams. Developers can benefit from this as follows. First, our tools can help developers discover software defects at an early stage. For example, due to its low overhead, FSMOVE can run as part of CI scripts triggered every time developers make a change in their configuration management and build scripts. Second, our tools can prevent both regression and long-latent bugs by improving the effectiveness of developers' test suites. For example, in Sections 5.1.2 and 5.2.2, we showed that plenty of issues discovered by CYNTHIA and HEPHAESTUS were regression bugs. Therefore, CYNTHIA and HEPHAESTUS can be useful

for assisting developers in extending their test suites with test inputs that exercise interesting SUT's behaviors. We believe that CYNTHIA and HEPHAESTUS can potentially exhibit similar value for developers, as `GraphicsFuzz` did for Google developers [Donaldson et al., 2020].

Using our tools, we discovered bugs with a plethora of consequences, and characteristics. There were long-lasting bugs that remained unnoticed for years, regression bugs that lied in development branches, bugs that could potentially lead to data loss, miscompilations, wrong application results, runtime failures, and more. Using the bug reports produced by our tool, we grouped the discovered bugs into categories based on their characteristics and root causes. This categorization can be helpful for the developers and practitioners to identify (1) which are the main pitfalls when developing their software, (2) where to focus their development and testing efforts.

Another important aspect for the software industry that stems from work is applicability / generalizability. We believe that our work minimizes the waste of engineering effort, as developers and testers (e.g., compiler developers, ORM developers) can adapt our techniques to test a new piece of software.

7.4 Future Work

The methods presented in this thesis entail a number of promising directions for further research and work:

Extending the proposed frameworks for enhancing the reliability of other advanced software: A first interesting future direction could be the extension of the proposed methods and frameworks for detecting typing bugs, data-related bugs, and dependency bugs in other advanced software systems. For example, we could extend HEPHAESTUS to test the implementation of other compilers, such as the Scala and TypeScript compilers. More interestingly, it would be nice to investigate how effective the techniques implemented in HEPHAESTUS are in revealing typing bugs in languages with completely different typing algorithms from those studied in this thesis (see Haskell, OCaml, Go, and Rust, or even verification platforms like Dafny and Coq [Irfan et al., 2022]).

The data-oriented differential approach for testing ORM implementations could be also useful in testing other data-centric software, including stream libraries and frameworks, or NoSQL database engines. For example, as an effort to find bugs in the standard library of Java, the functional nature of AQL could be leveraged for the generation of stream pipelines (i.e., queries) that exercise the Stream API[1] of Java in a complex manner.

Similarly, FSMOVE is a generic model that opens up many opportunities: beyond running FSMOVE to detect dependency bugs in scripts written in other build and configuration management systems (Ninja, Ansible), we could apply FSMOVE to other domains with partially ordered constructs, such as the asynchronous callbacks of JavaScript. Recent work [Wang et al., 2017; Davis et al., 2017] has showed that many concurrency bugs in Node.js applications are caused by data races that appear in files instead of memory locations. Therefore, we could

[1] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

build upon FSMove to detect such concurrency faults in JavaScript server-side applications. To instrument Node.js programs, we could consider various alternatives: (1) static instrumentation on source code to enable *async hooks*[2], a module for running code before and after the execution of asynchronous callbacks, or (2) dynamic instrumentation on node binary to monitor file system operations and capture the inter-dependencies of asynchronous callbacks by instrumenting the underlying `libuv`[3] calls. Notably, `libuv` is the library used by Node.js for performing asynchronous I/O.

Designing sophisticated mutations for compiler typing bugs: Further sophisticated mutations could be developed on top of Hephaestus's IR. For example, a promising direction could be the development of a mutation that targets bugs in the resolution algorithms of compilers, a category of bugs that is quite frequent (see Section 2.2.4). Such a mutation could modify an input program in a way where new overloaded methods are derived from a specific call-site. The goal of this mutation is to locate resolution bugs that come from cases where given a specific call-site, the buggy compiler fails to resolve the correct overloaded method.

Targeting soundness compiler bugs: Language designers strive to develop expressive, yet *sound* type systems that work well with the rest features of their languages. Despite sound in theory, we have shown that language implementations can be unsound in practice due to typing bugs found in the corresponding compilers. Although pretty simple at conceptual level, our type overwriting mutation produced promising results (Section 3.1.4.2): we already identified a couple of interesting soundness bugs in well-established compilers. Because of the potential security implications, it is worth spending more time on detecting soundness bugs. Researchers could focus their endeavors on designing testing methods that increase the confidence of developers that compiler implementations do not accept ill-typed programs. To do so, future researchers could build upon the abstraction of type graph and inject type errors by taking into account both type information flow and type shapes. The ultimate goal of this mutation is to create programs so that it is less obvious for the compiler to locate the injected type error.

Smarter program generation: A key challenge when building program generators like the ones of Hephaestus and CYNTHIA is the creation of small, yet expressive programs so that a bug-revealing test case produced by the program generator can be sent directly to developers without employing any form of test case reduction [Regehr et al., 2012]. Currently, the program generators of Hephaestus and CYNTHIA result in large programs consisting of multiple lines of code. Instead of performing manual or automated test case reduction, another interesting direction could be the design of novel generation algorithms that are capable of creating small programs that exhibit high bug-triggering capability.

For example, one of the findings of the bug study we conducted on previously-reported compiler typing bugs indicates that a large portion of these bugs is triggered by test cases that use the standard library, and especially the collection API (Section 2.2.7). Inspired by this observation and other program synthesis techniques for testing libraries [Takashima et al., 2021], we could design a program generator that yields small and self-contained programs that use collection API (or other similar APIs) in complex manners. Producing small test inputs has

[2] https://nodejs.org/api/async_hooks.html
[3] https://github.com/libuv/libuv

the following benefits. First, there is no need for test case reduction. Hephaestus currently produces programs consisting of roughly 500-1k lines of code. Therefore, we need to manually reduce test cases before opening bug reports. Second, having small test inputs implies a small search space for mutations. Mutations typically perform small changes in the input program by modifying those program locations that are more promising for revealing bugs. When the input program is large, there is an astronomical number of such candidate program locations. A small test input offers a significantly smaller search space for mutations, while preserving much of the complexity of a large program.

Testing other aspects of ORMs: Beyond studying the translation mechanisms employed by ORM systems, future work could examine the reliability of other important aspects of these systems, including transaction management, insert, update, or delete operations, or migrations. For example, AQL currently supports only read operations. The implementations of ORM API methods associated with read operations are much more complex than those related to write operations (a write operation is straightforwardly translated into `INSERT`, `DELETE` or `UPDATE` queries). Thus, examining read operations for finding bugs is more promising. Note though that AQL can be easily extended for supporting write queries. Also, supporting write operations would not require to take into account schema properties that we are currently ignoring (e.g., column constraints), because such properties affect the configuration of ORM models and not the way an ORM translates a write query into an SQL statement.

Automated program repair and improvement of DSLs: The fixes of the dependency bugs discovered by FSMove contain very few lines of code, i.e., the bugs are typically fixed in 1–2 lines. Extending our FSMove approach to automatically generate fixes for the reported bugs (as in the work of Weiss et al. [2017]) is an interesting future direction. For example, based on the issues reported by FSMove, we could identify where to apply the fix by inspecting the AST of Puppet manifests to add missing dependencies and notifiers. At the same time, new and existing build and configuration management systems could benefit from FSMove, and provide means for managing dependencies automatically, simplifying their DSLs and specification languages.

Today's software is getting more and more complex, exposes intricate functionalities, while at the same time, it affects millions of users around the globe. Software testing is naturally at the heart of every software development workflow, and its effectiveness has a tremendous impact on every single individual. We hope this thesis help in the design of effective bug-finding tools that can assist developers and testers in improving the quality, reliability, and robustness of modern software. Furthermore, we believe that this work will inspire researchers to continue their efforts at understanding and improving the fundamental software that most of the world's infrastructure relies on, such as compilers, data-oriented systems, or software that automates the process of system configuration and provision.

Appendix A

Supplementary Material of Section 2.2

Table A.1: Distribution of language features. Each entry contains the frequency of a language feature in the examined test cases. This table contains the full data of Figure 2.20

Feature	Category	# Test Cases	Feature	Category	# Test Cases
Lambda	Functional programming	61	Parameterized function	Parametric polymorphism	84
SAM type	Functional programming	43	Bounded type parameters	Parametric polymorphism	57
Function reference	Functional programming	28	Use-site variance	Parametric polymorphism	17
Function type	Functional programming	24	F-bounds	Parametric polymorphism	14
Eta expansion	Functional programming	2	Declaration-site variance	Parametric polymorphism	12
Inheritance	OOP features	77	Higher-kinded type	Parametric polymorphism	11
Overriding	OOP features	50	Multi-bounds	Parametric polymorphism	3
Overloading	OOP features	36	Conditionals	Standard language features	21
Nested class	OOP features	25	Array	Standard language features	18
Anonymous class	OOP features	18	Cast	Standard language features	15
Access modifier	OOP features	16	Import	Standard language features	11
Singleton object	OOP features	14	Variable arguments	Standard language features	11
Property	OOP features	13	Try / catch	Standard language features	10
Case class	OOP features	12	Augmented assignment operator	Standard language features	8
Special method overriding	OOP features	10	Enums	Standard language features	7
Static method	OOP features	9	Arithmetic expression	Standard language features	6
Operator overloading	OOP features	7	Loops	Standard language features	4
Multiple implements	OOP features	7	Collection API	Standard library	70
This	OOP features	7	Function API	Standard library	17
Secondary constructor	OOP features	5	Reflection API	Standard library	12
Delegation	OOP features	4	Coroutines API	Standard library	4
Sealed class	OOP features	3	Stream API	Standard library	2
Self type	OOP features	3	Delegation API	Standard library	1
Property reference	OOP features	3	Type argument inference	Type inference	102
Value class	OOP features	2	Parameter type inference	Type inference	26
Data class	OOP features	1	Variable type inference	Type inference	20
Implicits	Other	19	Flow typing	Type inference	11
Java interoperability	Other	17	Build-style inference	Type inference	4
Pattern matching	Other	17	Return type inference	Type inference	2
Extension function / property	Other	12	Subtyping	Type system-related features	52
Type annotations	Other	9	Wildcard type	Type system-related features	24
Named arguments	Other	7	Type definition / member	Type system-related features	16
Option type	Other	5	Primitive type	Type system-related features	14
Elvis operator	Other	5	Nullable type	Type system-related features	13
Call by name	Other	4	Algebraic data type	Type system-related features	11
Inline	Other	4	Dependent type	Type system-related features	7
Template string	Other	3	Intersection type	Type system-related features	5
Safe navigation operator	Other	2	Nothing	Type system-related features	3
Erased parameter	Other	1	Type lambdas	Type system-related features	3
Default initializer	Other	1	Type projection	Type system-related features	3
Null assertion	Other	1	Union type	Type system-related features	2
With	Other	1	Opaque type	Type system-related features	1
Parameterized type	Parametric polymorphism	149	Mixins	Type system-related features	1
Parameterized class	Parametric polymorphism	96	Match type	Type system-related features	1

Table A.2: The 30 most frequent and the 30 least frequent features supported by all studied languages.

Most frequent features		Least frequent features	
Feature	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	46.56%	Subtyping	16.25%
Type argument inference	31.87%	Overriding	15.62%
Parameterized class	30.00%	SAM type	13.44%
Parameterized function	26.25%	Overloading	11.25%
Inheritance	24.06%	Function reference	8.75%
Collection API	21.88%	Parameter type inference	8.12%
Lambda	19.06%	Nested class	7.81%
Bounded type parameters	17.81%	Wildcard type	7.50%
Subtyping	16.25%	Function type	7.50%
Overriding	15.62%	Conditionals	6.56%
SAM type	13.44%	Variable type inference	6.25%
Overloading	11.25%	Anonymous class	5.62%
Function reference	8.75%	Array	5.62%
Parameter type inference	8.12%	Use-site variance	5.31%
Nested class	7.81%	Function API	5.31%
Wildcard type	7.50%	Java interoperability	5.31%
Function type	7.50%	Access modifier	5.00%
Conditionals	6.56%	Cast	4.69%
Variable type inference	6.25%	Property	4.06%
Anonymous class	5.62%	Reflection API	3.75%
Array	5.62%	Import	3.44%
Use-site variance	5.31%	Variable arguments	3.44%
Function API	5.31%	Try / catch	3.12%
Java interoperability	5.31%	Augmented assignment operator	2.50%
Access modifier	5.00%	Multiple implements	2.19%
Cast	4.69%	This	2.19%
Property	4.06%	Enums	2.19%
Reflection API	3.75%	Arithmetic expression	1.88%
Import	3.44%	Loops	1.25%
Variable arguments	3.44%	Sealed class	0.94%

Table A.3: The 20 most bug-triggering features per language.

Java		Scala		Kotlin		Groovy	
Feature	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	51.25%	Parameterized type	41.25%	Parameterized type	36.25%	Parameterized type	57.50
Type argument inference	42.50%	Collection API	35.00%	Parameterized class	33.75%	Parameterized class	42.50
SAM type	37.50%	Type argument inference	35.00%	Type argument inference	32.50%	Inheritance	32.50
Parameterized function	35.00%	Lambda	25.00%	Parameterized function	26.25%	Implicits	23.75
Parameterized class	30.00%	Parameterized function	21.25%	Inheritance	25.00%	Parameterized function	22.50
Inheritance	22.50%	Subtyping	21.25%	Lambda	25.00%	Pattern matching	21.25
Collection API	22.50%	Parameter type inference	17.50%	Function type	17.50%	Type argument inference	17.50
Subtyping	21.25%	SAM type	15.00%	Nullable type	16.25%	Type definition / member	17.50
Lambda	21.25%	Parameterized class	13.75%	Extension function / property	15.00%	Bounded type parameters	17.50
Bounded type parameters	20.00%	Inheritance	13.75%	Collection API	13.75%	Collection API	16.25
Function reference	17.50%	Primitive type	12.50%	Conditionals	13.75%	Singleton object	15.00
Overloading	16.25%	Overriding	11.25%	Function reference	13.75%	Case class	15.00
Function API	15.00%	Variable type inference	10.00%	Overriding	13.75%	Higher-kinded type	13.75
Overriding	13.75%	Property	10.00%	Subtyping	12.50%	Algebraic data type	13.75
Use-site variance	12.50%	Array	8.75%	Flow typing	8.75%	Function type	12.50
Cast	11.25%	Named arguments	6.25%	Bounded type parameters	8.75%	Wildcard type	12.50
Nested class	10.00%	Access modifier	6.25%	Wildcard type	8.75%	Special method overriding	12.50
Conditionals	10.00%	Java interoperability	6.25%	Java interoperability	8.75%	Overriding	11.25
Array	10.00%	Flow typing	5.00%	Variable type inference	8.75%	Subtyping	10.00
Anonymous class	8.75%	Overloading	5.00%	Operator overloading	8.75%	Nested class	10.00

Appendix B

The Command-Line Interface of Hephaestus

The command-line interface of Hephaestus is shown in the following:

```
usage: hephaestus.py [-h] [-s SECONDS] [-i ITERATIONS] [-t TRANSFORMATIONS] [--batch
    BATCH] [-b BUGS] [-n NAME] [-T [{TypeErasure} [{TypeErasure} ...]]]
                     [--transformation-schedule TRANSFORMATION_SCHEDULE] [-R REPLAY] [-e]
                         [-k] [-S] [-w WORKERS] [-d] [-r] [-F LOG_FILE] [-L] [-N] [--
                         language {kotlin,groovy,java}]
                     [--max-type-params MAX_TYPE_PARAMS] [--max-depth MAX_DEPTH] [-P] [--
                         timeout TIMEOUT] [--cast-numbers] [--disable-use-site-variance]
                         [--disable-contravariance-use-site]
                     [--disable-bounded-type-parameters] [--disable-parameterized-
                         functions]

optional arguments:
  -h, --help show this help message and exit
  -s SECONDS, --seconds SECONDS
                        Timeout in seconds
  -i ITERATIONS, --iterations ITERATIONS
                        Iterations to run (default: 3)
  -t TRANSFORMATIONS, --transformations TRANSFORMATIONS
                        Number of transformations in each round (default: 0)
  --batch BATCH Number of programs to generate before invoking the compiler
  -b BUGS, --bugs BUGS Set bug directory (default: /home/hephaestus/bugs)
  -n NAME, --name NAME Set name of this testing instance (default: random string)
  -T [{TypeErasure} [{TypeErasure} ...]], --transformation-types [{TypeErasure} [{
      TypeErasure} ...]]
                        Select specific transformations to perform
  --transformation-schedule TRANSFORMATION_SCHEDULE
                        A file containing the schedule of transformations
  -R REPLAY, --replay REPLAY
                        Give a program to use instead of a randomly generated (pickled)
  -e, --examine Open ipdb for a program (can be used only with --replay option)
  -k, --keep-all Save all programs
  -S, --print-stacktrace
```

```
                    When an error occurs print stack trace
-w WORKERS, --workers WORKERS
                    Number of workers for processing test programs
-d, --debug
-r, --rerun Run only the last transformation. If failed, start from the last and go
    back until the transformation introduces the error
-F LOG_FILE, --log-file LOG_FILE
                    Set log file (default: /home/hephaestus/logs)
-L, --log Keep logs for each transformation (bugs/session/logs)
-N, --dry-run Do not compile the programs
--language {kotlin,groovy,java}
                    Select specific language
--max-type-params MAX_TYPE_PARAMS
                    Maximum number of type parameters to generate
--max-depth MAX_DEPTH
                    Generate programs up to the given depth
-P, --only-correctness-preserving-transformations
                    Use only correctness-preserving transformations
--timeout TIMEOUT Timeout for transformations (in seconds)
--cast-numbers Cast numeric constants to their actual type (this option is used to
    avoid re-occrrence of a specific Groovy bug)
--disable-use-site-variance
                    Disable use-site variance
--disable-contravariance-use-site
                    Disable contravariance in use-site variance
--disable-bounded-type-parameters
                    Disable bounded type parameters
--disable-parameterized-functions
                    Disable parameterized functions
```

Below, there is a brief description for each command-line option:

–bugs (Optional): Set the directory to save the results of the testing session. The default directory is `$(pwd)/bugs`.

–name (Optional): Name of the current testing session. The default name is a randomly generated 5-character long string (e.g., `hl43S`).

–language: When running Hephaestus, you should specify which language's the compiler you want to test. The available options are `kotlin`, `groovy`, and `java`. Hephaestus uses the selected language's compiler that is on the `$PATH`. If you want to test a specific compiler version, you should configure it as the current session's default compiler.

–seconds and –iterations: You should always specify either the `–seconds` or the `–iterations` option. The former specifies how much time (in seconds) Hephaestus should test a compiler seconds, whereas the second option specifies how many test cases Hephaestus should generate and run. For example, when providing the `–seconds 120` option, Hephaestus runs for 2 minutes.

–batch (Optional): When running Hephaestus, most of the testing time is spent on compiling the test programs. Instead of generating one program at a time, you can specify the number of programs you want to generate and then compiling all of them as a batch. The default option is 1.

–workers (Optional): When the `-batch` option is larger than one, you can specify the number of workers that generate, mutate, and compile programs in parallel. The default option is 1.
–transformation-types and –only-correctness-preserving-transformations: Hephaestus supports two transformations, those that produce well-typed programs and those that produce ill-typed programs. Currently, Hephaestus implements the Type Erasure Mutator (TEM) and the Type Overwriting Mutator (TOM). The former constructs well-typed programs, while the latter yields ill-typed.

By default, TOM is always running after generating a test program. To disable TOM, you should use the option `–only-correctness-preserving-transformations`. The `–transformation-types` option specifies the correctness-preserving transformations to run in a testing session. Currently, Hephaestus implements only one mutator that produces well-typed programs (i.e., TEM).
–transformations and –transformation-schedule: You should always specify one of those options. The `–transformations` specify the number of mutations that are applied per test program. If the given value is `0`, Hephaestus runs only the generator. Note that this option only specifies how many correctness-preserving mutations are applied. The `–transformation-schedule` expect a path for a file containing the schedule of transformations. This file should specify a mutator per line.

For example, when providing the option `–transformation-schedule transformations.txt`, Hephaestus performs the transformations declared in the file `transformations.txt`. The contents of this file are:

```
1 TypeErasure
```

–keep-all (Optional): By default, Hephaestus only stores programs that result in compiler bugs. When `–keep-all` is enabled, Hephaestus saves all the generated and mutated test programs regardless of whether they trigger compiler bugs or not.
–dry-run (Optional): When this option is used, Hephaestus produces and mutates test programs, but it does not invoke the compiler under test.
–log-file (Optional): By default, Hephaestus keeps logs in a file called `logs`, which resides in the current working directory. However, with `–log-file` option, you can specify another user-defined file path to save the logs.
–replay (Optional): Use a seed program written in Hephaestus's IR, instead of invoking Hephaestus's program generator. The input program should be pickled.
–debug (Debugging option): Print debug messages before every step (i.e., program generation, mutation, compilation). Use this option only when `–workers` option is set to 1 and `-batch` is set to 1.
–examine (Debugging option): Open a debugger session to inspect the IR of the generated program. This option can only be used with '–replay' option.
–print-stacktrace (Debugging option): Print stacktaces when encountering Hephaestusinternal errors.
–cast-numbers (Optional): This option is used to cast numeric constants to their actual type in Groovy programs. We use this option to avoid the re-occurrence of a specific Groovy bug. This option has an effect only when providing the `–language groovy` option.

–disable-use-site-variance (Optional): Generate programs that do not use use-site variance.
–disable-contravariance-use-site (Optional): Generate programs that do not use contravariance in use-site variance.
–disable-parameterized-functions (Optional): Generate programs that do not declare parameterized functions.
–disable-bounded-type-parameters (Optional): Generate programs that do declare type parameters with upper bounds.
–max-type-params (Optional): Specify the maximum number of type parameters for a parameterized class or a parameterized function. The default value is 3.
–max-depth (Optional): Generate program expressions up to a given depth. The default maximum depth is 6.

Appendix C

The Command-Line Interface of Cynthia

The command line interface of CYNTHIA can be found below:

```
cynthia@0fbedf262c3d:~$ cynthia --help
Cynthia version: 0.1
Usage: cynthia [test|generate|replay|run|inspect|clean] [options]

Cynthia: Data-Oriented Differential Testing of Object-Relational Mapping Systems

  --help Prints this usage text
  --version Prints the version of Cynthia
Command: test [options]

  -n, --queries <value> Number of queries to generate for each schema (default value:
      200)
  -s, --schemas <value> Number of schemas to generate (default value: 1)
  --timeout <value> Timeout for testing in seconds
  -o, --orms <value> ORMs to differentially test
                        (Available options: 'django', 'sqlalchemy', 'sequelize', '
                            peewee', 'activerecord', or 'pony')
  -d, --backends <value> Database backends to store data
                        (Available options: 'sqlite', 'postgres', 'mysql', 'mssql', '
                            cockroachdb', default value: sqlite)
  -u, --db-user <value> The username to log in the database
  -p, --db-pass <value> The password used to log in the database
  -S, --store-matches Save matches into the 'sessions' directory
  --combined Generate AQL queries consting of other simpler queries
  -r, --records <value> Number of records to generate for each table
  --min-depth <value> Minimum depth of the generated AQL queries
  --max-depth <value> Maximum depth of the generated AQL queries
  --no-well-typed Generate AQL queries that are type incorrect
  --solver Generate database records through a solver-based approach
  --solver-timeout <value>
                        Solver timeout for each query
  --random-seed <value> Make the testing procedure deterministic by giving a random seed
  --only-constrained-queries
                        Generate only constrained queries
Command: generate [options]
```

```
33
34    -n, --queries <value> Number of queries to generate for each schema (default value:
          200)
35    -s, --schemas <value> Number of schemas to generate (Default value: 1)
36    --combined Generate AQL queries consting of other simpler queries
37    -r, --records <value> Number of records to generate for each table
38    --min-depth <value> Minimum depth of the generated AQL queries
39    --max-depth <value> Maximum depth of the generated AQL queries
40    --no-well-typed Generate AQL queries that are type incorrect
41    --solver Generate database records through a solver-based approach
42    --solver-timeout <value>
43                            Solver timeout for each query
44    --random-seed <value> Make the testing procedure deterministic by giving a random seed
45    --only-constrained-queries
46                            Generate only constrained queries
47  Command: replay [options]
48
49    -c, --cynthia <value> The cynthia directory for replaying missmatches (default value:
          .cynthia)
50    -s, --schema <value> schema to replay
51    -a, --all Replay all queries.
52    -m, --mismatches <value>
53                            Replay queries for which ORM previously produced different
                                results
54    --generate-data Re-generate data while replaying testing sessions
55    -o, --orms <value> ORMs to differentially test
56                            (Available options: 'django', 'sqlalchemy', 'sequelize', '
                                peewee', 'activerecord', or 'pony')
57    -d, --backends <value> Database backends to store data
58                            (Available options: 'sqlite', 'postgres', 'mysql', 'mssql', '
                                cockroachdb', default value: sqlite)
59    -u, --db-user <value> The username to log in the database
60    -p, --db-pass <value> The password used to log in the database
61    -r, --records <value> Number of records to generate for each table
62    --solver Generate database records through a solver-based approach
63    --solver-timeout <value>
64                            Solver timeout for each query
65    --random-seed <value> Make the testing procedure deterministic by giving a random seed
66  Command: run [options]
67
68    -s, --sql <value> File with the sql script to generate and feed the database
69    -a, --aql <value> A file with an AQL query or a directory with many AQL queries
70    -o, --orms <value> ORMs to differentially test
71                            (Available options: 'django', 'sqlalchemy', 'sequelize', '
                                peewee', 'activerecord', or 'pony')
72    -d, --backends <value> Database backends to store data
73                            (Available options: 'sqlite', 'postgres', 'mysql', 'mssql', '
                                cockroachdb', default value: sqlite)
74    -u, --db-user <value> The username to log in the database
75    -p, --db-pass <value> The password used to log in the database
76    -S, --store-matches Save matches into the 'sessions' directory
```

```
77 Command: inspect [options]
78
79   -c, --cynthia <value> The cynthia directory for inspecting missmatches (default value:
         .cynthia)
80   -s, --schema <value> schema to inspect
81   -m, --mismatches <value>
82                           mismatches to inspect
83 Command: clean [options]
84
85   --only-workdir Clean only the working directory '.cynthia'
86   -u, --db-user <value> The username to log in the database
87   -p, --db-pass <value> The password used to log in the database
```

cynthia test

This is the main sub-command for testing ORMs. `cynthia test` expects at least two ORMs to test (by providing the `–orms` option), and some database systems (i.e., backends) specified by the `–backends` options. Note that when the option `–backends` is not given, the SQLite database system is used by default.

`cynthia test` first generates a number of relational database schemas. The number of the generated schemas is specified by the `–schemas` option. Every generated schema corresponds to a *testing session*. In every testing session, `cynthia test` generates a number of random AQL queries (the number of AQL queries is given by the `–queries` option), translates every AQL query into the corresponding executable ORM query, and finally runs every ORM query on the given backends. Note that for a given AQL query, CYNTHIA generates multiple ORM queries, one for every backend.

Example: In the following scenario, we differentially test the peewee and Django ORMs. The ORM queries are run on top of the SQLite and PostgreSQL databases, and we spawn 5 testing sessions (`–schemas 5`). In every testing session, we generate 100 AQL queries (`–queries 100`). To populate the underlying databases with data, we use the Z3 solver (`–solver`) to generate five records (`–records 5`) by solving the constraints of every generated AQL query. Finally, the option `–store-matches` is used to store the information coming from all AQL query runs inside the `.cynthia/sessions/` directory (see below). If this option is not provided, CYNTHIA stores only the AQL queries for which the ORMs under test produce different results.

```
cynthia test \
  --schemas 5 \
  --queries 100 \
  --orms django,peewee \
  --backends postgres \
  --solver \
  --records 5 \
  --random-seed 1 \
  --store-matches
```

The above command will produce an output similar to the following

```
1  Testing Serially 100% [======================== Passed: 95, Failed: 0, Unsp: 5,
       Timeouts: 1
2  Testing Cucumbers 100% [======================== Passed: 98, Failed: 0, Unsp: 2,
       Timeouts: 3
3  Testing Mumbles 100% [========================== Passed: 97, Failed: 0, Unsp: 3,
       Timeouts: 3
4  Testing Subhead 100% [========================== Passed: 98, Failed: 0, Unsp: 2,
       Timeouts: 2
5  Testing Wild 100% [============================ Passed: 96, Failed: 0, Unsp: 4,
       Timeouts: 0
```

Notably, CYNTHIA processes testing sessions in parallel by using Scala futures. CYNTHIA also dumps some statistics for every testing session. For example, consider line 2 of the previous listing. This line indicates that in the testing session named `Cucumbers`, CYNTHIA generated 100 AQL queries of which

- 98 / 100 queries passed (i.e., the ORMs under test produced exact results).
- 0 / 100 queries failed (i.e., the ORMs under test produced different results). Note that failed queries indicate a potential bug in at least one ORM implementation.
- 2 / 100 queries were unsupported meaning that the ORMs were unable to execute these queries, because these queries contained features that are not currently supported by the ORMs under test.
- 3 / 100 queries timed out, i.e., the SMT solver timed out and failed to generate records for populating the underlying databases.

Remark: When solver times out, CYNTHIA is still able to test the ORM implementations. However this time, the underlying database contains the data stemming from the previous AQL query, as the solver did not manage to generate records that satisfy the constraints of the current AQL query in a reasonable time limit. This is why the number of `Passed + Failed + Unsp + Timeouts > 100`

The .cynthia working directory: CYNTHIA produces a directory named `.cynthia` (inside the current working directory) where it stores important information about each run. The `.cynthia` directory has the following structure.

- `.cynthia/cynthia.log`: A file containing logs associated with the current run.
- `.cynthia/dbs/`: This is the directory where the SQLite database files of each testing session are stored.
- `.cynthia/schemas/`: This directory contains SQL scripts corresponding to the database schema of each testing session. Every SQL script contains all the necessary `CREATE TABLE` statements for creating the relational tables defined in each schema.
- `.cynthia/projects/`: A directory where CYNTHIA creates and executes the corresponding ORM queries.

- `.cynthia/sessions/`: This directory contains complete information about the ORM runs that take place in every testing session.

In particular, by inspecting the structure of the `.cynthia/sessions/` directory, we have the following:

- `.cynthia/sessions/<Session Name>/<Query ID>/data.sql`: This is the SQL script that populates the underlying database with data generated by the SMT solver. Note that these data are targeted, meaning that satisfy the constraints of the query identified by `<Query ID>`.
- `.cynthia/sessions/<Session Name>/<Query ID>/diff_test.out`: The output of differential testing. Either `"MATCH"`, `"MISMATCH"`, or `"UNSUPPORTED"`.
- `.cynthia/sessions/<Session Name>/<Query ID>/<orm>_<backend>.out`: The output produced by the query written by using the API of ORM named `<orm>`, and when this query is run on the backend `<backend>`.
- `.cynthia/sessions/<Session Name>/<Query ID>/query.aql`: The AQL query in human readable format.
- `.cynthia/sessions/<Session Name>/<Query ID>/query.aql.json`: The AQL query in JSON format. This is the format that CYNTHIA expects as input when replaying an AQL query.
- `.cynthia/sessions/<Session Name>/<Query ID>/<orm>/`: This directory contains all ORM-specific files for executing the ORM queries written in ORM named `<orm>`. For example, by executing

  ```
  python .cynthia/sessions/Cucumbers/22/django/driver_postgres.py
  ```

  You re-execute the Django query stemming from the AQL query with id 22. This query is run on the PostgresSQL database.

cynthia replay

This sub-command replays the execution of a particular testing session based on information extracted from the `.cynthia` directory. This command is particularly useful when we want to run the same queries with different settings (i.e., running the same AQL queries on different database systems).

Example1: Replay all testing sessions previously created by `cynthia test`.

```
cynthia replay \
  --orms django,peewee \
  --backends postgres \
  --all
```

Example2: Replay the execution of a specific testing session, and run ORM queries on MySQL instead of PostgreSQL.

```
cynthia replay \
  --schema Cucumbers \
  --orms django,peewee \
  --backends mysql \
  --all
```

cynthia run

The sub-command `cynthia run` tests the given ORMs against certain AQL queries (provided by the user) based on a given database schema, which is also provided by the user (not generated by CYNTHIA).

Example: The command below tests Django and peewee against the AQL query located in the directory `cynthia_src/examples/books/10.aql.json` and the database schema defined by the script `cynthia_src/examples/book.sql`. The underlying database system is SQLite.

```
cynthia run \
  --sql cynthia_src/examples/books.sql \
  --aql cynthia_src/examples/books/10.aql.json \
  --orms django,peewee \
  --store-matches
```

cynthia generate

`cynthia generate` generates a number of relational database schemas, and for each database schema, it produces a number of AQL queries and data. This command does *not* test ORMs, i.e., CYNTHIA does not translate the generated AQL queries into concrete ORM queries. Every generated query is stored inside the `.cynthia/sessions/` directory. In order to test ORMs, the queries generated by `cynthia generate` can be later executed using the `cynthia replay` command as documented above.

Example: Generate 5 random database schema. For every schema, generate 100 AQL queries. The generated data are generated by a solver-based approach, and each table contains 5 records.

```
cynthia generate \
 --schemas 5 \
 --queries 100 \
 --records 5 \
 --solver
```

cynthia inspect

This helper sub-command inspects the results, and reports the queries for which the ORMs under test produced different results. To do so, `cynthia inspect` extracts information from the `.cynthia` directory.

Example: Inspect the testing session named ‘Cucumbers‘.

```
cynthia inspect --schema Cucumbers
```

Appendix D

The Command-Line Interface of FSMoVe

The command-line interface of FSMovecan be seen below.

```
fsmove help
Detecting Dependency Bugs in File System Resources

  fsmove SUBCOMMAND

=== subcommands ===

  gradle-build This is the sub-command for analyzing and detecting dependency bugs in
               Gradle scripts
  make-build This is the sub-command for analyzing and detecting dependency bugs in
               Make scripts
  puppet This is the sub-command for analyzing and detecting dependency bugs in
               Puppet scripts
  version print version information
  help explain a given subcommand (perhaps recursively)
```

The command-line interface of the `fsmove gradle-build` command is:

```
fsmove gradle-build -help
This is the sub-command for analyzing and detecting dependency bugs in Gradle scripts

  fsmove gradle-build

=== flags ===

  -build-dir Build directory
  -mode Analysis mode; either online or offline
  [-build-task Build] task to execute
  [-dump-tool-out File] to store output from Gradle execution (for debugging
                        only)
  [-graph-file File] to store the task graph inferred by fsmove.
  [-graph-format Format] for storing the task graph of the fsmove program.
  [-print-stats] Print stats about execution and analysis
  [-trace-file Path] to trace file produced by the strace tool.
  [-help] print this help text and exit
                        (alias: -?)
```

The command-line interface of the `fsmove make-build` command is:

```
This is the sub-command for analyzing and detecting dependency bugs in Make scripts

  fsmove make-build

=== flags ===

  -build-dir Build directory
  -mode Analysis mode; either online or offline
  [-build-db Path] to Make database
  [-dump-tool-out File] to store output from Make execution (for debugging
                        only)
  [-graph-file File] to store the task graph inferred by fsmove.
  [-graph-format Format] for storing the task graph of the fsmove program.
  [-print-stats] Print stats about execution and analysis
  [-trace-file Path] to trace file produced by the strace tool
  [-help] print this help text and exit
                        (alias: -?)
```

Finally, the command-line interface of the `fsmove puppet` command is:

```
> fsmove puppet
This is the sub-command for analyzing and detecting dependency bugs in Puppet scripts

  fsmove

=== flags ===

  -catalog Path to the compiled catalog of Puppet manifest.
  -mode Analysis mode; either online or offline
  [-dump-puppet-out File] to store output from Puppet execution (for debugging
                          only)
  [-graph-file File] to store the task graph inferred by fsmove.
  [-graph-format Format] for storing the task graph of the fsmove program.
  [-manifest Path] to the entrypoint manifest that we need to apply.
                          (Avaiable only when mode is 'online')
  [-modulepath Path] to the directory of the Puppet modules. (Available
                          only when mode is 'online')
  [-package-notify] Consider missing notifiers from packages to services
  [-print-stats] Print stats about execution and analysis
  [-trace-file Path] to the trace file produced by the strace tool
  [-version] print the version of this build and exit
  [-help] print this help text and exit
                          (alias: -?)
```

Appendix E

Links to Bug Reports and Pull Requests

Compiler Typing Bugs

Groovy

1. https://issues.apache.org/jira/browse/GROOVY-10011
2. https://issues.apache.org/jira/browse/GROOVY-10033
3. https://issues.apache.org/jira/browse/GROOVY-10079
4. https://issues.apache.org/jira/browse/GROOVY-10080
5. https://issues.apache.org/jira/browse/GROOVY-10081
6. https://issues.apache.org/jira/browse/GROOVY-10082
7. https://issues.apache.org/jira/browse/GROOVY-10086
8. https://issues.apache.org/jira/browse/GROOVY-10087
9. https://issues.apache.org/jira/browse/GROOVY-10091
10. https://issues.apache.org/jira/browse/GROOVY-10092
11. https://issues.apache.org/jira/browse/GROOVY-10094
12. https://issues.apache.org/jira/browse/GROOVY-10095
13. https://issues.apache.org/jira/browse/GROOVY-10096
14. https://issues.apache.org/jira/browse/GROOVY-10098
15. https://issues.apache.org/jira/browse/GROOVY-10100
16. https://issues.apache.org/jira/browse/GROOVY-10107
17. https://issues.apache.org/jira/browse/GROOVY-10111
18. https://issues.apache.org/jira/browse/GROOVY-10113
19. https://issues.apache.org/jira/browse/GROOVY-10114
20. https://issues.apache.org/jira/browse/GROOVY-10115
21. https://issues.apache.org/jira/browse/GROOVY-10116
22. https://issues.apache.org/jira/browse/GROOVY-10125
23. https://issues.apache.org/jira/browse/GROOVY-10127
24. https://issues.apache.org/jira/browse/GROOVY-10128
25. https://issues.apache.org/jira/browse/GROOVY-10129
26. https://issues.apache.org/jira/browse/GROOVY-10130
27. https://issues.apache.org/jira/browse/GROOVY-10153
28. https://issues.apache.org/jira/browse/GROOVY-10158

29. https://issues.apache.org/jira/browse/GROOVY-10220
30. https://issues.apache.org/jira/browse/GROOVY-10221
31. https://issues.apache.org/jira/browse/GROOVY-10222
32. https://issues.apache.org/jira/browse/GROOVY-10225
33. https://issues.apache.org/jira/browse/GROOVY-10226
34. https://issues.apache.org/jira/browse/GROOVY-10227
35. https://issues.apache.org/jira/browse/GROOVY-10228
36. https://issues.apache.org/jira/browse/GROOVY-10230
37. https://issues.apache.org/jira/browse/GROOVY-10251
38. https://issues.apache.org/jira/browse/GROOVY-10254
39. https://issues.apache.org/jira/browse/GROOVY-10264
40. https://issues.apache.org/jira/browse/GROOVY-10265
41. https://issues.apache.org/jira/browse/GROOVY-10266
42. https://issues.apache.org/jira/browse/GROOVY-10267
43. https://issues.apache.org/jira/browse/GROOVY-10268
44. https://issues.apache.org/jira/browse/GROOVY-10269
45. https://issues.apache.org/jira/browse/GROOVY-10270
46. https://issues.apache.org/jira/browse/GROOVY-10271
47. https://issues.apache.org/jira/browse/GROOVY-10272
48. https://issues.apache.org/jira/browse/GROOVY-10277
49. https://issues.apache.org/jira/browse/GROOVY-10280
50. https://issues.apache.org/jira/browse/GROOVY-10283
51. https://issues.apache.org/jira/browse/GROOVY-10291
52. https://issues.apache.org/jira/browse/GROOVY-10294
53. https://issues.apache.org/jira/browse/GROOVY-10306
54. https://issues.apache.org/jira/browse/GROOVY-10308
55. https://issues.apache.org/jira/browse/GROOVY-10309
56. https://issues.apache.org/jira/browse/GROOVY-10310
57. https://issues.apache.org/jira/browse/GROOVY-10315
58. https://issues.apache.org/jira/browse/GROOVY-10316
59. https://issues.apache.org/jira/browse/GROOVY-10317
60. https://issues.apache.org/jira/browse/GROOVY-10322
61. https://issues.apache.org/jira/browse/GROOVY-10323
62. https://issues.apache.org/jira/browse/GROOVY-10324
63. https://issues.apache.org/jira/browse/GROOVY-10327
64. https://issues.apache.org/jira/browse/GROOVY-10330
65. https://issues.apache.org/jira/browse/GROOVY-10336
66. https://issues.apache.org/jira/browse/GROOVY-10337
67. https://issues.apache.org/jira/browse/GROOVY-10339
68. https://issues.apache.org/jira/browse/GROOVY-10342
69. https://issues.apache.org/jira/browse/GROOVY-10343
70. https://issues.apache.org/jira/browse/GROOVY-10344
71. https://issues.apache.org/jira/browse/GROOVY-10351
72. https://issues.apache.org/jira/browse/GROOVY-10356

73. https://issues.apache.org/jira/browse/GROOVY-10357
74. https://issues.apache.org/jira/browse/GROOVY-10358
75. https://issues.apache.org/jira/browse/GROOVY-10359
76. https://issues.apache.org/jira/browse/GROOVY-10360
77. https://issues.apache.org/jira/browse/GROOVY-10362
78. https://issues.apache.org/jira/browse/GROOVY-10363
79. https://issues.apache.org/jira/browse/GROOVY-10364
80. https://issues.apache.org/jira/browse/GROOVY-10365
81. https://issues.apache.org/jira/browse/GROOVY-10367
82. https://issues.apache.org/jira/browse/GROOVY-10368
83. https://issues.apache.org/jira/browse/GROOVY-10369
84. https://issues.apache.org/jira/browse/GROOVY-10370
85. https://issues.apache.org/jira/browse/GROOVY-10371
86. https://issues.apache.org/jira/browse/GROOVY-10373
87. https://issues.apache.org/jira/browse/GROOVY-10480
88. https://issues.apache.org/jira/browse/GROOVY-10482
89. https://issues.apache.org/jira/browse/GROOVY-10499
90. https://issues.apache.org/jira/browse/GROOVY-9907
91. https://issues.apache.org/jira/browse/GROOVY-9931
92. https://issues.apache.org/jira/browse/GROOVY-9934
93. https://issues.apache.org/jira/browse/GROOVY-9935
94. https://issues.apache.org/jira/browse/GROOVY-9937
95. https://issues.apache.org/jira/browse/GROOVY-9945
96. https://issues.apache.org/jira/browse/GROOVY-9947
97. https://issues.apache.org/jira/browse/GROOVY-9948
98. https://issues.apache.org/jira/browse/GROOVY-9952
99. https://issues.apache.org/jira/browse/GROOVY-9953
100. https://issues.apache.org/jira/browse/GROOVY-9956
101. https://issues.apache.org/jira/browse/GROOVY-9960
102. https://issues.apache.org/jira/browse/GROOVY-9963
103. https://issues.apache.org/jira/browse/GROOVY-9967
104. https://issues.apache.org/jira/browse/GROOVY-9970
105. https://issues.apache.org/jira/browse/GROOVY-9972
106. https://issues.apache.org/jira/browse/GROOVY-9979
107. https://issues.apache.org/jira/browse/GROOVY-9983
108. https://issues.apache.org/jira/browse/GROOVY-9984
109. https://issues.apache.org/jira/browse/GROOVY-9985
110. https://issues.apache.org/jira/browse/GROOVY-9986
111. https://issues.apache.org/jira/browse/GROOVY-9994
112. https://issues.apache.org/jira/browse/GROOVY-9995

Kotlin

1. https://youtrack.jetbrains.com/issue/KT-43806
2. https://youtrack.jetbrains.com/issue/KT-43846
3. https://youtrack.jetbrains.com/issue/KT-44082
4. https://youtrack.jetbrains.com/issue/KT-44551
5. https://youtrack.jetbrains.com/issue/KT-44595
6. https://youtrack.jetbrains.com/issue/KT-44627
7. https://youtrack.jetbrains.com/issue/KT-44651
8. https://youtrack.jetbrains.com/issue/KT-44742
9. https://youtrack.jetbrains.com/issue/KT-45118
10. https://youtrack.jetbrains.com/issue/KT-45339
11. https://youtrack.jetbrains.com/issue/KT-46662
12. https://youtrack.jetbrains.com/issue/KT-46684
13. https://youtrack.jetbrains.com/issue/KT-46864
14. https://youtrack.jetbrains.com/issue/KT-47073
15. https://youtrack.jetbrains.com/issue/KT-47117
16. https://youtrack.jetbrains.com/issue/KT-47184
17. https://youtrack.jetbrains.com/issue/KT-47231
18. https://youtrack.jetbrains.com/issue/KT-47458
19. https://youtrack.jetbrains.com/issue/KT-47508
20. https://youtrack.jetbrains.com/issue/KT-47530
21. https://youtrack.jetbrains.com/issue/KT-48671
22. https://youtrack.jetbrains.com/issue/KT-48764
23. https://youtrack.jetbrains.com/issue/KT-48765
24. https://youtrack.jetbrains.com/issue/KT-48766
25. https://youtrack.jetbrains.com/issue/KT-48838
26. https://youtrack.jetbrains.com/issue/KT-48840
27. https://youtrack.jetbrains.com/issue/KT-48858
28. https://youtrack.jetbrains.com/issue/KT-48958
29. https://youtrack.jetbrains.com/issue/KT-49024
30. https://youtrack.jetbrains.com/issue/KT-49092
31. https://youtrack.jetbrains.com/issue/KT-49101
32. https://youtrack.jetbrains.com/issue/KT-49583

Java

1. https://bugs.openjdk.java.net/browse/JDK-8267220
2. https://bugs.openjdk.java.net/browse/JDK-8267610
3. https://bugs.openjdk.java.net/browse/JDK-8268159
4. https://bugs.openjdk.java.net/browse/JDK-8269348
5. https://bugs.openjdk.java.net/browse/JDK-8269386
6. https://bugs.openjdk.java.net/browse/JDK-8269586
7. https://bugs.openjdk.java.net/browse/JDK-8269737
8. https://bugs.openjdk.java.net/browse/JDK-8269738

9. https://bugs.openjdk.java.net/browse/JDK-8272077
10. https://bugs.openjdk.java.net/browse/JDK-8274183
11. https://bugs.openjdk.java.net/browse/JDK-8276081

ORM Bugs

1. https://code.djangoproject.com/ticket/31445
2. https://github.com/sqlalchemy/sqlalchemy/issues/5344
3. https://github.com/sequelize/sequelize/issues/12073
4. https://github.com/sequelize/sequelize/issues/12099
5. https://github.com/sequelize/sequelize/issues/12315
6. https://code.djangoproject.com/ticket/31651
7. https://code.djangoproject.com/ticket/31659
8. https://github.com/coleifer/peewee/issues/2200
9. https://github.com/sqlalchemy/sqlalchemy/issues/5395
10. https://code.djangoproject.com/ticket/31699
11. https://code.djangoproject.com/ticket/31679
12. https://github.com/sqlalchemy/sqlalchemy/issues/5443
13. https://github.com/coleifer/peewee/issues/2220
14. https://code.djangoproject.com/ticket/31773
15. https://github.com/rails/rails/issues/39857
16. https://github.com/sequelize/sequelize/issues/12519
17. https://github.com/sqlalchemy/sqlalchemy/issues/5469
18. https://github.com/sqlalchemy/sqlalchemy/issues/5470
19. https://github.com/coleifer/peewee/issues/2233
20. https://code.djangoproject.com/ticket/31880
21. https://github.com/sequelize/sequelize/pull/12622
22. https://github.com/ESSolutions/django-mssql-backend/issues/62
23. https://code.djangoproject.com/ticket/31916
24. https://code.djangoproject.com/ticket/31919
25. https://github.com/sqlalchemy/sqlalchemy/pull/5539
26. https://github.com/sqlalchemy/sqlalchemy/issues/5511
27. https://github.com/coleifer/peewee/issues/2224

Dependency Bugs in Build and Configuration Management Scripts

1. https://github.com/kncept/junit-reporter/pull/14
2. https://github.com/zlatinb/muwire/pull/24
3. https://github.com/real-logic/aeron/pull/762
4. https://github.com/Netflix/conductor/pull/1385
5. https://github.com/xtext/xtext-gradle-plugin/pull/163
6. https://github.com/rundeck/rundeck/pull/5707

7. https://github.com/EvidentSolutions/apina/pull/56
8. https://github.com/ben-manes/caffeine/pull/383
9. https://github.com/Polidea/RxAndroidBle/pull/647
10. https://github.com/jMonkeyEngine/jmonkeyengine/pull/1209
11. https://github.com/alibaba/tsar/pull/102
12. https://github.com/orangeduck/Cello/pull/127
13. https://github.com/nicolasff/webdis/pull/168
14. https://github.com/centaurean/density/pull/88
15. https://github.com/notdodo/VRP-tabu/pull/1
16. https://github.com/okbob/pspg/pull/122
17. https://github.com/janet-lang/janet/pull/235
18. https://github.com/rickyrockrat/parcellite/pull/41
19. https://salsa.debian.org/debian/hdparm/-/merge_requests/3
20. https://github.com/nelhage/reptyr/pull/115
21. https://github.com/osqzss/gps-sdr-sim/pull/226
22. https://github.com/dspinellis/cscout/pull/55
23. https://github.com/haad/proxychains/pull/95
24. https://github.com/tomasbjerre/gradle-scripts/pull/2
25. https://github.com/rdk/p2rank/pull/9
26. https://github.com/diffplug/goomph/pull/110
27. https://github.com/adamldavis/groocss/pull/16
28. https://github.com/phatblat/ShellExec/pull/62
29. https://salsa.debian.org/installer-team/anna/-/merge_requests/1
30. https://github.com/fujaba/fulibGradle/pull/6/
31. https://github.com/seqeralabs/nf-tower/pull/185
32. https://github.com/xenit-eu/alfresco-gradle-sdk/pull/34
33. https://github.com/wpilibsuite/GradleRIO/pull/361
34. https://github.com/47degrees/helios/pull/131
35. https://github.com/holgerbrandl/kscript/pull/252
36. https://github.com/kt3k/coveralls-gradle-plugin/pull/101
37. https://github.com/radarsh/gradle-test-logger-plugin/pull/140
38. https://salsa.debian.org/debian/joystick/-/merge_requests/2
39. https://github.com/dspinellis/cqmetrics/pull/13|14
40. https://github.com/jenkinsci/JenkinsPipelineUnit/pull/167
41. https://github.com/gigaSproule/swagger-gradle-plugin/pull/161
42. https://github.com/int128/gradle-swagger-generator-plugin/pull/186
43. https://github.com/soramitsu/gradle-sora-plugin/pull/16
44. https://github.com/ssloy/tinyrenderer/pull/42
45. https://github.com/cs50/libcs50/pull/192
46. https://github.com/jhawthorn/fzy/pull/134
47. https://github.com/LuaJIT/LuaJIT/pull/546

Bibliography

2019. CVE-2019-7164. https://nvd.nist.gov/vuln/detail/CVE-2019-7164. [Online; accessed 29-July-2020].

2020. CVE-2020-9402. https://nvd.nist.gov/vuln/detail/CVE-2020-9402. [Online; accessed 29-July-2020].

S. Abdul Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 238–247.

Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL Query Generation for Systematic Testing of Database Engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 329–332. https://doi.org/10.1145/1858996.1859063

Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. 2016. An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 960–1001.

B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. 2007. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*. 114–123. https://doi.org/10.1109/ICSM.2007.4362624

Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Build Code Errors in Makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. ACM, New York, NY, USA, 600–601. https://doi.org/10.1145/2591062.2591135

Alert Logic Inc. 2019. Alert Logic Agent Puppet Module. https://forge.puppet.com/alertlogic/agents.

S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018), 1–1.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types*. Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Glenn Ammons. 2006. Grexmk: Speeding up Scripted Builds. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis* (Shanghai, China) *(WODA '06)*. Association for Computing Machinery, New York, NY, USA, 81–87. https://doi.org/10.1145/1138912.1138928

Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5, Article 66 (sep 2017), 36 pages. https://doi.org/10.1145/3106739

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 19, 17 pages.

Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020a. Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).

Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 748–758. https://doi.org/10.1109/ICSE.2019.00083

Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020b. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 387–397. https://doi.org/10.1109/ICST46399.2020.00046

Bill Allombert Avery Pennarun and Petter Reinholdtsen. 2020. Debian Popularity Contest. https://popcon.debian.org/.

Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 507–517. https://doi.org/10.1145/2635868.2635916

Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor Concurrency Bugs: A Comprehensive Study on Symptoms, Root Causes, API Usages, and Differences. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 214 (Nov. 2020), 32 pages. https://doi.org/10.1145/3428282

Sebastian Bank. 2016. negated EXISTS result type not bool with SQLite dialect. https://github.com/sqlalchemy/sqlalchemy/issues/3682. [Online; accessed 29-July-2020].

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1243–1251.

Christian Bauer, Gavin King, and Gary Gregory. 2015. *Java Persistence with Hibernate* (2nd ed.). Manning Publications Co., USA.

Michael Bayer. 2020. SQLAlchemy - The Database Toolkit for Python. https://www.sqlalchemy.org/. [Online; accessed 29-July-2020].

Bazel. 2020. Build and test software of any size, quickly and reliably. https://bazel.build.

Cor-Paul Bezemer, Shane Mcintosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3117–3148. https://doi.org/10.1007/s10664-017-9510-8

Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 122–131.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/286936.286957

Brian Goetz. 2020. State of Valhalla. https://cr.openjdk.java.net/~briangoetz/valhalla/sov/01-background.html. Online accessed; 05-03-2021.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, USA, 265–275.

Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2151024.2151043

N. Bruno, S. Chaudhuri, and D. Thomas. 2006. Generating Queries with Cardinality Constraints for DBMS Testing. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1721–1725.

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (feb 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 459–465.

John Calcote. 2020. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press.

Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/2950290.2950358

Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection across JVM Boundaries. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 809–820. https://doi.org/10.1145/3106237.3106297

Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. https://doi.org/10.1145/3519939.3523720

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of*

the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. https://doi.org/10.1145/3519939.3523427

Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to Prioritize Test Programs for Compiler Testing. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 700–711. https://doi.org/10.1109/ICSE.2017.70

Junjie Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. 2016. Test Case Prioritization for Compilers: A Text-Vector Based Approach. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 266–277. https://doi.org/10.1109/ICST.2016.19

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. https://doi.org/10.1145/3363562

T. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. 2016. An Empirical Study on the Practice of Maintaining Object-Relational Mapping Code in Java Systems. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 165–176.

Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting Performance Anti-Patterns for Applications Developed Using Object-Relational Mapping. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012. https://doi.org/10.1145/2568225.2568259

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01* (1998).

W. Chen, G. Wu, and J. Wei. 2018. An Approach to Identifying Error Patterns for Infrastructure as Code. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 124–129.

Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. *SIGPLAN Not.* 51, 6 (June 2016), 85–99. https://doi.org/10.1145/2980983.2908095

Yuting Chen and Zhendong Su. 2015a. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 793–804. https://doi.org/10.1145/2786805.2786835

Yuting Chen and Zhendong Su. 2015b. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 793–804. https://doi.org/10.1145/2786805.2786835

Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49, 4 (1995), 327–335.

Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/3377811.3380381

Maria Christakis, K. Rustan Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*. Springer, 643–657. https://doi.org/10.1007/978-3-319-06410-9_43

Derrick Coetzee, Anand Bhaskar, and George Necula. 2011. apmake: A reliable parallel build manager. In *2011 USENIX Annual Technical Conference (USENIX)*.

Keith Cooper and Linda Torczon. 2012. *Engineering a Compiler (Second Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

David Coppit and Jiexin Lian. 2005. Yagg: An Easy-to-Use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 356–359. https://doi.org/10.1145/1101908.1101969

Charlie Curtsinger and Daniel W. Barowy. 2022. Riker: Always-Correct and Fast Incremental Builds from Simple Specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 885–898. https://www.usenix.org/conference/atc22/presentation/curtsinger

Zahra Davar and Handoko. 2014. Refactoring Object-Relational Database Applications by Applying Transformation Rules to Develop Better Performance. In *Proceedings of the 16th International Conference on Information Integration and Web-Based Applications & Services* (Hanoi, Viet Nam) *(iiWAS '14)*. Association for Computing Machinery, New York, NY, USA, 283–288. https://doi.org/10.1145/2684200.2684304

James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 145–160. https://doi.org/10.1145/3064176.3064188

DB-Engines. 2022. Database engines ranking. https://db-engines.com/en/ranking. [Online; accessed 31-May-2022].

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Debian. 2020a. sbuild. https://wiki.debian.org/sbuild.

Debian. 2020b. UltimateDebianDatabase. https://wiki.debian.org/UltimateDebianDatabase/.

Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. 2010. A Survey of System Configuration Tools. In *Proceedings of the 24th International Conference on Large Installation System Administration* (San Jose, CA) *(LISA'10)*. USENIX Association, Berkeley, CA, USA, 1–8.

Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2187–2200. https://doi.org/10.1145/3133956.3134059

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 725–730. https://doi.org/10.1145/2642937.2642963

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 482–493. https://doi.org/10.1109/ASE.2015.65

Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-World Numerical Bug Characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 509–519. https://doi.org/10.1109/ASE.2017.8115662

Django Software Foundation. 2020a. Django Issues. https://code.djangoproject.com/query. [Online; accessed 29-July-2020].

Django Software Foundation. 2020b. Integrating Django with a legacy database. https://docs.djangoproject.com/en/3.0/howto/legacy-databases/. [Online; accessed 29-July-2020].

Django Software Foundation. 2020c. The Web Framework for Perfectionists with Deadlines. https://www.djangoproject.com/. [Online; accessed 29-July-2020].

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic Testing for (Graphics) Compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing* (Austin, Texas) *(MET '16)*. Association for Computing Machinery, New York, NY, USA, 44–47. https://doi.org/10.1145/2896971.2896978

Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3453483.3454092

Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 574–586. https://doi.org/10.1145/3236024.3236057

EclEmma. 2021. EclEmma Jacoco. https://www.eclemma.org/jacoco/. Online accessed; 26-10-2021.

Ibrahim K El-Far and James A Whittaker. 2002. Model-based software testing. *Encyclopedia of Software Engineering* (2002).

Mike Eltsufin. 2019. Bringing Hibernate ORM to Cloud Spanner for database adoption. https://cloud.google.com/blog/products/databases/bringing-hibernate-orm-cloud-spanner-database-adoption.

Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science* 217, C (2008), 5 – 21. https://doi.org/10.1016/j.entcs.2008.06.039

Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. *SIGPLAN Not.* 50, 10 (Oct. 2015), 89–106. https://doi.org/10.1145/2858965.2814316

Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1219–1223. https://doi.org/10.1145/3324884.3418933

Stuart I. Feldman. 1979. Make—A Program for Maintaining Computer Programs. *Software: Practice & Experience* 9, 4 (1979), 255–265.

Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

Julia Gabet and Nobuko Yoshida. 2020. Static Race Detection and Mutex Safety and Liveness for Go Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.4

Gavin Bierman. 2017. Pattern Matching for instanceof. https://openjdk.java.net/jeps/305. Online accessed; 05-03-2021.

Github. 2014. DNS Outage Post Mortem - The GitHub Blog. https://github.blog/2014-01-18-dns-outage-post-mortem/. [Online; accessed 28-January-2019].

Github Inc. 2021. The state of the Octoverse. https://octoverse.github.com/. Online accessed; 05-03-2021.

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/2771783.2771784

Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014a. Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 599–616. https://doi.org/10.1145/2660193.2660239

Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. 2014b. Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 599–616.

GNU Make. 2020a. Generating Prerequisites Automatically. https://www.gnu.org/software/make/manual/html_node/Automatic-Prerequisites.html.

GNU Make. 2020b. Handling Tools that Produce Many Outputs. https://www.gnu.org/software/automake/manual/html_node/Multiple-Outputs.html.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. *SIGPLAN Not.* 43, 6 (jun 2008), 206–215. https://doi.org/10.1145/1379022.1375607

Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential Regression Testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 312–323. https://doi.org/10.1145/3395363.3397374

Google. 2022. Honggfuzz: Security oriented software fuzzer. https://honggfuzz.dev/.

Google developers. 2021. Kotlin development stories. https://developer.android.com/kotlin/stories?linkId=94116374. Online accessed; 07-11-2021.

Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. *Input Algebras*. IEEE Press, 699–710. https://doi.org/10.1109/ICSE43902.2021.00070

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification: Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

Gradle Inc. 2020a. Build Cache. https://docs.gradle.org/current/userguide/build_cache.html.

Gradle Inc. 2020b. Developing Custom Gradle Plugins. https://docs.gradle.org/current/userguide/custom_plugins.html.

Gradle Inc. 2020c. Gradle - Plugins. https://plugins.gradle.org/.

Gradle Inc. 2020d. Gradle vs Maven: Performance Comparison. https://gradle.org/gradle-vs-maven-performance/.

Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 73–85. https://doi.org/10.1145/3009837.3009871

Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making No-Fuss Compiler Fuzzing Effective. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 194–204. https://doi.org/10.1145/3497776.3517765

Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/2338965.2336763

Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2020. Futures and Promises. https://docs.scala-lang.org/overviews/core/futures.html.

Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 144–156.

Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting Reliable Convergence for Configuration Management Scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 328–343.

Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Markham, Ontario, Canada) *(ESEM '17)*. IEEE Press, Piscataway, NJ, USA, 38–47. https://doi.org/10.1109/ESEM.2017.11

Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-driven Repair of Build Scripts. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1078–1089. https://doi.org/10.1145/3180155.3180181

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

J. Humble, C. Read, and D. North. 2006. The deployment production line. In *AGILE 2006 (AGILE'06)*. 6 pp.–118.

Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Middleware 2013*, David Eyers and Karsten Schwan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–388.

Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 556–567. https://doi.org/10.1145/3533767.3534382

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70. https://doi.org/10.14778/3357377.3357382

John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7, 3 (sep 1977), 305–317. https://doi.org/10.1007/BF00290339

Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 590–600.

Rashi Karanpuria and Aanand Shekhar Roy. 2018. *Kotlin Programming Cookbook: Explore more than 100 recipes that show how to build robust mobile and web applications with Kotlin, Spring Boot, and Android.* Packt Publishing Ltd.

Ke Mao. 2018. Sapienz: Intelligent automated software testing at scale. https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/. [Online; accessed 06-July-2022].

Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and Efficient API Misuse Detection via Exception Propagation and Search-based Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 192–203.

Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. *Lightweight and Modular Resource Leak Verification.* Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/3468264.3468576

Khronos Group. 2022. Khronos Vulkan, OpenGL, and OpenGL ES Conformance Tests. https://github.com/KhronosGroup/VK-GL-CTS. [Online; accessed 06-July-2022].

Paul King. 2022. Groovy release train: 4.0.4, 3.0.12, 2.5.18. https://blogs.apache.org/groovy/entry/groovy-release-train-4-0. [Online; accessed 05-September-2022].

Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/3293882.3330553

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable Incremental Building with Dynamic Task Dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 76–86. https://doi.org/10.1145/3238147.3238196

Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala Implicits Are Everywhere: A Large-Scale Study of the Use of Scala Implicits in the Wild. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 163 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360589

Andrei Lascu, Alastair F. Donaldson, Tobias Grosser, and Torsten Hoefler. 2022. Metamorphic Fuzzing of C++ Libraries. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'22)* (Online).

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015a. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

Vu Le, Chengnian Sun, and Zhendong Su. 2015b. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276495

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

Daniel Lehmann and Michael Pradel. 2018. Feedback-Directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 610–620. https://doi.org/10.1145/3236024.3236037

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International*

Symposium on Software Testing and Analysis (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/1111037.1111042

Nancy G Leveson and Clark S Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (1993), 18–41.

Nándor Licker and Andrew Rice. 2019. Detecting Incorrect Build Rules. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1234–1244. https://doi.org/10.1109/ICSE.2019.00125

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015a. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015b. Many-Core Compiler Fuzzing. *SIGPLAN Not.* 50, 6 (June 2015), 65–76. https://doi.org/10.1145/2813885.2737986

Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/3183440.3183495

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A Type-and-Effect System for Object Initialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 175 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428243

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Baltimore, MD) *(SSYM'05)*. USENIX Association, USA, 18.

LLVM Project. 2021. Fuzzing LLVM libraries and tools. https://llvm.org/docs/FuzzingLLVM.html. Online accessed; 07-11-2021.

LLVM Project. 2022. libFuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

James Loope. 2011. *Managing Infrastructure with Puppet: Configuration Management at Scale.* O'Reilly Media.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

M. Zalewski. 2013. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Online accessed; 05-08-2021.

Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 – December 2, 2020, Proceedings* (Fukuoka, Japan). Springer-Verlag, Berlin, Heidelberg, 125–144. https://doi.org/10.1007/978-3-030-64437-6_7

David Maier. 1990. *Representing Database Programs as Objects.* Association for Computing Machinery, New York, NY, USA, 377–386. https://doi.org/10.1145/101620.101642

Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360581

Ken Martin and Bill Hoffman. 2010. *Mastering CMake: a cross-platform build system.* Kitware.

Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 158 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360584

Bruno Gois Mateus and Matias Martinez. 2020. On the Adoption, Usage and Evolution of Kotlin Features in Android Development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/3382494.3410676

P.M. Maurer. 1990. Generating test data with enhanced context-free grammars. *IEEE Software* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

Richard McDougall, Jim Mauro, and Brendan Gregg. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris.* Prentice Hall PTR, Upper Saddle River.

Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An Empirical Study of Build Maintenance Effort. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/1985793.1985813

Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering* 20, 6 (01 Dec 2015), 1587–1633. https://doi.org/10.1007/s10664-014-9324-x

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating Targeted Queries for Database Testing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 499–510. https://doi.org/10.1145/1376616.1376668

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/1449764.1449798

J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. 2012. Searching for Build Debt: Experiences Managing Technical Debt at Google. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Zurich, Switzerland) *(MTD '12)*. IEEE Press, 1–6.

Kief Morris. 2016. *Infrastructure As Code: Managing Servers in the Cloud* (1st ed.). O'Reilly Media, Inc.

Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races That Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 381–392. https://doi.org/10.1145/2786805.2786820

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. An overview of the Scala programming language. (2004).

OpenJDK. 2021. OpenJDK Jir deployment. https://bugs.openjdk.java.net. Online accessed; 26-10-2021.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. *Semantic Fuzzing with Zest.* Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

J. Palsberg and C.B. Jay. 1998. The essence of the Visitor pattern. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*. 9–15. https://doi.org/10.1109/CMPSAC.1998.716629

Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 152 (Oct. 2021), 19 pages. https://doi.org/10.1145/3485529

S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

Kevin Pelgrims. 2015. *Gradle for Android.* Packt Publishing Ltd.

T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632.

Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic Test Case Reduction for OpenCL. In *4th International Workshop on OpenCL (IWOCL'16)* (Vienna, Austria). https://doi.org/10.1145/2909437.2909439

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/143165.143228

Shawn Plummer and David Warden. 2016. Puppet: Introduction, Implementation, & the Inevitable Refactoring. In *Proceedings of the 2016 ACM on SIGUCCS Annual Conference* (Denver, Colorado, USA) *(SIGUCCS '16)*. ACM, New York, NY, USA, 131–134.

Puppet. 2019a. Provides an interface for managing Apt source, key, and definitions with Puppet. https://forge.puppet.com/puppetlabs/apt.

Puppet. 2019b. Puppet Forge. https://forge.puppet.com/.

Puppet Labs. 2016. Idempotence: not just a big and scary word. https://puppet.com/blog/idempotence-not-just-a-big-and-scary-word. [Online; accessed 28-January-2019].

Puppet Labs. 2018. Catalog compilation - Puppet (PE and open source) 5.5. https://puppet.com/docs/puppet/5.5/subsystem_catalog_compilation.html. [Online; accessed 28-January-2019].

Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175.

A. Rahman and L. Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 34–45. https://doi.org/10.1109/ICST.2018.00014

Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference*

on Programming Language Design and Implementation (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 71–81. https://doi.org/10.1145/3180155.3180224

Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie. 2019. Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 527–538. https://doi.org/10.1109/ASE.2019.00056

Manuel Rigger and Zhendong Su. 2020a. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

Manuel Rigger and Zhendong Su. 2020b. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428279

Manuel Rigger and Zhendong Su. 2020c. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

R. Rodriguez. 1986. A System Call Tracer for UNIX. In *USENIX Conference Proceedings* (Atlanta, GA). USENIX Association, Berkeley, CA, 72–80.

Kevin Roebuck. 2012. *Object-Relational Mapping (ORM): High-Impact Strategies-What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Publishing.

Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 220–228.

Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. 2017. Monadic Composition for Deterministic, Parallel Batch Processing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 73 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133897

Andreas Seltenreich. 2020. SQLsmith: A random SQL query generator. https://github.com/anse1/sqlsmith. [Online; accessed 29-July-2020].

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 416–430.

Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 189–200.

Ravjot Singh, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Optimizing the Performance-Related Configurations of Object-Relational Mapping Frameworks Using a Multi-Objective Genetic Algorithm. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 309–320. https://doi.org/10.1145/2851553.2851576

Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. *Pick Your Contexts Well: Understanding Object-Sensitivity*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1535–1547. https://doi.org/10.1109/ICSE43902.2021.00137

Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020a. A Model for Detecting Faults in Build Specifications. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 144 (nov 2020), 30 pages. https://doi.org/10.1145/3428212

Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.),

Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2019.8

Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020b. Practical Fault Detection in Puppet Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3377811.3380384

Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build Scripts with Perfect Dependencies. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 169 (nov 2020), 28 pages. https://doi.org/10.1145/3428237

D. Spinellis. 2012. Don't Install Software by Hand. *IEEE Software* 29, 4 (July 2012), 86–87.

Diomidis Spinellis. 2016. *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional, Boston, MA.

Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-implementation testing of supervised learning software. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2021. Type-Centric Kotlin Compiler Fuzzing: Preserving Test Program Correctness by Preserving Types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 318–328. https://doi.org/10.1109/ICST49551.2021.00044

Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/2338965.2336784

Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

Chengnian Sun, Vu Le, and Zhendong Su. 2016a. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

Chengnian Sun, Vu Le, and Zhendong Su. 2016b. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016c. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. https://doi.org/10.1145/3453483.3454084

Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. Association for Computing Machinery, New York, NY, USA, 366–369. https://doi.org/10.1145/2351676.2351749

TIOBE Software BV. 2021. TIOBE index. https://www.tiobe.com/tiobe-index/. Online accessed; 05-03-2021.

Seth Tisue. 2017. Bye bye JIRA — Scala issues migrated to GitHub scala/bug. https://contributors.scala-lang.org/t/bye-bye-jira-scala-issues-migrated-to-github-scala-bug/715.

Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. *Interactive Metamorphic Testing of Debuggers*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3293882.3330567

Alexandre Torres, Renata Galante, Marcelo S. Pimenta, and Alexandre Jonatan B. Martins. 2017. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology* 82 (2017), 1 – 18. https://doi.org/10.1016/j.infsof.2016.09.009

Jan Tretmans. 2008. *Model Based Testing with Labelled Transition Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 123–133. http://dl.acm.org/citation.cfm?id=2818754.2818772

Edward van der Bent, Juriaan Hage, Joost Visser, and Georgios Gousios. 2018. How good is your puppet? An empirically defined and validated quality model for Puppet. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Campobasso, Italy) *(SANER 2018)*. 164–174.

Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, and Armijn Hemel. 2014. Tracing Software Build Processes to Uncover License Compliance Inconsistencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 731–742. https://doi.org/10.1145/2642937.2643013

Joost Visser, Sylvan Rigal, Gijs Wijnholds, and Zeeger Lubsen. 2016a. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.".

Joost Visser, Sylvan Rigal, Gijs Wijnholds, Pascal Van Eck, and Rob van der Leek. 2016b. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.".

Abhijeet Viswa. 2020. select_for_update() with "of" uses wrong tables from (multi-level) model inheritance. https://code.djangoproject.com/ticket/31246. [Online; accessed 29-July-2020].

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 520–531. https://doi.org/10.1109/ASE.2017.8115663

J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. 2017. A comprehensive study on real world concurrency bugs in Node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 520–531.

Gary Wassermann and Zhendong Su. 2008. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 171–180. https://doi.org/10.1145/1368088.1368112

Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: Interactive System Configuration Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 625–636.

Elaine J Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020a. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428261

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020b. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. https://doi.org/10.1145/3385412.3385985

Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: Reinforcement Learning for Input-Grammar Inference. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/3338906.3338958

Valentin Wüstholz and Maria Christakis. 2020. *Harvey: A Greybox Fuzzer for Smart Contracts*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 244–259.

Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6, 11 (2013), 1319–1330.

Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How Not to Structure Your Database-Backed Web Applications: A Study of Performance Bugs in the Wild. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 763–773. https://doi.org/10.1145/3338906.3338932

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming*

Language Design and Implementation (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

David Zubrow. 2010. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

Acronyms

AFL American Fuzzy Lop. 16, 119, 151

API Application Programming Interface. xi, 23, 24, 25, 43, 45, 47, 48, 50, 51, 56, 66, 83, 85, 91, 113, 114, 129, 130, 131, 136, 140, 141, 142, 150, 162, 163, 164

AQL Abstract Query Language. 83, 84, 85, 86, 87, 88, 89, 90, 91, 111, 112, 113, 134, 153, 161, 162, 164

AST Abstract Syntax Tree. 37, 38, 39, 50, 87, 88, 89, 90, 113, 164

CLP Constraint Logic Programming. 151

DBMS Database Management System. 5, 49, 51, 83, 85, 91, 92, 111, 112, 128, 129, 131, 152, 153

DNS Domain Name System. 57, 135

DSL Domain-Specific Language. 55, 56, 64, 66, 93, 94, 103, 139, 161, 164

EMI Equivalence Modulo Input. 151

I/O Input / Output. 113, 163

IaC Infrastructure as Code. 52, 106, 116, 154

IR Intermediate Representation. 4, 68, 69, 71, 75, 76, 80, 108, 109, 110, 111, 161, 163

JDK Java Development Kit. 20, 24

JLS Java Language Specification. 35, 37

JVM Java Virtual Machine. 7, 20, 22, 27, 29, 30, 45, 55, 62, 110, 118, 147, 151, 155, 158

MSSQL Microsoft's SQL Server. 129, 131

NoREC Non-Optimizing Reference Engine Construction. 153

OOP Object-Oriented Programming. 43, 123

ORM Object-Relational Mapping. vi, x, xii, 5, 6, 8, 47, 48, 49, 50, 51, 52, 65, 66, 67, 83, 84, 85, 88, 91, 92, 128, 129, 130, 131, 132, 133, 134, 152, 153, 159, 160, 161, 162, 164

OS Operating System. 63, 64, 94, 103, 106

PQS Pivoted Query Synthesis. 153

REST Representational State Transfer. 18, 24

SAM Single Abstract Method. 29

SMT Satisfiability Modulo Theories. 83, 88, 89, 90, 112, 152, 153, 154, 159

SPE Skeletal Program Enumeration. 46, 152

SSL Secure Socket Layer. 18

SUT System Under Test. 13, 67, 162

TCE Type-Centric Enumeration. 152

TEM Type Erasure Mutation. xii, 80, 81, 82, 119, 123, 124, 125, 127, 128, 151

TLP Ternary Logic Partitioning. 153

TOM Type Overwriting Mutation. xii, 81, 82, 119, 123, 124, 125, 127, 151, 152

UCTE Unexpected Compile-Time Error. xii, 119, 120, 122

URB Unexpected Runtime Behavior. xii, 120, 122