Software Engineering Research Group MSc Thesis Style

Version of January 13, 2023

Călin-Andrei Georgescu

Software Engineering Research Group MSc Thesis Style

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Călin-Andrei Georgescu
born in Bucharest, Romania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Some Company
With it's address
ThePlace, the Netherlands
`www.url.nl`

Software Engineering Research Group MSc Thesis Style

Author:	Călin-Andrei Georgescu
Student id:	123456
Email:	`caling@protonmail.com`

Abstract

This document describes the standard thesis style for the Software Engineering department at Delft University of Technology. The document and it's source are an example of the use of the standard LaTeX style file. In addition the final appendix to this document contains a number of requirements and guidelines for writing a Software Engineering MSc thesis.

Your thesis should either employ this style or follow it closely.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Company supervisor:	Drs. E.X. Ternal, Some Company
Committee Member:	Dr. S.T.A.F.F. Member, Faculty EEMCS, TU Delft

Preface

This is where you thank people for helping you etc.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus massa pede, feugiat sit amet, mollis in, sodales at, augue. Ut sit amet nisi egestas risus consequat adipiscing. Nulla non diam. Proin volutpat, lacus quis volutpat scelerisque, leo urna rhoncus arcu, vel ultrices dui lacus id lorem. Nam pulvinar adipiscing odio. Etiam tellus lorem, malesuada in, scelerisque sit amet, consequat a, tellus. Curabitur non urna. Mauris facilisis tempor nulla. Nam euismod semper massa. Nullam id nulla. Duis mattis nunc ut ipsum. Proin libero purus, posuere ut, tincidunt sit amet, accumsan sit amet, nisl. Integer commodo. Pellentesque suscipit, diam vel bibendum interdum, magna mauris venenatis lorem, vitae tristique nibh lacus convallis velit. Sed tellus. Mauris placerat lectus ut tellus rutrum blandit. Aliquam erat volutpat.

Suspendisse potenti. Proin sodales eros non lacus. Nam magna sapien, tristique ut, hendrerit ultricies, pretium ut, ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. In libero risus, pellentesque vitae, interdum id, tincidunt ut, sapien. Mauris nec massa sit amet leo dictum pretium. Curabitur iaculis euismod mauris. Donec diam sem, pulvinar at, luctus id, blandit nec, pede. Nam scelerisque sollicitudin nunc. Nam malesuada mauris id ligula. Donec suscipit posuere justo. Mauris sed libero in mi nonummy tincidunt.

Călin-Andrei Georgescu
Delft, the Netherlands
January 13, 2023

Contents

List of Figures

Chapter 1

Introduction

Compilers are ubiquitous and pivotal components at the core of innumerable software systems and programming language ecosystems. As a consequence, the ramifications that arise from flawed compiler implementations can be both far-reaching and severe. The outstanding complexity and feature richness of compilers has created several formidable challenges, the scope of which is perhaps best exemplified by Hoare [25] declaring the task of building a provably correct compiler to be a *Grand Challenge* in computing research. Researchers and practitioners have invested tremendous resources into compiler research in recent decades, attempting to improve performance, quality, and reliability. Such efforts are especially beneficial for newer software systems, that require robust compilation pipelines to establish themselves as worthy alternatives to mature standards.

1.1 Compiler Testing

Because of their cornerstone role in many critical applications, compilers are often subject to thorough quality assurance processes. A significant portion of research focuses on testing as a means of assessing and improving the correctness of compilers. Empirical evidence has spotlighted the necessity of such approaches, with Sun et al. [57] and Holler et al. [26] highlighting the existence of bugs in widely used C compilers and JavaScript engines, respectively. These findings stand out particularly because of the maturity of the software ecosystems that pivot around these technologies and showcases the necessity of effective and adaptable testing methods.

One approach to testing compilers is to assemble manually-written test suites. This strategy has several advantages, including the possibility to individually target specific components of the compiler and the developer's ability to reason about the tests' semantics and expected outputs. However, the labor-intensive nature of the process and the imposing complexity of compiler code bases have led to test case generation methods superseding their manual counterparts [8, 65]. Though compelling and time-effective, automated test case generation strategies concurrently give rise to an arduous set of challenges. The infinite number of correct possible programs, the syntactic dependencies between code snippets, the diversity required to exercise a broad range of compiler features, and the many semantic nuances attributed to each programming language all constitute obstacles that an effective generative algorithm must address.

Fuzzing, or random testing, lends itself naturally to this task thanks to its ability to generate code snippets that trigger diverse behavior within the compiler. Because of this, researchers have proposed a myriad of language-specific approaches, fixated around two overarching paradigms. Algorithms that generate standalone code such as those proposed by Yang et al. [62], Holler et al. [26], Veggalam et al. [61], and Havrikov and Zeller [24] have the ability to generate novel test programs, which in turn may contain seldom encountered constructs and relations. By contrast, mutation-centric approaches such as those put forward by Le et al. [30, 31], Sun et al. [56], and Stepanov et al. [54] rely on an input (often called the seed) that serves as a basis for subsequent transformations. While the former approach may be able to explore more of the space of feasible input programs, the latter effectively leverages large corpora that become available as programming language ecosystems mature.

1.2 The Kotlin Programming Language

Kotlin [52] is a relatively new programming language whose development initiative is led by JetBrains. Initially designed as a Java alternative, the Kotlin ecosystem has steadily grown and currently envelops a broad range of use-cases, including server-side development, web front-end interfaces, and mobile applications. Kotlin showcases several compelling attributes, including expressiveness, safety, and interoperability, which have played a key role in Google embracing a ”Kotlin-first” approach in its popular Android mobile operating system [11].

The number of research initiatives investigating Kotlin's impact in the software development process has risen in tandem with its adoption. Several studies have analyzed both Kotlin's influence on the quality of code bases, as well as developers' perception of Kotlin in comparison to their previous experience. Flauzino et al. [15] and Góis Mateus and Martinez [19] independently found that code bases which at least partially employ Kotlin tend to exhibit fewer code smells than their Java counterparts. Ardito et al. [3] suggest that transitioning projects from Java to Kotlin can meaningfully increase code conciseness. Chauhan et al. [7] provide empirical evidence indicating that Kotlin's coroutine implementation vastly outperforms Java's thread-centric concurrency framework in terms of speed. Oliveira et al. [45] study Android practitioners' opinions regarding Kotlin adoption and reveal that developers find Kotlin easy to adopt and understand. Their study also showcases that developers value Kotlin's null-safety guarantees, as well as its interoperability with Java.

Despite the sharp rise in Kotlin's popularity and an increasing number of systems relying on its ecosystem, the study by Stepanov et al. [54] is the only one to propose a tailored algorithm for automatically testing the Kotlin compiler. Their work heavily relies on manually written test suites to provide seeds for an enumeration-based fuzzer aimed at detecting compilation errors between different compiler implementations. Currently, no tool that is capable of generating entirely novel, semantically meaningful, and diverse test programs for the Kotlin compiler exists. Creating such an algorithm would not only aid the quality assurance process of the Kotlin compiler, but also provide valuable insight relating to which language features require additional work and which fuzzing strategies are best suited for revealing compiler bugs.

1.3 Research Aim and Scope

This thesis seeks to advance the current understanding of the effectiveness of test program-based fuzzing as a means of automated compiler testing. To narrow the scope of this goal, we identify and acknowledge several key limitations. First, this thesis only concerns itself with empirical analyses regarding the Kotlin compiler. Though the prototype implementation may well be extended to target other systems, we use a single benchmark to ensure feasibility. Second, this study investigates the ability of the proposed fuzzing techniques to find bugs in the targeted compiler, and does not emphasize the practical or commercial implications of the uncovered bugs. Further, research concerning other widely studied defect-centric practices such as bug localization, program minimization, and bug deduplication are outside the scope of this study. Within these constraints, we seek to achieve the following research aim:

The aim of this research is to gain insight into how effective test program-based fuzzing is at uncovering bugs in the Kotlin compiler.

To address this aim, we implement a grammar-aided fuzzing tool that generates Kotlin code snippets. The generated pieces of code serve as input to the compiler, whose behavior is in turn assessed through differential testing. We divide the overarching research aim into several, more granular research questions. We first consider the effectiveness of our approach in comparison to existing mutation-based fuzzing algorithms, leading to the following research question:

***RQ1:** How does the effective is test program-based fuzzing compared to mutation-based fuzzing at uncovering bugs in the Kotlin compiler?*

To better understand the reasons behind the relative performance of our approach, we seek to isolate its most crucial components. Such insight establishes a baseline for future research to expand upon by providing insight into which fuzzing techniques result in better performance. We encapsulate this goal into the following research question:

***RQ2:** How effective are different test program-based fuzzing heuristics at uncovering bugs in the Kotlin compiler?*

A vital point to consider when designing a fuzzing algorithm is its potential to adapt to the goals of its users.• TODO show that any random testing approach eventually reaches a saturation point, where novel input is exceedingly unlikely to trigger detecting new faults. To combat this, practitioners can either design new tools, adapt existing ones, or intertwine approaches. While flexible design can help mitigate this shortcoming, studying the bug finding capability of a combination of tools enables the exploration of a broader range of the input space. To explore this potential, we study the effectiveness of integrating of test program- and mutation-based fuzzing. To this end, we propose the following research question:

> ***RQ3:*** *How effective is the combination of test program- and mutation-based fuzzing techniques at uncovering bugs in the Kotlin compiler?*

1.4 Contributions and Significance

This thesis makes both theoretical and practical advances, relevant to both the field of compiler testing and the Kotlin practitioner community. To summarize, we expect this study to result in the following contributions:

1. At least one novel grammar-aided, Kotlin-specific compiler fuzzing algorithm.
2. A modular and configurable prototype implementing the novel algorithm.
3. Empirical anlysis comparing the bug finding capabilities of test program- and mutation-based fuzzers for Kotlin, as well as the combination of the two
4. A replication package containing open-source artifacts, able to reproduce the studies

From a theoretical perspective, we expect our contributions to deepen the insight into the applicability and effectiveness of test program-based fuzzing and its adjacent heuristics, as a means of compiler testing. The significance of this knowledge is tied to the continued interest in establishing reliable quality assurance measures for compilers. On a practical level, we expect the resulting prototype implementation to become a useful tool for improving the reliability and quality of the Kotlin compiler. Compiler engineers can leverage such a tool as an additional step in a continuous integration pipeline, or choose to employ it when developing novel features in future versions of the language.

1.5 Overview of the Study

This thesis consists of eight additional chapters situated in three main parts. Part I consists of Chapters 2 and 3, and provides an overview of the relevant body of knowledge and literature. Chapter 2 introduces the theoretical background that constitutes the foundation of this study. Chapter 3 discusses related work and positions this thesis within the current research landscape.

Part II encompasses Chapters 4, 5, 6, and 7 and details the main contributions of this thesis. Chapter 4 describes the conceptual outline of the fuzzing procedure and the rationale behind its underpinning design choices. Chapter 5 delineates the prototypical implementation of our tool and highlights its practical applications. Chapter 6 outlines the design of the empirical study carried out to evaluate the performance of our tool. Chapter 7 analyzes the results of the empirical study and ties them to the main research questions.

Part III includes Chapters 8 and 9 and reflects upon the outcomes of this study. Chapter 8 fixates on discussing the main findings of this research within a broader landscape and considers their practical ramifications with respect to the Kotlin compiler. Finally, Chapter 9 reflects upon this thesis and makes concluding recommendations for future research.

Chapter 2

Background

This chapter establishes the theoretical basis required for investigating the aim introduced in Section 1.3. This includes the individual nuances of the Kotlin compiler, the encompassing field of software testing, and the theoretical background of fuzzing. In addition, this chapter lays a contextual and historical foundation required for the analysis of related work examined in Chapter 3.

2.1 The Kotlin Compiler

The Kotlin project began in 2010, with version `1.0` releasing in 2016 [54]. Following Google's decision to support it as an official Android programming language [12], Kotlin's popularity increased drastically. This exceptional growth is perhaps best exemplified through StackOverflow's annual developer survey, which aggregates data from tens of thousands of active developers from varied backgrounds. This survey clearly showcases Kotlin's sharp ascent: the percentage of developers who actively use Kotlin increased from 0.12 per cent in 2016 to 9.16 per cent in 2022 [46, 47]. Such sustained growth brought both opportunities and challenges to the expanding community, which resulted in the development of increasingly complex and feature-rich tools.

Though it supports several platforms, the Kotlin ecosystem pivots around the Java Virtual Machine (JVM) system and its adjacent Java bytecode instruction set. Kotlin is one of several successful languages to leverage the JVM, together with Java, Scala, Clojure, and Groovy. One can think of the JVM as a compatibility layer between a Java bytecode input and a target-specific target language. The advantage of this system lies in the flexibility it affords language engineers. Leveraging the JVM as an intermediate compilation step, language designers can focus on building a single compilation tool that targets Java bytecode. This approach delegates the task of machine code generation to the JVM and frees co12mpiler engineers from platform- and hardware-specific idiosyncrasies. This study focuses on testing the end-to-end compilat1ion pipeline of Kotlin through the open-source `kotlinc` utility.

2.2 Software Testing

Testing is an integral component of the Software Development Lifecycle (SDLC) [27]. At its core, software testing is the process of validating the behavior of a target System Under Test (SUT). Over time, specialized testing approaches have emerged as the practice of software engineering itself matured. To effectively navigate the numerous alternatives, we use the taxonomy established by Umar [60].

Black-, White-, and Grey-Box Testing

A key characteristic of a testing strategy consists of the requirements it imposes on the SUT. White-Box (WB) approaches require a full understanding of the target system, which often translates to unrestricted access to its source code. The information extracted through access to the system's structure and implementation often provides valuable guidance criteria. By contrast, Black-Box (BB) approaches function without access to the underlying source code, which diminishes the amount of information they can exploit. This sacrifice, however, extends the applicability of BB approaches to a broader range of systems. Grey-Box (GB) approaches constitute a trade-off between the two extremes, and assume access to a subset system components. This requires some level of understanding of the SUT's behavior, but imposes lesser constraints than WB standards.

Unit-, Integration-, and System-Level Testing

Testing methods also differ in the degree to which they exercise the SUT. Three main paradigms emerge from this distinction: Unit-Level Testing (UT), Integration-Level Testing (IT), and System-Level Test (ST). Daka and Fraser [10] demarcate UT as generally small and automatically executable tests that aim to validate the correctness of small components (units) of the SUT. Leung and White [33] establish IT at a higher-level scope, targeting the integration of several smaller modules within a system. Finally, Umar [60] identify ST as tests that validate the entire system's compliance against its underlying requirements.

The Oracle Problem

Irrespective of access to the SUT and its level of granularity, test cases must be capable of distinguishing between correct and faulty behavior. This challenge and is referred to as the *test oracle problem*, and its automation is among the greatest obstacles in automating test case generation [5]. Approaches like specification inference and regression testing offer approximate oracles that can partially automate this process. Originally proposed by McKeeman [38] in 1998, Differential Testing (DT) offers a compelling and versatile alternative, that uses different versions of the same SUT to heuristically discover bugs. DT identifies possibly incorrect behavior when at least one version of the SUT returns a result that differs from the others. While inexact in its nature, DT has become popular in the field of compiler testing thanks to its lightweight constraints, which increase its applicability to especially complex pieces of software.

2.3 Automated Test Case Generation

Traditional (manual) software testing poses several practical challenges, not least because of its imposing complexity. Candea and Godefroid [6] regard testing as the most demanding and laborious task in the SDLC. Lehman [32] established that to maintain relevance, software systems must adapt, while Zaidman et al. [63] showed that test suites often evolve in tandem with their code bases. These findings suggest that to adequately test an evolving system, practitioners not only allocate significant resources to writing test cases, but also do so throughout the software's lifetime. In an effort to alleviate the human resources involved in the testing process, researchers developed a plethora of automated test case generation methods. The remainder of this section briefly covers the most prevalent techniques in this field.

2.3.1 Fuzzing

Miller et al. [41] coined the term *fuzz* in 1979 referring to a program capable of generating a stream of random characters. This program served as a means of assessing the reliability of several tools that were part of the Unix Operating System (OS). At its core, fuzzing is perhaps the simplest possible implementation of input generation: the program simply samples data driven by a pseudo-random random generator and executes the target program with the generated input. This original formulation of fuzzing bears a close resemblance to well established test case generation algorithms such as Random Testing (RT) [14] and its descendant, Adaptive Random Testing (ART) [9]. Nonetheless, its simplicity and deceptive effectiveness as a defect discovery tool led to fuzzing becoming one of the most prevalent input and test case generation techniques.

Since its inception, the meaning of fuzzing has adapted to better accommodate the extensive domain of applications that benefited from its application. Such applications include security testing [44, 55], data structure signature generation [13], database system management testing [66], and REST API testing [4, 18]. To encapsulate the various complexities that fuzzing has grown to exhibit, Manès et al. [37] define fuzzing as the execution of the SUT using input sampled from an input space protruding that of the SUT's. Saavedra et al. [49] adopt a similar definition and additionally establish a taxonomy, that distinguishes between three fundamental fuzzing paradigms:

- **Mutation-based fuzzers** rely on predefined input (seeds), which they manipulate in heuristic ways to cover the input space. The performance of these approaches is often heavily correlated with the quality of the corpus of seed inputs as the main source of expliting program knowledge.

- **Generation-based fuzzers** sample the input space guided by an a-priori configuration or specification of the SUT. Such approaches are generally more sophisticated than their mutation-based counterparts, which they tend to outperform [49]. However, these benefits come at the cost of complexity and adherence to the limitations provided through the configuration.

- **Evolutionary fuzzers** employ evolutionary computation techniques, including fitness evaluation, mutation, crossover, and selection, to iteratively improve the quality of a set of solutions. These algorithms benefit from the rich literature surrounding evolutionary computation and may be better suited at exploring specific volumes of the search space. Evolutionary fuzzers impose an additional requirement on the SUT, namely that it must be capable of generating meaningful feedback for evaluating input.

2.3.2 Search-Based Software Testing

Software engineering entails numerous problems that present nuanced, multivariate, and often conflicting objectives. Metaheuristic search algorithms lend themselves naturally as frameworks for effectively exploring the space of possible solutions with the aim of finding (near-)optimal candidates [22]. The study and application of metaheuristics to software-centric tasks has brought forth many state-of-the-art approaches, and is jointly known as Search-Based Software Engineering (SBSE). Approaches fixating on the testing phase of the SDLC belong to a branch of SBSE referred to as Search-Based Software Testing (SBST).

Search-based methods generally involve more sophisticated mechanisms than fuzzing. Irrespective of the problem, a notion of *fitness* allows the comparison of any two (valid) solutions on the basis of at least one *objective*. In the realm of SBST, solutions may consist of test cases, and the corresponding fitness of a solution may be determined on the structural coverage a solution (test case) achieves, with the underlying objective of uncovering bugs. Search algorithms often change solutions by means of *mutation*. Mutation operators introduce small, possibly stochastic perturbations to alter the representation of a candidate, implicitly equipping the search space with a *topology*. Over the years, researchers developed successful variations of this high-level framework, which in many cases are able to satisfactorily solve combinatorial problems in reasonable time [40]. The remainder of this subsection coarsely examines some of the most widespread metaheuristic search approaches and grounds them through an SBST lens.

Local Search Metaheuristics

Local Search (LS) is among the most straight-forward metaheuristic frameworks. Hill Climbing (HC) is a simple implementation of LS, that greedily exploits the neighborhood structure of a solution. It consists of an iterated procedure that repeatedly attempts to improve a solution by applying mutations [50]. If the mutated candidate exceeds the fitness of the previous solution, the former replaces the latter, concluding the iteration. This basic strategy has been an effective tool for solving combinatorial problems for over half a century [35]. Both greedy and stochastic variations of LS exist, providing compromises between the quality of the final solution and the time to convergence. From an SBST perspective, Arcuri [1] shows that LS asymptotically outperforms RT, but falls short of more sophisticated global search techniques. Harman and McMinn [23] further reinforce this hierarchy with empirical evidence.

Miller and Spooner [42] have demonstrated the first application of HC in automated test data generation in 1976. Since then, researchers developed several variations of the LS framework, such as Tabu Search (TS) [17] and Simulated Annealing (SA). Introduced by Kirkpatrick et al. [29] in 1983, SA finds its inspiration in statistical mechanics. SA behaves similarly to LS, but additionally incorporates an annealing mechanic that enables the stochastic exploration of candidates that exhibit a lower fitness than previous solutions. A cooling schedule governs this mechanisms, which generally encourages a broader exploration of the domain early in the search procedure.

Evolutionary Algorithms

While LS and its variants provide useful optimization frameworks, Global Search (GS) techniques often enable more effective and robust domain sampling mechanisms [39]. Of the many GS search algorithms developed over the years, Evolutionary Algorithms (EA) have gained the most notoriety within the SBSE community [21].

EAs are widely applicable and robust, population-based algorithms that practitioners have employed in a broad range of engineering tasks [21]. Though its conceptual origins trace back to Alan Turing's "*Computing Machinery and Intelligence*" [59], it was John Holland [28] who in 1975 formalized and cemented the modern EA paradigm. A plethora of specialized EAs have emerged from a vast spectrum of research domains, but arguably the most widespread variation is the class of Genetic Algorithms (GA). Algorithm 8 provides the pseudocode for the the simple GA proposed by Holland. This constitutes the foundation of specialized automated test case generation algorithms for several tasks, including unit testing [58, 48], data flow testing [16], and REST API testing [2]. Biology-inspired by nature, GAs share several core ingredients:

- A *representation* of *individuals*, often referred to as *chromosomes*. Each individual represents a solution to the underlying problem. Together, the collection of individuals present at a given iteration forms a *population*.
- A *fitness* function that determines the quality of individuals. The fitness function is generally linked to one or more underlying *objectives*.
- A *variation* mechanism that alters the individuals in a population to create *offspring*. Variation often consists of a *mutation* operator that changes an individual's chromosome, and a *crossover* operator that combines individuals to create new offspring.
- A *selection* mechanism that pressures the is responsible for pressuring the population toward better solutions. Selection is generally driven by the fitness of individuals and strongly influences the rate of convergence of the algorithm.

2.4 Context-Free Grammars

Grammars are powerful tools used to study, specify, and understand languages. A Context-Free Grammar (CFG) leverages a recursive structure to specify a language's syntactic prop-

Algorithm 1: Simple Genetic Algorithm

```
Input  : Population size $N$
Output: $N$ solutions
1 begin
2     $t \leftarrow 1$
3     $P_1 \leftarrow$ InitializeAndEvaluatePopulation($n$)
4     while $\neg$ ShouldTermiante($P_t$) do
5         $O_t \leftarrow$ CreateAndEvaluateOffspring($P_t, n$)
6         $P_{t+1} \leftarrow$ SelectIndividuals($P_t, O_t, n$)
7         $t \leftarrow t + 1$
8     return $P_t$
```

erties, without accounting for contextual semantics. CFGs have become central to many fuzzers thanks to their ability to effectively constrain generative processes to input that is syntactically meaningful for the target SUT. For the remainder of this study, we use the terms terms *grammar* and CFG interchangeably, and we adhere to the definition put forward by Sipser [51].

Formally, a CFG is a 4-tuple $G = (V, \Sigma, R, S)$, with (i) V a finite set of *variables* or *symbols*, (ii) Σ a set of *terminals* or *terminal symbols* such that $\Sigma \cap V = \emptyset$, (iii) $R \subseteq V \times \Sigma \cup R$ a set of *rules* or *productions* that define the ways in which a variable can be *expanded*, and (iv) $S \in V$ a *start symbol* or *start variable*.

Chapter 3

Related Work

This chapter surveys relevant academic work within the fields of compiler testing and fuzzing. Section 3.1 examines existing automated compiler testing tools and their underlying rationale. Section 3.2 surveys algorithms for specification fuzzing, a problem whose elemental challenges share many commonalities with of compiler testing.

3.1 Automated Compiler Testing

Over the years, specialized testing approaches have emerged as tools for improving compiler reliability. To navigate the ample space of approaches, we adopt a simplified version of the taxonomy proposed by Chen et al. [8]. Most notably, we distinguish between test program generation and program mutation approaches, discussed in subsections 3.1.1 and 3.1.2, respectively. The former generates programs for scratch, without any external seed, while the latter mutates existing programs to generate new variations. Test program generation approaches also differ in the way in which they utilize the grammar of the target language. Based on this property, Chen et al. [8] distinguish three categories. Grammar-directed approaches solely rely on the language grammar to generate novel test cases. Grammar-aided approaches heuristically exploit a given grammar, which is often enriched with semantically significant context. Finally, fuzzers may not directly rely on a grammar, and instead define sound heuristic that respect a given linguistic syntax.

3.1.1 Test Program Generation

Yang et al. [62] propose CSMITH, a grammar-aided test program generation tool aimed at finding bugs in commercial C compiler using DT. CSMITH first creates a set of `struct` type declaration, with a stochastically-driven number of members. It then stores these generated types as additional information to its grammar. CSMITH then creates programs in a top-down fashion, starting from a single program entry point, the `main` function. At each iteration, it interweaves two steps: (i) instantiating a randomly chosen production from its grammar, accompanied by target completion, and (ii) validating the intermediate program using a series of safety checks with the goal of avoiding undefined behavior. The generation process follows an ”on-demand” paradigm, where `struct` members and functions are generated when previously generated expressions access them. CSMITH's effectiveness has inspired several variations such in Lidbury et al.'s CLSMITH [34] and Morisset et al.'s [43] tool for uncovering concurrency bugs.

Livinskii et al. [36] introduce YARPGEN, a C/C++ program generation tool that aims to increase the expressiveness and diversity of test programs. YARPGEN proceeds in top-down fashion and does not explicitly utilize a formal language grammar. Instead, it tracks a *type environment* that stores all visible composite data types, which implicitly guide the generative process. The environment is iteratively enriched with newly generated types, each having access to all previous entries. Once established, the generated types drive the creation of global variables that can be used in future statements. YARPGEN incrementally builds functions that utilize the previously generated types and symbols. Each expression in a statement is recursively constructed by a Undefined Behavior (UB). To further increase the soundness of its program, YARPGEN performs local fixes that eliminate several common UB instances. Finally, *generation policies* provide a mechanism for adapting the probability distribution dictating YARPGEN's choices. This feature serves as a way of incorporating prior knowledge into the program and enables users to target specific features of the language or compiler.

Holler et al. [26] developed LANGFUZZ, a grammar-directed fuzzing approach that combines generative and mutative processes. The mutation process follows a two-phase approach, first *learning* code fragments from existing test suites, and then constructing executable mutants from its input. A grammar enables the extraction of code fragments, which LANGFUZZ obtains by breaking down hand-written test suites. In practice, a *fragment pool* emerges from the non-terminal instantiations of grammar rules within the input test suites. To complement this mutative process, LANGFUZZ leverages a generative procedure that stochastically expands the input grammar in a breadth-first manner. The two approaches work in tandem, comprising a trade-off between the flexibility of the input grammar and the rich semantics of test suites.

Han et al. [20] develop the notion of *semantics-aware assembly*, which they implement in the CODEALCHEMIST tool, aimed at JavaScript engine testing. The key concept in semantics-aware assembly consists of building blocks, also referred to as *code bricks*, a notion similar to LANGFUZZ's *code fragments* [26]. A code brick consists of a valid JavaScript Abstract Syntax Tree (AST) that is additionally annotated with an *assembly constraint*. Assembly constraints are sets of pre- and post-condition that must be satisfied for two code bricks to be compatible. Such constraints specify what variables must be defined in preceding code bricks, and what additional objects are visible in subsequent bricks, respectively. CODEALCHEMIST produces fuzzed programs by splitting input seeds into code bricks, which it then stochastically interconnects in ways that adhere to the bricks' assembly constraints.

3.1.2 Program Mutation

Le et al. [30] introduce the notion of Equivalence Modulo Inputs (EMI), a framework for establishing semantic equivalence between programs. Given a set of valid program inputs I, two programs P_1 and P_2 are equivalent modulo I if and only if their outputs are identical for all inputs $i \in I$. This notion lends itself naturally to the DT framework: since P_1 and P_2 are expected to produce identical results for all input in I, any discrepancies that emerge during execution can be attributed to compiler faults. ORION is a mutation-centric realization of this framework for the C programming language. To derive semantically equivalent programs, ORION profiles the execution of a program P over an input set I and determines which code snippets remain are not executed. ORION then exploits this information to stochastically prune the unexecuted lines of code, maintaining the semantics of the original program P.

Le et al. [31] extend ORION in a tool named ATHENA, equipped with a heuristic guidance mechanism. Still reliant on the EMI framework, ATHENA improves the random mutation-driven search of ORION with heuristic diversity-based probability distribution. The goal of this strategy is to exercise an ample range of compiler optimization mechanics, which is more likely as when the equivalent programs themselves are more diverse. To achieve this, ATHENA leverages a Markov Chain Monte Carlo (MCMC) sampling technique. A CFG-based similarity metric guides the MCMC search, favoring programs that significantly differ in their CFG representation. Novel programs emerge from either the deletion of unused code, or the insertion of code in unused sections.

Zhang et al. [64] introduce the notion of Skeletal Program Enumeration (SPE), a framework for exhaustively representing variable usage patterns in small programs. SPE breaks a program down into three components: a syntactic skeleton, a collection of variable placeholders, and a set of variables. Variable usages within a program are conceptually transformed into *holes*, that can be *filled* to *realize* a concrete program. This framework enables the enumeration of all possible arrangements of variables in such a way that filling the holes results in novel programs. The authors further define the concept of α*-equivalence* between programs. Two programs are α-equivalent if they exhibit the same control- and data-dependence properties. Notably, naive enumeration can result in numerous α-equivalent programs under different variable permutations. To avoid the generation of such equivalent programs due to α*-renaming*, the authors show the SPE can be reduced to the *set partition problem*, which displays the same equivalence structure. The proposed algorithm handles variable scopes by promoting certain *local holes* to global ones and computing the Cartesian product between the two disjoint sets.

Stepanov et al. [54] propose Type-Centric Enumeration (TCE), an extension of SPE that strongly emphasizes types as a powerful medium for generating semantically diverse programs. Similarly to SPE, TCE deconstructs a program into a skeleton, a collection of placeholders, and a set of variables. IN addition to this breakdown, however, TCE equips each placeholder with an adherent type, and further considers a set of *callabels* that it uses alongside variables. The algorithm proceeds in two phases. The *generation* phase is responsible for the creation of a set of typed expressions, which it extracts from the callable set of its seed program. The *mutation* phase operates on a different seed program, which it first merges with that of the previous stage. Subsequently, the expressions obtained in the generation phase fill type-compatible placeholders placed in the latter seed program. Additional expressions obtained either from the standard library or primitive types are additionally employed to populate remaining unfilled placeholders.

3.2 Specification Fuzzing

Steinhöfel and Zeller [53] propose the Input Specification Language (ISLA), a declarative language for integrating context-dependent semantic constraints in otherwise coarser CFG definitions. Programming languages often impose complex semantic relations that CFGs cannot encapsulate. This includes even commonplace constraints such as *"a variable must be declared before it is assigned"*. ISLA builds upon constraint solvers to incorporate complex annotations that can model such relations. The ISLA solver iteratively expands a tree representation each constraint annotation applied to the grammar according to production rules. Expansions that do not match the annotations are excluded from the cost function-guided solving procedure, as valid candidates emerge. Eventually, if a the solver discovers a matching expansion, it first (approximately) translates it to a regular expression representation before querying a specialized constraint solver. ISLA then converts this solution back to a derivation tree representation and substitutes it in the original grammar graph.

Havrikov and Zeller [24] study coverage criteria of CFGs and accompanying coverage-centric generative algorithms. *Symbol coverage* is a straight-forward metric for evaluating how much of a specification grammar a derivation tree covers. Symbolic coverage is simply measured the fraction of symbolic nodes in the grammar, that the derivation tree covers. *k-Path* coverage is a more nuanced criterion that additionally takes into account the *context* surrounding a symbolic node's position within a tree. Formally, a *k-path* is a directional sequence of nodes in a derivation that contains exactly *k* symbolic nodes. The concept of *k-path* coverage naturally follows from this definition: a collection of inputs achieves *k-path* coverage if each *k-path* in the grammar graph is present in the underlying set of derivation trees. The authors derive an enumeration-based *k-path* algorithm stochastically iterates through all *k-paths* up to a user-defined depth and instantiating encompassing derivation trees.

Chapter 4

Algorithm

Chapter 5

Tool

Chapter 6

Empirical Study

Chapter 7

Results

Chapter 8

Discussion

Chapter 9

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

9.1 Contributions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus massa pede, feugiat sit amet, mollis in, sodales at, augue. Ut sit amet nisi egestas risus consequat adipiscing. Nulla non diam. Proin volutpat, lacus quis volutpat scelerisque, leo urna rhoncus arcu, vel ultrices dui lacus id lorem. Nam pulvinar adipiscing odio. Etiam tellus lorem, malesuada in, scelerisque sit amet, consequat a, tellus. Curabitur non urna. Mauris facilisis tempor nulla. Nam euismod semper massa. Nullam id nulla. Duis mattis nunc ut ipsum. Proin libero purus, posuere ut, tincidunt sit amet, accumsan sit amet, nisl. Integer commodo. Pellentesque suscipit, diam vel bibendum interdum, magna mauris venenatis lorem, vitae tristique nibh lacus convallis velit. Sed tellus. Mauris placerat lectus ut tellus rutrum blandit. Aliquam erat volutpat.

Suspendisse potenti. Proin sodales eros non lacus. Nam magna sapien, tristique ut, hendrerit ultricies, pretium ut, ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. In libero risus, pellentesque vitae, interdum id, tincidunt ut, sapien. Mauris nec massa sit amet leo dictum pretium. Curabitur iaculis euismod mauris. Donec diam sem, pulvinar at, luctus id, blandit nec, pede. Nam scelerisque sollicitudin nunc. Nam malesuada mauris id ligula. Donec suscipit posuere justo. Mauris sed libero in mi nonummy tincidunt.

Suspendisse porta massa at nulla. Nam a ante eu orci consectetuer tincidunt. Cras interdum mi sed purus. Aliquam lacinia convallis diam. Praesent tristique vehicula leo. Maecenas egestas erat at mauris. Nam eget nibh. Nunc eleifend dolor ac est. Vivamus in justo. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam eget lectus. Sed rutrum pulvinar risus. Morbi in neque non dolor sodales pellentesque. Curabitur eu lectus. Sed tempor. Morbi eget enim. In dui erat, rutrum rhoncus, pharetra at, adipiscing id, odio. Aliquam pretium est ac turpis. Cras aliquam, lectus a suscipit ornare, est dolor mollis nulla, sed interdum justo ligula semper erat. Fusce vestibulum enim vitae felis.

9.2 Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus massa pede, feugiat sit amet, mollis in, sodales at, augue. Ut sit amet nisi egestas risus consequat adipiscing. Nulla non diam. Proin volutpat, lacus quis volutpat scelerisque, leo urna rhoncus arcu, vel ultrices dui lacus id lorem. Nam pulvinar adipiscing odio. Etiam tellus lorem, malesuada in, scelerisque sit amet, consequat a, tellus. Curabitur non urna. Mauris facilisis tempor nulla. Nam euismod semper massa. Nullam id nulla. Duis mattis nunc ut ipsum. Proin libero purus, posuere ut, tincidunt sit amet, accumsan sit amet, nisl. Integer commodo. Pellentesque suscipit, diam vel bibendum interdum, magna mauris venenatis lorem, vitae tristique nibh lacus convallis velit. Sed tellus. Mauris placerat lectus ut tellus rutrum blandit. Aliquam erat volutpat.

Suspendisse potenti. Proin sodales eros non lacus. Nam magna sapien, tristique ut, hendrerit ultricies, pretium ut, ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. In libero risus, pellentesque vitae, interdum id, tincidunt ut, sapien. Mauris nec massa sit amet leo dictum pretium. Curabitur iaculis euismod mauris. Donec diam sem, pulvinar at, luctus id, blandit nec, pede. Nam scelerisque sollicitudin nunc. Nam malesuada mauris id ligula. Donec suscipit posuere justo. Mauris sed libero in mi nonummy tincidunt.

Suspendisse porta massa at nulla. Nam a ante eu orci consectetuer tincidunt. Cras interdum mi sed purus. Aliquam lacinia convallis diam. Praesent tristique vehicula leo. Maecenas egestas erat at mauris. Nam eget nibh. Nunc eleifend dolor ac est. Vivamus in justo. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam eget lectus. Sed rutrum pulvinar risus. Morbi in neque non dolor sodales pellentesque. Curabitur eu lectus. Sed tempor. Morbi eget enim. In dui erat, rutrum rhoncus, pharetra at, adipiscing id, odio. Aliquam pretium est ac turpis. Cras aliquam, lectus a suscipit ornare, est dolor mollis nulla, sed interdum justo ligula semper erat. Fusce vestibulum enim vitae felis.

9.3 Discussion/Reflection

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus massa pede, feugiat sit amet, mollis in, sodales at, augue. Ut sit amet nisi egestas risus consequat adipiscing. Nulla non diam. Proin volutpat, lacus quis volutpat scelerisque, leo urna rhoncus arcu, vel ultrices dui lacus id lorem. Nam pulvinar adipiscing odio. Etiam tellus lorem, malesuada in, scelerisque sit amet, consequat a, tellus. Curabitur non urna. Mauris facilisis tempor nulla. Nam euismod semper massa. Nullam id nulla. Duis mattis nunc ut ipsum. Proin libero purus, posuere ut, tincidunt sit amet, accumsan sit amet, nisl. Integer commodo. Pellentesque suscipit, diam vel bibendum interdum, magna mauris venenatis lorem, vitae tristique nibh lacus convallis velit. Sed tellus. Mauris placerat lectus ut tellus rutrum blandit. Aliquam erat volutpat.

Suspendisse potenti. Proin sodales eros non lacus. Nam magna sapien, tristique ut, hendrerit ultricies, pretium ut, ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. In libero risus, pellentesque vitae, interdum id, tincidunt ut, sapien. Mauris nec massa sit amet leo dictum pretium. Curabitur iaculis euismod mauris. Donec diam sem, pulvinar at, luctus id, blandit nec, pede. Nam scelerisque sollicitudin nunc. Nam malesuada mauris id ligula. Donec suscipit posuere justo. Mauris sed libero in mi nonummy tincidunt.

Suspendisse porta massa at nulla. Nam a ante eu orci consectetuer tincidunt. Cras interdum mi sed purus. Aliquam lacinia convallis diam. Praesent tristique vehicula leo. Maecenas egestas erat at mauris. Nam eget nibh. Nunc eleifend dolor ac est. Vivamus in justo. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam eget lectus. Sed rutrum pulvinar risus. Morbi in neque non dolor sodales pellentesque. Curabitur eu lectus. Sed tempor. Morbi eget enim. In dui erat, rutrum rhoncus, pharetra at, adipiscing id, odio. Aliquam pretium est ac turpis. Cras aliquam, lectus a suscipit ornare, est dolor mollis nulla, sed interdum justo ligula semper erat. Fusce vestibulum enim vitae felis.

Nulla tempus gravida sapien. In id diam ac augue congue interdum. Aliquam at nibh a diam feugiat eleifend. Aliquam luctus est. Curabitur vehicula sapien sed pede. Vivamus auctor odio ac ante. Etiam pretium consectetuer dui. Mauris ornare lacus et felis. Aliquam tortor turpis, ornare ut, ornare quis, adipiscing sed, dolor. Fusce quam sem, pharetra vel, consequat ut, porttitor eu, tellus.

9.4 Future work

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus massa pede, feugiat sit amet, mollis in, sodales at, augue. Ut sit amet nisi egestas risus consequat adipiscing. Nulla non diam. Proin volutpat, lacus quis volutpat scelerisque, leo urna rhoncus arcu, vel ultrices dui lacus id lorem. Nam pulvinar adipiscing odio. Etiam tellus lorem, malesuada in, scelerisque sit amet, consequat a, tellus. Curabitur non urna. Mauris facilisis tempor nulla. Nam euismod semper massa. Nullam id nulla. Duis mattis nunc ut ipsum. Proin libero purus, posuere ut, tincidunt sit amet, accumsan sit amet, nisl. Integer commodo. Pellentesque suscipit, diam vel bibendum interdum, magna mauris venenatis lorem, vitae tristique nibh lacus convallis velit. Sed tellus. Mauris placerat lectus ut tellus rutrum blandit. Aliquam erat volutpat.

Suspendisse potenti. Proin sodales eros non lacus. Nam magna sapien, tristique ut, hendrerit ultricies, pretium ut, ante. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. In libero risus, pellentesque vitae, interdum id, tincidunt ut, sapien. Mauris nec massa sit amet leo dictum pretium. Curabitur iaculis euismod mauris. Donec diam sem, pulvinar at, luctus id, blandit nec, pede. Nam scelerisque sollicitudin nunc. Nam malesuada mauris id ligula. Donec suscipit posuere justo. Mauris sed libero in mi nonummy tincidunt.

Bibliography

[1] Andrea Arcuri. Theoretical analysis of local search in software testing. In *International Symposium on Stochastic Algorithms*, pages 156–168. Springer, 2009.

[2] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[3] Luca Ardito, Riccardo Coppola, Giovanni Malnati, and Marco Torchiano. Effectiveness of kotlin vs. java in android app development tasks. *Information and Software Technology*, 127:106374, 2020.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[6] George Candea and Patrice Godefroid. Automated software test generation: some challenges, solutions, and recent advances. *Computing and Software Science*, pages 505–531, 2019.

[7] Kshitij Chauhan, Shivam Kumar, Divyashikha Sethia, and Mohammad Nadeem Alam. Performance analysis of kotlin coroutines on android in a model-view-intent architecture pattern. In *2021 2nd International Conference for Emerging Technology (INCET)*, pages 1–6. IEEE, 2021.

[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53 (1):1–36, 2020.

[9] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[10] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[11] Android Developers. Android's kotlin-first approach, 2019. URL `https://developer.android.com/kotlin/first`. Visited on 2022-12-29.

[12] Android Developers. Celebrating 5 years of kotlin on android, 2022. URL `https://android-developers.googleblog.com/2022/08/celebrating-5-years-of-kotlin-on-android.html`. Visited on 2022-12-29.

[13] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, 2009.

[14] Joe W Duran and Simeon Ntafos. A report on random testing. In *ICSE*, volume 81, pages 179–183. Citeseer, 1981.

[15] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius HS Durelli, and Rafael S Durelli. Are you still smelling it? a comparative study between java and kotlin language. In *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, pages 23–32, 2018.

[16] Moheb R Girgis. Automatic test data generation for data flow testing using a genetic algorithm. *J. Univers. Comput. Sci.*, 11(6):898–915, 2005.

[17] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[18] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.

[19] Bruno Góis Mateus and Matias Martinez. An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, 24(6):3356–3393, 2019.

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[21] Mark Harman. Software engineering meets evolutionary computation. *Computer*, 44 (10):31–39, 2011.

[22] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[23] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, 2007.

[24] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199. IEEE, 2019.

[25] Tony Hoare. The verifying compiler: A grand challenge for computing research. In *International Conference on Compiler Construction*, pages 262–272. Springer, 2003.

[26] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[27] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.

[28] Holland John. Adaptation in natural and artificial systems. *Ann Arbor*, 1975.

[29] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[30] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[31] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[32] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[33] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[34] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[35] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[36] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–25, 2020.

[37] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[38] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[39] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[40] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[41] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[42] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3):223–226, 1976.

[43] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. *ACM SIGPLAN Notices*, 48(6):187–196, 2013.

[44] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2): 58–62, 2005.

[45] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the adoption of kotlin on android development: A triangulation study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216. IEEE, 2020.

[46] Stack Overflow. Stack overflow 2016 developer survey, 2016. URL `https://insights.stackoverflow.com/survey/2016`. Visited on 2022-12-29.

[47] Stack Overflow. Stack overflow 2022 developer survey, 2022. URL `https://survey.stackoverflow.co/2022/#most-popular-technologies-language`. Visited on 2022-12-29.

[48] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[49] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[50] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of cognitive science*, 81:82, 2006.

[51] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1): 27–29, 1996.

[52] JetBrains s.r.o. Kotlin, 2010. URL `https://kotlinlang.org/`. Visited on 2022-12-29.

[53] Dominic Steinhöfel and Andreas Zeller. Input invariants. In *ESEC/FSE 2022: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 583—-594, 2022.

[54] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 318–328. IEEE, 2021.

[55] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[56] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.

[57] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 294–305, 2016.

[58] Paolo Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[59] Alan M Turing and J Haugeland. Computing machinery and intelligence. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, pages 29–56, 1950.

[60] Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. 2020.

[61] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[63] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3): 325–364, 2011.

[64] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

[65] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *2009 ICSE Workshop on Automation of Software Test*, pages 36–43. IEEE, 2009.

[66] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

Appendix A

Acronyms

ART Adaptive Random Testing. 8

AST Abstract Syntax Tree. 13

BB Black-Box. 7

CFG Context-Free Grammar. 10, 11, 14, 15

DT Differential Testing. 7, 12, 14

EA Evolutionary Algorithms. 10

EMI Equivalence Modulo Inputs. 13, 14

GA Genetic Algorithms. 10

GB Grey-Box. 7

GS Global Search. 10

HC Hill Climbing. 9, 10

IT Integration-Level Testing. 7

JVM Java Virtual Machine. 6

LS Local Search. 9, 10

MCMC Markov Chain Monte Carlo. 14

OS Operating System. 8

RT Random Testing. 8, 9

SA Simulated Annealing. 10

SBSE Search-Based Software Engineering. 9, 10

SBST Search-Based Software Testing. 9

SDLC Software Development Lifecycle. 7–9

SPE Skeletal Program Enumeration. 14

ST System-Level Test. 7

SUT System Under Test. 7–9, 11

TCE Type-Centric Enumeration. 14

TS Tabu Search. 10

UB Undefined Behavior. 13

UT Unit-Level Testing. 7

WB White-Box. 7

Appendix B

Glossary

program mutation A compiler testing technique that generates alters existing programs. 12

test program generation A compiler testing technique that generates novel valid programs without relying on a pre-existing corpus. 12