Delft University of Technology

Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler

Georgescu, Călin; Olsthoorn, Mitchell; Derakhshanfar, Pouria ; Akhin, Marat; Panichella, Annibale

DOI
10.1145/3663529.3663864

Publication date
2024

Document Version
Final published version

Published in
FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering

Citation (APA)
Georgescu, C., Olsthoorn, M., Derakhshanfar, P., Akhin, M., & Panichella, A. (2024). Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler. In M. d'Amorim (Ed.), *FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (pp. 197-207). (FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering). ACM. https://doi.org/10.1145/3663529.3663864

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler

Călin Georgescu
Delft University of Technology
Delft, The Netherlands
C.A.Georgescu@tudelft.nl

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

Pouria Derakhshanfar
JetBrains Research
Amsterdam, The Netherlands
Pouria.Derakhshanfar@jetbrains.com

Marat Akhin
JetBrains Research
Amsterdam, The Netherlands
Marat.Akhin@jetbrains.com

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

ABSTRACT

Compiler correctness is a cornerstone of reliable software development. However, systematic testing of compilers is infeasible, given the vast space of possible programs and the complexity of modern programming languages. In this context, differential testing offers a practical methodology as it addresses the oracle problem by comparing the output of alternative compilers given the same set of programs as input. In this paper, we investigate the effectiveness of differential testing in finding bugs within the Kotlin compilers developed at JetBrains. We propose a black-box generative approach that creates input programs for the K1 and K2 compilers. First, we build workable models of Kotlin semantic (semantic interface) and syntactic (enriched context-free grammar) language features, which are subsequently exploited to generate random code snippets. Second, we extend random sampling by introducing two genetic algorithms (GAs) that aim to generate more diverse input programs. Our case study shows that the proposed approach effectively detects bugs in K1 and K2; these bugs have been confirmed and (some) fixed by JetBrains developers. While we do not observe a significant difference w.r.t. the number of defects uncovered by the different search algorithms, random search and GAs are complementary as they find different categories of bugs. Finally, we provide insights into the relationships between the size, complexity, and fault detection capability of the generated input programs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Search-based software engineering**; **Compilers**; • **Theory of computation** → **Evolutionary algorithms**.

KEYWORDS

Code Generation, Compiler Fuzzing, Evolutionary Testing, Kotlin

This work is licensed under a Creative Commons Attribution 4.0 International License.

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663864

ACM Reference Format:
Călin Georgescu, Mitchell Olsthoorn, Pouria Derakhshanfar, Marat Akhin, and Annibale Panichella. 2024. Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3663529.3663864

1 INTRODUCTION

Compilers are an essential part of the software development ecosystem. They allow developers to write programs in a high-level language, that can be understood by a human, and convert these to a format that machines can understand. Kotlin is a popular upcoming high-level programming language that was developed by JetBrains in 2011, as an alternative to Java. Currently, Kotlin is used by 15.9M users and it is the main development language for Android.

Up until recently, Kotlin used the compiler that was introduced with the language when it was released, called K1. This compiler, however, is limited by its technical debt (*i.e.,* a consequence of software that expedited features over maintainability), making it harder to extend in the future. Therefore, with the release of Kotlin 2.0, JetBrains is introducing an improved new compiler, called K2. As K2 is not just a refactored version of K1, but a complete rewrite of its frontend based on a different architecture, it is important to make sure that the two versions behave similarly. The new frontend comprises 115k lines of Kotlin within the 565k core compiler code.

Software verification [9] is a specialized field of research that mathematically checks if a program complies with its requirements. However, it faces two main limitations when applied to compilers [14]: (1) they do not scale to the size and complexity of compilers, and (2) they require a model (*i.e.,* oracle) to determine if the output for a given input is correct. An alternative approach that circumvents the oracle problem [8] is differential testing [16, 29]. Differential testing takes two different versions of the same program, supplies these with identical inputs, and compares their outputs. Eventual discrepancies highlight bugs in one of the two versions.

Prior studies have successfully applied differential testing to find bugs in compilers for various programming languages, such as Java [15], C/C++ [27, 40], and JavaScript [22, 23]. However, existing approaches either require an initial set of programs (seeds) to mutate [38] or generate programs relying on a context-free grammar (CFG) specification (*e.g.,* [22, 23]). The latter approaches do not account for the rich semantic nuances often accompanying the grammar specifications. In the case of Kotlin, the CFG does not satisfy simple semantic rules without providing additional *context*. This increases the likelihood of generating invalid code if no additional semantic constraints are considered.

This paper presents a case study of black-box differential testing for the Kotlin compilers (K1 and K2 in particular) developed at JetBrains. We use a block-box approach since high coverage can be easily achieved with few simple programs according to the Kotlin developers. Our generative approach uses three stages to produce valid Kotlin code, addressing the challenges accompanying the above-mentioned Kotlin specifications.

First, our approach truncates the Kotlin specifications and replaces them with two context-aware models, namely the *enriched CFG* and the *semantic interface*. The former is an enriched version of the CFG augmented with additional constraints. The latter encodes the meaning and the semantic relations between different code segments. Second, our approach implements two categories of search algorithms that generate valid Kotlin programs based on context-aware models, namely random search (RS) and genetic algorithms (GAs). RS simply samples new Kotlin programs by randomly traversing the context-aware models. GAs, instead, evolve a pool (called *population*) of randomly generated yet valid Kotlin programs with the aim of promoting/generating a diverse set of programs over time. We investigate two flavors of GAs, namely (1) a single-objective variant that maximizes the program diversity and (2) a many-objective variant that considers program size as well. Finally, we perform differential testing that aims to identify inconsistencies in the output of the two compilers under test.

Our experimental results show that RS effectively detects differential bugs in the K1 and K2 compilers, namely *out-of-memory errors* and *resolution ambiguity*. Furthermore, GAs successfully detect further bugs related to *conflicting overloads*. While RS and GA are statistically equivalent *w.r.t.* the number of compiler bugs they identify, they are complementary as they uncover different categories of bugs. We have reported three categories of bugs found by our approach, and they have been verified and confirmed by JetBrains developers. Some of these bugs have already been resolved in more recent compiler releases, while others are planned to be resolved in future releases.

This paper makes the following contributions:

- A three-stage generative approach that intertwines CFG, programming language semantics, and meta-heuristic search for differential testing of compilers.
- A case study on the effectiveness of random search and meta-heuristics in testing Kotlin's K1 and K2 compilers.
- The discovery and analysis of new differential bugs reported to and confirmed by JetBrains developers.
- An in-depth analysis of the relationship between the characteristics of the generated programs (complexity and size) and their ability to uncover compiler bugs.
- A replication package with code [12] and data [11].

While our work focuses on Kotlin compilers, our approach can be applied to other compilers. Our study provides insights into the behavior of alternative search methods, which are valuable for compiler developers seeking to enhance the robustness and reliability of compilers across diverse programming languages.

2 BACKGROUND AND RELATED WORK

This section provides background information about Kotlin compilers and summarizes the related work in compiler testing and search-based software testing.

2.1 Kotlin

Kotlin[1] is a relatively new language that was developed by JetBrains in 2011. It is a general-purpose, high-level programming language that is statically typed and cross-platform. JetBrains developed Kotlin as an alternative to Java, but now it runs not only on the JVM, but also on JavaScript, Native, and even WebAssembly. Over the years, Kotlin has been gaining traction with the developer community. According to the latest annual report[2], Kotlin has 15.9M users and 90 of the global top 100 companies use Kotlin. In 2019, Google announced that Kotlin is the preferred language for Android.

The original compiler introduced with the language is called K1. This compiler has been updated with new features throughout the years, while at the same time accumulating technical and architectural debt. Recently, JetBrains has been working on a new frontend for the compiler, called K2[3]. K2 aims to address the existing debt, speed up the development of new language features, improve the performance of the compiler, and fix bugs and inconsistencies in the compiler behavior. Since the K2 compiler is not an iteration of the old compiler but a complete rewrite of its frontend component, it is important to ensure the two compilers behave similarly.

2.2 Compiler Testing

In recent decades, researchers and practitioners have invested tremendous resources to improve compiler testing through automation [10, 14]. Modern approaches make use of Differential Testing (DT) to heuristically assess different compilers of the same programming language. They generate code snippets, which are used as the input for each compiler version with the goal of uncovering differences in behavior [29]. Any difference in the compiler's outcome (*i.e.,* crashes, non-compiling snippets, or errors) highlights implementation errors that may affect real-world applications. By making use of DT, we can circumvent the *test oracle problem* [8].

For this study, we make the distinction between *test program generation* and *program mutation* approaches as classified by Chen et al. [14]. The former generates programs from scratch, without any external seed, while the latter mutates existing programs to generate new variations. Test program generation approaches also differ in their utilization of the language grammar. *Grammar-directed* approaches solely rely on the grammar to generate novel code snippets, whereas *grammar-aided* approaches heuristically exploit the grammar, which is often enriched with semantically rich context.

Purdom [35] devised one of the first algorithms aimed at testing the correctness of Context-Free Grammar (CFG) parsers. They use a grammar-directed approach that applies iterative rewriting rules beginning with a starting symbol, to eventually exhaust the entire

[1] https://kotlinlang.org/
[2] https://www.jetbrains.com/lp/annualreport-2023/
[3] https://blog.jetbrains.com/kotlin/2023/02/k2-kotlin-2-0/

grammar specification. Yang et al. [40] propose CSMITH, a grammar-aided test program generation tool aimed at finding bugs in C compilers using DT. CSMITH fundamentally differs from Purdom's approach in that it follows a generation pattern that primarily focuses on the semantic properties of C rather than its grammar.

Livinskii et al. [27] introduce YARPGEN, a C/C++ program generation tool that aims to increase the expressiveness and diversity of test programs. YARPGEN proceeds in a 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. Han et al. [22] introduce the notion of *semantics-aware assembly*, which they implement in the CODEALCHEMIST tool, aimed at testing the JavaScript engine. 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* [23]. A code brick consists of a valid JavaScript abstract syntax tree (AST) that is additionally annotated with an *assembly constraint*.

Specification fuzzing approaches envelop a third category of code generation techniques that generally favor language-agnostic declarative formulations over system-specific approaches. The Input Specification Language (ISLA) [37] is one such approach that allows users to annotate a grammar with semantic, context-dependent constraints. In this approach, users can introduce annotations that match the nuances of real programming languages and thus circumvent implementing narrow-scoped fuzzers. ISLA uses a *solver* to iteratively expand the constrained grammar specification, which allows it to generate code that is both semantically and syntactically aligned. The Language Specification Language (LALA) [25] allows for declarative descriptions of attribute grammars [24] as a means of introducing semantics to a context-free language. The LALA framework translates a given specification into Java classes that are the input to the fuzzing process. At runtime, the fuzzer instantiates ASTs that conform to the attribute properties, while simultaneously checking for *fail patterns*.

Compared to the related work, we enhance the differential testing process by introducing heuristic-based guidance targeting the Kotlin language. We use similar semantic modelling techniques as CSMITH [40] and YARPGEN [27], but we allow users of our tool to customize the root environment to a much greater extent. Our implementation enables users to include arbitrary Kotlin code that is automatically parsed and integrated into the semantic context. During sampling, we repeatedly query the context and use its constituents (*i.e.,* user-provided classes) to generate new code. Similar to CODEALCHEMIST [22] and LANGFUZZ [23], we structurally decompose code. However, in contrast to those approaches, we only employ decomposition as a means of adding variation to our code during generation rather than extracting information from existing code bases. We also decided against using ISLA [37] and LALA [25] because of the significant effort required to encode the rich semantics of Kotlin within their respective frameworks. Additionally, the fuzzers of both approaches heavily rely on an enumeration of random expansions, whereas our approach enables a more sophisticated heuristic search. Finally, Stepanov et al. [38] also targets Kotlin with its mutation-driven method, which, unlike our approach, relies on existing code as a starting point for the fuzzier.

2.3 Search-Based Software Testing

Search-based Software Testing (SBST) is a broad umbrella of approaches and tools aimed at automating the process of generating test cases [30]. The automation is achieved by utilizing optimization algorithms (e.g., genetic algorithms) that iteratively evolve a set of randomly generated test cases toward optimizing given testing criteria (e.g., branch coverage) [4]. To this aim, meta-heuristics leverage a fitness function (or objective) to evaluate how closely the test execution aligns with those goals. Depending on which type of information the fitness function relies on, SBST techniques can be classified in *white-box* and *black-box*. The former techniques use the internal information (e.g., coverage data) of the program under test to assess the "fitness" of the generated tests [30]. The latter techniques do not require access to the source code (bytecode) but rely on external information [18], such as input diversity [2].

Previous research has demonstrated the effectiveness of SBST across various testing levels, including unit [20, 31, 32], integration [19], system levels [6, 18], and concurrency testing [39]. SBST approaches have shown to be particularly promising, outperforming random testing, in achieving extensive coverage [13, 33], detecting defects [19, 21], or testing cyber-physical systems [1].

In this work, we aim to investigate the effectiveness of genetic algorithms and random search when applied to the detection of bugs in Kotlin compilers using differential testing. As we illustrate in the next section, our approach is black-box, thus focusing on the input and output of the compilers under test.

3 APPROACH

Our approach aims to automatically generate 100% valid code that uncovers defects in the Kotlin compilers. To achieve this goal, we designed a multi-stage approach comprising three phases, as depicted in Fig. 1. Phase I consists of building workable models of Kotlin semantic and syntactic language features, which are subsequently exploited to generate (*sample*) random code snippets. We elaborate on this step in Section 3.1. Section 3.2 covers Phase II, in which we provide *guidance* to the random sampling process by introducing search objectives and utilizing Evolutionary Algorithms (EAs).

Phase III performs *differential testing* (DT) on K1 and K2 using the output of Phase II as compiler input. In this final step, we differentiate between three different cases. If both compilers display the same behavior (*i.e.,* they both successfully compile the code), no defects have been uncovered. If the two compilers give different verdicts, (*i.e.,* one compiles the code while the other raises an error, as shown in Fig. 1), then one of the versions under test is erroneous. Finally, if either of the compilers *crashes*, we individually analyze the cause of the crash through the compiler's logging mechanism.

3.1 Random Code Generation

In this subsection, we delve into the models that our approach uses to encompass the structure (syntax) and the meaning (semantics) of programs. These models serve as the basis for generating valid Kotlin code and form the foundation of the EAs in Phase II.

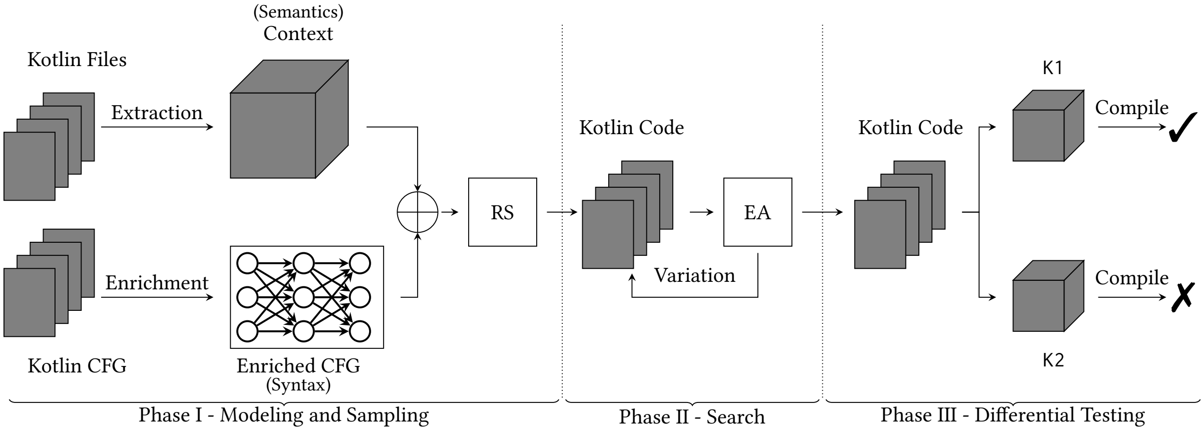


Figure 1: Overview of our approach.

3.1.1 *Syntactic Interface.*

Kotlin is a rich programming language, the structure of which is defined in terms of a formal Context-Free Grammar (CFG) [3]. The language developers made a full grammar specification available in the popular ANTLR4 [34] parser-generator framework. Though crucial for verifying the structural soundness of Kotlin code, the CFG poses several practical challenges.

With over 480 symbols and 700+ productions, the Kotlin CFG is a complex aggregation of structural relations. However, the inherent lack of context of the grammar gives rise to several obstacles that make straightforward generation algorithms impractical. For instance, classic grammar-based fuzzers such as the one introduced by Purdom [35] do not account for the rich semantic nuances often accompanying CFG specifications. In the case of Kotlin, the CFG cannot satisfy even simple semantic rules such as *a variable name must be defined before assignment.* When paired with the significant complexity of the specification, this causes any attempt to sample the grammar independently to result in invalid code with an overwhelming likelihood, due to the accumulation of semantic constraints that are unaccounted for in the grammar.

To address this limitation, we implemented two key operations that modify the standard Kotlin CFG. First, we selectively *truncate* the grammar at points in the specification where significant semantic nuance occurs. This operation preserves the *shape* in the grammar (*i.e.,* no new symbols or rules are created, but transitions between symbols are retained), while drastically reducing its complexity and increasing the likelihood of generating valid code. Second, we endow each grammar rule with a `sample` property that overrides the standard specification of the symbol in the CFG.

We only perform this latter operation on semantically rich symbols that remain in the CFG's composition after the truncation step. Simpler nodes (such as *cardinality* or *optionality*) do not require specific `sample` implementations and instead fall back on their default specification. The fallback mechanism guarantees that grammar-abiding options are always available, even if individual `sample` rules are not implemented. This, in turn, enables a trade-off between specificity and implementation complexity.

The purpose of the `sample` property is to ensure that grammar traversal algorithms additionally account for constraints not included in the grammar specification. Together, the two operations simplify and augment the Kotlin CFG in a process we call *enrichment* (Figure 1, Phase I). The output of this transformation is an annotated Directed Acyclic Graph (DAG), where nodes correspond to symbols and edges pertain to rules in the original grammar.

3.1.2 *Semantic Interface.*

To effectively utilize the syntactic interface of the enriched CFG, our method requires a corresponding semantic counterpart that encodes the meaning and semantic relation between different code segments. The abstraction that encompasses these concepts is hereafter referred to as *semantic context.*

Our implementation of the semantic context fulfills three key requirements within the fuzzer. Firstly, it furnishes the enriched CFG with a mechanism for querying useful information from the available code. This allows traversal algorithms to discern which productions are feasible. Secondly, the context actively tracks changes to the semantics of a program as it is generated. This *mutability* enables an iterative increase in the complexity of the sampled code, as previously generated code (*i.e.,* variables) can appear again in later lines. Finally, we construct the semantic context through an *extraction* process, that parses and extracts information from provided Kotlin files. This process ensures the versatility of our approach, as users can provide arbitrary files as input to constrain (or broaden) the generative process.

The context data structure tracks three traits of Kotlin programs. First, it maintains a set of visible *callables* within reachable code. We use the term *callables* in a similar fashion to [38], encompassing all visible functions, properties, constructors, variables, constants, and primitives provided to the fuzzer. This data is stored in a λ-calculus-like representation that captures their properties. The context additionally accounts for the *type hierarchy* of its programs and the constraints related to parameterized types. Lastly, various semantic constraints are embedded within the queries that the context supports, including both universal Kotlin rules and context-sensitive restrictions (*i.e., only sample types that at least one callable in the context can return*).

3.1.3 *Random Sampling/Search.*

The final step in Phase I combines the enriched CFG and the semantic context in a straightforward manner that produces random semantically valid Kotlin code. To do so, we follow the enriched grammar structure, randomly selecting

```
Algorithm 1: Random Sampling
Input :CFG $\mathcal{N}$, Context $C$, Time budget $s$
1 $A \leftarrow \emptyset$
2 while $\neg$ TimeElapsed($s$) do
3     $c \leftarrow$ Clone($C$)
4     $b \leftarrow$ sample($\mathcal{N}$, $c$)
5     $A \leftarrow A \cup \{b\}$
6 return $A$
```

transitions between nodes, while simultaneously querying the attached semantic context. We refer to this procedure interchangeably as Random Sampling or Random Search (RS).

Algorithm 1 outlines RS. The algorithm begins by initializing an archive A (line 1) that tracks all generated code. The main loop (lines 2-5) proceeds by first creating a clone of the root context (line 3) before querying the `sample` property of the given target CFG node (line 4). The purpose of the context clone is to ensure that changes performed in a sampling round do not propagate to later, independent samples. RS then adds the obtained sample to the archive (line 5) and returns the collected samples (line 6).

3.2 Evolutionary Fuzzing

Though effective at drawing random samples of valid Kotlin code, Algorithm 1 lacks direction. Specifically, it lacks a mechanism that would allow sampled code to undergo structural changes that increase the likelihood of uncovering compiler defects. To introduce such a mechanism, we rely on EAs as a framework of performing iterative changes on randomly sampled Kotlin code in Phase II of our approach. We first address the genetic representation of code, before describing the variation operators and finally, formulating the overarching EA in detail.

3.2.1 Solution Encoding. The individuals that make up the population of the EA consist of independently valid pieces of code, hereafter referred to as *blocks*. The goal of this denomination is to isolate pieces of code that are both syntactically and semantically *self-contained* (*i.e.,* that have no external dependencies). Each block B is comprised of an ordered sequence of *snippets* $[s_1, ..., s_n]$, representing isolated pieces of Kotlin code with inner scopes (*i.e.,* classes, functions). We order snippets based on the topology of their dependencies to more efficiently detect and extract self-contained structures. Each snippet s is itself a 4-tuple $\langle N, \Lambda, D, F \rangle$ with N the name of the snippet, Λ the λ-calculus formatted metadata which encodes the snippet's input-output behavior, D a list of snippets s depends on, and F an ordered list of *fragments*, the lowest denomination of our representation. Fragments are simply text-encoded pieces of Kotlin code that generally correspond to single lines of code. These generally include statements and expressions.

This hierarchical representation provides two key advantages. First, it captures the structural composition of the underlying Kotlin program with increasing levels of granularity. This enables the design of variation operators that perform complex alterations to the structure of sampled programs, which sampling alone cannot. Second, it allows for automated dependency and conflict analysis. Since snippets include metadata regarding their signatures and dependencies, it is straightforward to reason about the conflicts that may arise when including two snippets in the same block or about the requirements that removing a snippet may entail. Fragments include no such metadata, reducing overhead. The dependencies of a snippet are equivalent to the cumulative dependencies of all of the snippet's fragments. In our formulation, fragments are bound to a single snippet's scope and are not subject to any further variation during the search. Because of this, reasoning about the snippet's dependencies suffices to ensure the validity of the code.

3.2.2 Variation Operators. Before describing the variation operators of our EA, we first establish the notion of a *self-contained partition* of a block. Given a block $B = [s_1, ..., s_n]$, we can obtain a partition $B' = [s'_1, ..., s'_m]$ starting from a snippet $s_i = s'_1 \in B$ by including (i) s_i, (ii) all snippets that s_i directly or indirectly depends on, (iii) all snippets that directly or indirectly depend on s_i, and (iv) recursively performing selections (ii) and (iii). The new partition B' is a block, as it has no external dependencies.

Using this notion, we define three mutation operators that perform changes on arbitrary blocks B. The *removal* operator first selects a random snippet $s_r \in B$ and removes its corresponding self-contained partition $B_{s_r} \subseteq B$. The *context-free addition operator* samples a new block B_{cfa} from the root context and appends its snippets to B. Finally, the *context-aware addition operator* first merges the context of B with the root context before performing a sample operation that results in a new block B_{caa}. The mutation consists of appending the snippets of B_{caa} to B.

We additionally implement a recombination operator that takes as input two parent blocks B_{p_1} and B_{p_2} and swaps two self-contained partitions $B_{x_1} \subseteq B_{p_1}$ and $B_{x_2} \subseteq B_{p_2}$ to obtain two new offspring blocks $B_{o_1} = (B_{p_1} - B_{x_1}) \cup B_{x_2}$ and $B_{o_2} = (B_{p_2} - B_{x_2}) \cup B_{x_1}$. We perform conflict analysis prior to recombination such that we only select pairs of blocks with no conflicting signatures.

3.2.3 Heuristics. We bring together the tools developed in this section in two formulations of genetic algorithms (GAs). Both algorithms attempt to optimize the *diversity* of sampled code under the hypothesis that structurally diverse code is more likely to stress different components of the compiler and thus uncover more defects.

Each algorithm implements a different measure of diversity through the fitness function involved in the selection mechanism. To map individuals to a numerical fitness space, we first establish a notion of *similarity* between blocks. For any two blocks B_1 and B_2 we define a mapping $m : \mathcal{B} \rightarrow \mathbb{N}^k$ that transforms blocks of Kotlin code from the abstract space $\mathcal{B}$ to k-dimensional natural number vectors. Each position of the vector represents the number of times a particular Kotlin language feature (*i.e.,* `if` expressions, functions) appears in the input block, capturing a crude estimation of the structural composition of the underlying program. Using this mapping, any common measure of distance $d : \mathbb{N}^k \times \mathbb{N}^k \rightarrow \mathbb{R}$ (*i.e.,* euclidean norm) can be used to determine the similarity between two blocks. Using these notions, we define the *population-wide dissimilarity* of a block in B in population P according to Equation (1):

$$dis(B, P) = \min_{B_i \in P - \{B\}} \{d\left(m(B), m(B_i)\right)\} \tag{1}$$

Intuitively, Equation (1) measures the distance between B and its most similar individual in P. We use this formula to construct the population-wide diversity fitness function in Equation (2). We

Algorithm 2: Single-Objective Diversity-based GA

```
Input :Population size $n$, CFG $\mathcal{N}$, Context $\mathcal{C}$, Time budget $s$
1 $t \leftarrow 1$
2 $P_1, P^* \leftarrow$ InitializeAndEvaluatePopulation($n, \mathcal{N}, \mathcal{C}$)
3 while $\neg$ TimeElapsed($s$) do
4     $O_t \leftarrow$ CreateAndEvaluateOffspring($P_t$)
5     $P_{t+1} \leftarrow$ SelectIndividuals($P_t, O_t, f^{(SO)}_{\texttt{DIV}}$)
6     if $\sum_{b \in P_{t+1}} f^{(SO)}_{\texttt{DIV}}(b, P_{t+1}) > \sum_{b \in P^*} f^{(SO)}_{\texttt{DIV}}(b, P^*)$ then
7         $P^* \leftarrow P_{t+1}$
8     $t \leftarrow t + 1$
9 return $P^*$
```

define $f^{(SO)}_{\texttt{DIV}}$ as a single-objective (SO) fitness function, which seeks to minimize a value proportional to the inverse of Equation (1), effectively maximizing dissimilarity:

$$\min_{B \in \mathcal{B}} f^{(SO)}_{\texttt{DIV}}(B, P) = \frac{1}{1 + dis(B, P)} \tag{2}$$

Algorithm 2 describes the Single Objective Diversity Genetic Algorithm (SODGA) that optimizes $f^{(SO)}_{\texttt{DIV}}$. The algorithm begins by initializing a counter and instantiating the population by means of RS (lines 1, 2). In addition to the standard population, SODGA also tracks the most diverse population encountered during the run in the variable P^*. Next, the algorithm proceeds in a loop (lines 3-8) where the variation operators give rise to offspring (line 4) before triaging the population through selection (line 5). In each generation, the most diverse population is updated if a better (more diverse) set of individuals emerges (lines 6, 7). After the time budget has been exhausted, the algorithm returns the population P^*.

The fitness function of SODGA is fundamentally dependent on each generation's population, which prevents it from ever converging to a stable solution set. While this is desirable for lengthy fuzzing campaigns, we propose a second algorithm that aims to provide a stable converging behavior. In contrast to SODGA, we design this alternative as a many-objective (MO) approach that seeks to construct a stable archive of diverse yet small Kotlin programs. Equation (3) describes the MO fitness function that attempts to simultaneously minimize the size of generated programs (f_{sz}) and maximize the number of each language features present in them (f_{l_i} from an abstract set of language features $\mathbb{L}$):

$$\max_{B \in \mathcal{B}} f^{(MO)}_{\texttt{DIV}}(B) = \left\{-f_{sz}(B),\ f_{l_1}(B),\ \dots,\ f_{l_n}(B) \mid l_i \in \mathbb{L}\right\} \tag{3}$$

In contrast to the SO approach, the size objective in MODGA favors small programs, which helps isolate uncovered defects.

Algorithm 3 describes the implementation of the MO Diversity GA (MODGA). It first initializes an elitist archive (line 1) before sampling a random initial population (line 2). The main loop proceeds in standard GA fashion, with an additional archive update step (line 4) that processes newly generated files. Selection (line 6) is carried out by means of Pareto-domination counting [28]. MODGA returns the elements of the archive at the end of the run.

4 EMPIRICAL STUDY

Our empirical study aims to evaluate RS, SODGA, and MODGA *w.r.t.* their capability of uncovering bugs in the Kotlin compiler. We begin by examining the impact of internal parameter values on each algorithm to gain insights into their behavior. Specifically, we analyze how the notion of *expression simplicity* influences block generation in RS. Additionally, we investigate the role of the chosen distance metric in Equation (1) on the performance of SODGA. Furthermore, we conduct a comparative analysis of the performance of different algorithms in terms of the number of defects they find during each run. We summarize these goals within the following research questions:

Algorithm 3: Many-Objective Diversity-based GA

```
Input :Population size $n$, CFG $\mathcal{N}$, Context $\mathcal{C}$, Time budget $s$
1 $A \leftarrow$ InitializeElitistArchive($f^{(MO)}_{\texttt{DIV}}$); $t \leftarrow 1$
2 $P_1 \leftarrow$ InitializeAndEvaluatePopulation($n, \mathcal{N}, \mathcal{C}$)
3 while $\neg$ TimeElapsed($s_t$) do
4     $A \leftarrow$ ProcessNewArchiveEntries($P_t, f^{(MO)}_{\texttt{DIV}}$)
5     $O_t \leftarrow$ CreateAndEvaluateOffspring($P_t$)
6     $P_{t+1} \leftarrow$ DominationSelection($P_t, O_t, f^{(MO)}_{\texttt{DIV}}$);
7     $t \leftarrow t + 1$
8 return $A$
```

Table 1: Overview of evaluated algorithms.

Name	Fitness	Objectives	Selection	Parameter
RS	-	-	-	Simplicity Bias (p_b)
SODGA	$f^{(SO)}_{\texttt{DIV}}$	1	Tournament	Distance (l^2 or l^∞)
MODGA	$f^{(MO)}_{\texttt{DIV}}$	$\mid \mathbb{L} \mid +1 = 7$	Dom. Rank	-

RQ1. *How does expression simplicity impact the properties of files generated by RS?*

RQ2. *How does the distance measure influence the properties of files generated by SODGA?*

RQ3. *How effective are RS, SODGA, and MODGA in terms of uncovering bugs in the Kotlin compiler?*

4.1 Framework

We implemented the methods described in this paper in an open-source repository[4] that includes additional heuristics not detailed in this study. In addition to the fuzzer implementation, it contains an extensive framework for thorough customization and analysis of heuristics. Furthermore, it provides utilities for replicating the results of this study [11, 12]. This includes functionality that automates DT and defect classification, as well as the aggregation of data related to the fuzzer's performance.

4.2 Configurations

We assess the implementation of three distinct algorithms: RS, SODGA, and MODGA. All implementations operate on a subset of the Kotlin CFG that includes five language features: function declarations[5], statements[6], assignments[7], and four types of expressions[8]. This means the output of Equation (3) is a seven-dimensional vector.

[4] https://github.com/ciselab/kotlin-compiler-fuzzer
[5] https://kotlinlang.org/spec/declarations.html#declarations
[6] https://kotlinlang.org/spec/statements.html#statements
[7] https://kotlinlang.org/spec/statements.html#assignments
[8] https://kotlinlang.org/spec/expressions.html#expressions

The first entry of the vector denotes the program size, and the remaining six entries count the frequency (number) of each language feature in the corresponding block.

Table 1 lays out the overview of the configuration for each evaluated algorithm. To answer RQ1, we introduce the *simplicity bias* parameter that governs the complexity of sampled blocks. This parameter operates within the `sample` property of CFG nodes for the expressions. The bias influences the probability of sampling *simple* expressions (*i.e.,* function calls and property accesses) as opposed to more complex counterparts (*i.e.,* `if`-expressions). The simplicity bias is expressed as a number $p_b \in [0, 1]$, where p_b indicates the probability of sampling *simple* expressions while $1 - p_b$ is the probability of sampling complex ones. Similarly, we experiment with two distance measure implementations for the d function of Equation (1) that correspond to the l^2 (or Euclidean) and l^∞ norms:

$$l^2(m(B_1), m(B_2) = \sum_{1 \le i \le |\mathbb{L}|+1} \sqrt{(m(B_1)^{(i)} - m(B_2)^{(i)})} \quad (4)$$

$$l^\infty(m(B_1), m(B_2)) = \max_{1 \le i \le |\mathbb{L}|+1} \mid m(B_1)^{(i)} - m(B_2)^{(i)} \mid \quad (5)$$

For SODGA and MODGA parameters, we turn to established literature values to determine the population size and further selection details. However, applications of evolutionary fuzzing to compiler testing are scarce, and established evolutionary fuzzing tools like VUzzer [36] and V-Fuzz [26] make no recommendation in these regards. As such, we turn to standard values used in search-based testing literature, particularly of EvoSuite, which uses a population size of 50 and a *tournament selection* with tournament size of 10 [20, 32]. We apply the mutation and recombination (crossover) operators described in Section 3.2.2.

Past research by Arcuri and Fraser [7] has shown that default values can provide solid performance in a broad set of scenarios in software testing. Though these findings provide no guarantees for the task of compiler testing in particular, we believe these are sensible starting points that circumvent the expensive requirements of parameter tuning of all proposed algorithms.

4.3 Experimental Protocol

To understand how the simplicity bias influences the nature of generated files, we perform several runs of RS with different parameter settings. We experiment with simplicity bias values between 0.4 and 0.6, as we empirically found that values outside this range either lead to blocks too large to scale effectively, or too small to build sufficiently complex programs. We collect information regarding the number of Kotlin programs generated, their size, and the types of compiler crashes each simplicity bias reveals.

To answer RQ2, we run both SODGA and MODGA with l^2 and l^∞ norms and we analyze the properties of the generated programs in a manner analogous to that done for RQ1. Finally, we answer RQ3 by considering the effectiveness and efficiency of RS, and the GAs with best-performing parameter values (for diversity) based on the results of the previous experiments (RQs). We measure effectiveness in terms of the number of differential bugs uncovered by each algorithm at the end of each search/fuzz run. For efficiency, we collect the number of bugs uncovered over time (every 180 seconds) and compute the Area-Under-Curve (AUC) of the resulting bugs-over-time graph. For RQ3, we perform statistical analysis using the Wilcoxon [17] signed-rank test to determine whether the underpinning distributions are significantly different.

In total, we ran five instances of RS to answer RQ1, three instances of SODGA and MODGA to address RQ2, and ten runs of RS and SODGA to answer RQ3. Each run lasts 90 minutes for a total of 28 total fuzzing sessions amounting to 42 hours of fuzzer runtime. In each run, we store *snapshots* of the population of SODGA and of the archive of MODGA every 180 seconds. We carry out all DT procedures on Kotlin version `1.8.20-RC-release-288`, which contains both K1 and K2 releases in the same package[9].

All experiments and runs were executed in isolated containers using *Docker* and performed on the same machine using an AMD Ryzen 7 5800H with 16 GB of RAM. Each Kotlin compiler uses the default 4 GB of heap memory.

5 RESULTS AND ANALYSIS

5.1 Effects of Expression Simplicity

To investigate the impact of the simplicity bias, we analyze the size and number of files generated by the Random Search (RS) algorithm when varying the simplicity bias p_b. Figure 2 depicts the relationship between (a) the average size of the Kotlin programs generated through random sampling and (b) the number of files the algorithm outputs in a 90-minute interval.

The average size of a file decreases from 5,084 characters for a bias of 0.40 to 2,205 for a bias of 0.45 and 1,545 for a bias of 0.50. The rate of change diminishes for bias values 0.55 and 0.6, with average sizes of 1,132 and 941 characters, respectively. This rate of change follows an exponential trend according to the Anderson-Darling test of goodness-of-fit (p-value<0.001) [5].

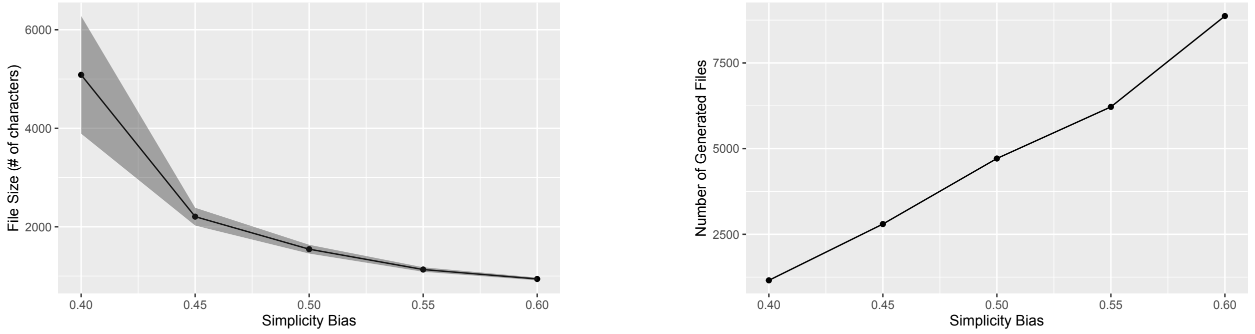
Conversely, the number of generated files (programs) increases from 1,157 for the lowest value of bias (0.40) to 8,869 for the highest (0.60). Values corresponding to biases between the two extremes follow the same trend, resulting in 2,799, 4,712, and 6,218 files generated for simplicity biases of 0.45, 0.50, and 0.55, respectively.

Next, we analyze the relationship between the simplicity bias and the effectiveness of random sampling in identifying differential bugs between the two Kotlin compilers, namely K1and K2. Figure 3 presents a visual representation of the number of defects discovered through differential testing of the generated files, corresponding to each tested simplicity bias value. A crucial finding from these results is that *all* defects uncovered by RS are of a single type: out-of-memory (OOM) errors, specifically in the K1 compiler.

All OOM errors encountered in the five experimental runs are triggered by files with 10,000+ characters. Their distribution among the simplicity bias values aligns with the distribution of file sizes that each value entails. RS with a simplicity bias of 0.40 generates 15 OOM-inducing files, while higher values of 0.45 and 0.50 produce three such instances. Simplicity bias values exceeding 0.50 do not generate such files, due to their narrower file size distribution.

Summary RQ1: Lower simplicity bias values cause RS to generate larger and fewer files. Bias values of 0.50 or lower occasionally generate files of over 10,000 characters, which often trigger OOM errors in K1, but not in K2.

[9] https://kotlinlang.org/docs/whatsnew1820.html



(a) Mean size of a block generated by RS as a function of simplicity bias.

(b) Number of blocks generated by RS per 90 minutes as a function of simplicity bias.

Figure 2: Comparison of the number of files generated by RS and their size.

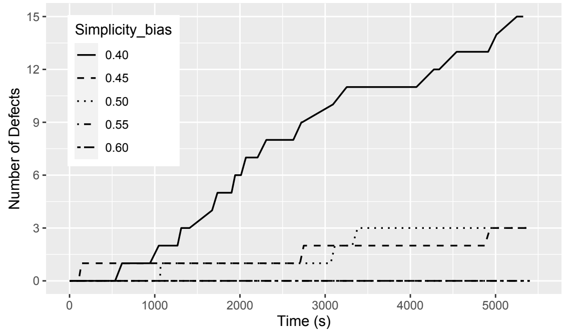


Figure 3: The number of defects uncovered by RS with varying simplicity bias values.

5.2 Effects of Diversity Interpretation

We begin by analyzing the properties of the Kotlin programs generated and collected in each snapshot of SODGA and MODGA with l^2 and l^∞ norms. Figure 4 (a) provides the visualization of the evolution of the average generated file size over time. For both variations of SODGA, the mean file size varies between 1,000 and 3,500 characters, which are well under the threshold of 10,000 characters that may trigger OOM errors.

The fluctuations stem from the influence of the population-sensitive fitness function. As the population tends towards larger programs, the diversity-based fitness function (and the selection operator) promotes smaller programs as they will be more diverse (i.e., more distant regions of the search space). As smaller programs take over the population, the trend reverses, and larger programs again introduce more diversity. This pattern holds true for both the l^2 and l^∞ norms. However, the latter norm induces more substantial shifts in program sizes due to its more rigorous measure of (dis)similarity. Consequently, the l^∞ variant yields both the largest (3,500) and smallest (900) average program sizes.

In contrast, the elitist archive of MODGA retains files that are far smaller than the population of its single-objective counterparts. Following initial fluctuations in the first six snapshots, the average program size in the population of MODGA stabilizes at around 300 characters. This aligns with the composition of the archive itself, which also converges to a static set of 70 solutions by the sixth snapshot. Subsequent changes over time are comparatively minimal, with the archive expanding by only three additional entries.

Figure 4 (b) depicts the program size distribution generated by SODGA and MODGA. The means of the distributions of the two SODGA variations are comparable: the l^∞ variant produces files that are, on average, only 1.50% (1,574 characters) smaller than l^2 (1,598 characters). However, the two distributions substantially differ in their third and fourth interquartile, with the l^∞ norm generating more outliers. MODGA produces files that are much smaller and less diverse than either SODGA. This pattern can be attributed to two primary factors: (1) the size component notably reduces the archive's overall size; (2) the archive has a tendency to retain only a few (less than 10) very large files, which independently dominate substantial portions of the search space.

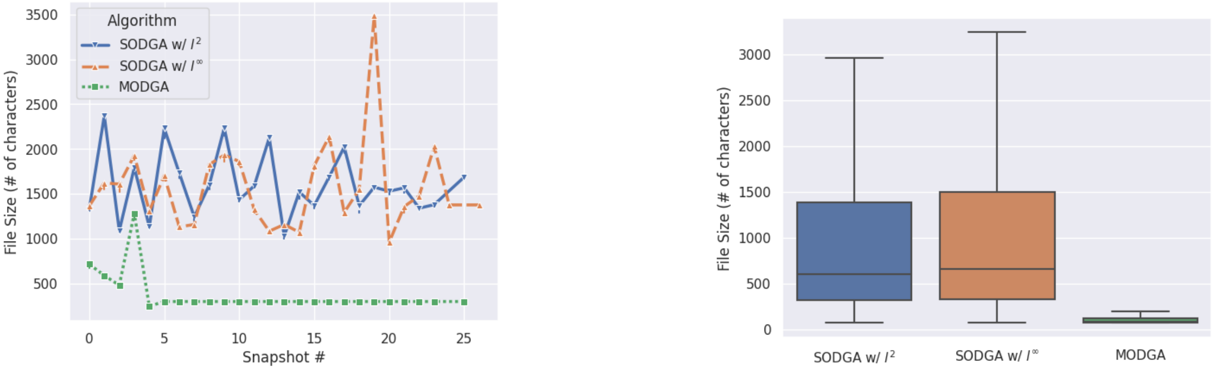
In terms of bug finding, the genetic algorithms allowed to discover two bugs. SODGA equipped with the l^∞ norm and MODGA both independently uncover one defect that causes K1 to raise an error, while K2 successfully compiles the generated code. Both instances of the defect emerge as a consequence of the genetic recombination operator and could not have been generated otherwise by RS. We discuss this further in Section 5.4.

Summary RQ2: SODGA behaves similarly when using l^2 and l^∞ norms, with the latter displaying a broader program distribution. MODGA's archive stagnates after a few iterations and retains smaller files. All GAs uncover defects that RS could not discover.

5.3 Comparative Analysis of the Detected Bugs

To gain additional insights into the comparative performance of the two classes of algorithms, we compare the defect detection capabilities of RS and SODGA. To this end, we performed 10 fuzzing sessions with RS and SODGA w/ l^2. We selected this variant of single-objective GA since it generates fewer outlier files than its l^∞ counterpart, and does not suffer from possibly premature convergence like MODGA. Both algorithms are run with a simplicity bias of 0.5 based on the results of RQ1.

In total, RS uncovers more defects (12 unique bugs) than SODGA (9 unique bugs). However, the Wilcoxon test revealed no statistically



(a) Mean size of blocks generated by SODGA and MODGA.

(b) Block size distribution of SODGA and MODGA.

Figure 4: Block size distribution of SODGA and MODGA.

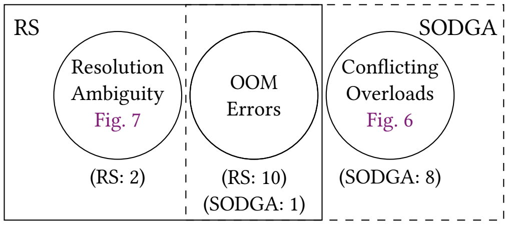


Figure 5: Overview of uncovered defects.

significant difference *w.r.t.* the number of detected bugs between RS and SODGA ($p = 0.459$). The RS runs reveal one novel defect that causes K2 to not compile a generated Kotlin program when K1 does. SODGA additionally finds 1 OOM error and eight defects of the same category as the ones addressed in Section 5.2, but no novel defects emerge. RS achieves, on average, a mean AUC of 0.719 (*i.e.,* bug detection over time), which is greater than SODGA's 0.446. However, the pairwise statistical analysis suggests these differences are not statistically significant ($p = 0.322$). In other words, SODGA and RS show similar efficiency in finding bugs over time. Figure 5 provides an overview of the categories of bugs each algorithm uncovers. Of the 21 bugs found in total, 11 are OOM errors that RS finds more frequently (10) than SODGA (1). RS additionally uncovers two instances of a resolution ambiguity defect, and SODGA distinctly finds eight instances of conflicting overload errors.

Summary RQ3: RS and SODGA with l^2 norm are statistically equivalent *w.r.t.* the number of bugs they uncover and their efficiency (bugs detected over time). However, the two algorithms are complementary as they find distinct categories of defects.

5.4 Bugs Confirmed by JetBrains Developers

Throughout the experiments carried out in this study, we encountered several dozen instances of erroneous behavior in the Kotlin compiler. After individual analysis and consultation with the Kotlin compiler developer team, we divided these instances into three distinct categories. This section briefly analyzes these categories, their impact on the Kotlin ecosystem, and the fuzzer components responsible for their detection.

```
fun main() {
    fun p() : Char { return 'c' }
    fun p() : Float { return 13.0f }
}
```

Figure 6: K2 false negative conflicting overloads.

5.4.1 OOM Errors. OOM errors occurring only in the K1 compiler are common among files that exceed 10,000 characters in length. The fact that such files do not cause equivalent errors in K2 makes them less important for the scope of this study, as they showcase a measurable improvement in performance, rather than an issue that requires developer attention. As a result, we did not report any of these issues to the developer team, instead focusing on bugs that affect K2. We occasionally encountered code that either (i) triggered OOM errors in both K1 and K2, or (ii) only triggered OOM errors in K2. Compiler developers confirmed several such instances for the current release of Kotlin, but consider them of minor importance or even acceptable.

5.4.2 K2 Nested Functions. Figure 6 contains a code snippet that K2 compiles without warnings, while K1 throws a *conflicting overloads* error. The latter is the intended behavior. The two p() functions cause a resolution conflict that the compiler is meant to warn about. After experimentation with this instance, we observed that K2 always resolves a p() function call to the first definition of the function, irrespective of the return type or the number of re-declarations. Notably, this problem only occurs when the definitions of p() are both *nested* inside a higher-scope function. The Kotlin compiler developers confirmed the existence of this bug and assigned it a medium priority. They plan to fix this error in Kotlin 2.0.

We uncovered several independent instances of this bug, all generated through GAs. The sampling process that RS depends on contains a check that prevents the generation of functions with the same name, which is tracked through the shared context. However, this constraint is relaxed during recombination, allowing for such scenarios to emerge during crossover occasionally.

5.4.3 K2 Concurrent Modification Exception. Figure 7 shows an error that affects the ConcurrentModificationException class

```
var J_: ConcurrentModificationException =
    ConcurrentModificationException()
J_ = J_ ?: ConcurrentModificationException(
    ConcurrentModificationException(J_))
```

Figure 7: K2 overload resolution ambiguity.

of the Kotlin standard library. The K2 compiler reports an overload resolution ambiguity error stemming from the expression in line 4. K1 compiles the code without error. The Kotlin developers verified the occurrence of this bug in several recent compiler releases. The compiler team traced the bug to the resolution and inference components of the compiler frontend, and, after careful consideration, decided this change is expected. Though any permutation of algorithm and configuration described in this study is in principle capable of generating this, it was RS that initially uncovered it. Finding such bugs depends on covering the input of the fuzzers' context, which RS is more effective at since it does not incur additional overhead from operations on higher-level language structures.

6 THREATS TO VALIDITY

Threats to *construct validity* stem from the connection between the practical measures employed to quantify theoretical aspects of the study. In this regard, we use standard DT procedures to quantify compiler defects and link uncovered defects to static, measurable properties of the generated code. We also employ common metrics of effectiveness and efficiency that are standard practices throughout empirical software engineering research.

Threats to *internal validity* regard factors that could lead to confounding the causes of observed phenomena. In our study, the main obfuscating factor in regards to causality is the heavy reliance on randomness. We take several measures to address the large degree of randomness inherent to fuzzing approaches. First, we individually assess each uncovered defect and analyze the components of the fuzzer involved in its generation. Through this procedure, we identify the root causes of generating compiler-crashing code and isolate the merits and disadvantages of different heuristics. Second, we ensure the fairness of the comparisons by sharing parameters between different configurations with default values from the literature. We also perform the comparison using a single implementation of the tool, that relies on the same sampling process and genetic operators. Lastly, to assess the effectiveness and efficiency of our algorithms, we repeat independent runs 10 times and report average values that are subject to standard statistical analyses.

Threats to *conclusion validity* affect the relation between the available data and the credibility of the conclusions we derive based on it. To this end, we base our analysis on over 50,000 generated files, and vary the essential hyperparameters of our algorithms to several sensible values. We additionally perform statistical tests to compare algorithms representative of their respective class. In particular, we base our performance conclusions on the application of the Wilcoxon signed-rank test, which is a statistical procedure that does not impose unreasonable restrictions on the data distribution.

Threats to *reproducibility* concern factors that might cause the application of the same research methods to result in significantly divergent or conflicting observations. To mitigate this, we supply the entire code base that implements our approach [12], in addition to the entire set of generated files and adjacent preprocessed data [11]. We also provide extensive documentation to detail our tool, its configuration, and its applicability. To ensure that no environmental factors interfere with the fuzzer, we use containerization to isolate the dependency management and runtime of our tool.

7 CONCLUSIONS AND FUTURE WORK

We proposed a generalizable three-stage approach that intertwines syntax, semantics, and meta-heuristic search. Our method prunes semantically rich grammar productions and replaces them with context-aware counterparts to generate valid code programs. We structure the code that emerges from sampling the enriched grammar structure into a hierarchical representation based on scope and complexity. This representation forms the basis of an evolutionary framework that provides guidance to the sampling process.

We introduced two instances of evolutionary algorithms that are novel to the field of compiler fuzzing. The algorithms seek to drive the population toward a diverse collection of code that exercise different combinations of language features. We implemented both single- and many-objective formulations of genetic algorithms (GAs), in addition to the standard random sampling (RS).

We analyzed the behavior and performance of the proposed approaches in an empirical analysis spanning 50,000 generated Kotlin files, which we analyzed through differential testing between the recent K1 compiler and the upcoming K2 version. Our results uncovered three previously unreported categories of bugs, which we reported to the Kotlin compiler developer team. The developers verified and replicated our instances on the current release of the Kotlin compiler. Compiler developers are either working on fixing the reported issues, or have already resolved them in more recent compiler releases. While a comparative analysis between RS and GAs shows no significant difference in the number of bugs they find, their driving mechanisms favor different code patterns, which in turn materialized in distinct bugs uncovered by each heuristic. We foresee multiple possible directions for future work.

Further grammar enrichment. The current version supports custom sample operations for a limited number of Kotlin CFG rules. Extending these to additional rules would allow the fuzzer to generate more varied and interesting code efficiently.

Compiler Integration. The heuristics and fitness functions explored in this study guide the generative sampling process towards promising areas of the Kotlin code space. In parallel to those methods, one could leverage compiler information directly into the search algorithm. Our tool includes a module that allows the fuzzer to query for files that trigger faults, according to DT.

Integration with Mutation Fuzzing Stepanov et al. [38] introduced a mutation-based fuzzer for Kotlin, that alters input code in a sound and type-aware manner. Since the variation operators of our GAs are both vastly different and less powerful than those in the mutation fuzzer, exploring the integration of the two could give rise to new, otherwise unattainable pieces of code.

ACKNOWLEDGEMENTS

This work has been produced in the scope of the AI for Software Engineering (AI4SE) research collaboration, funded by JetBrains.

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 143–154.

[2] Zohreh Aghababaeyan, Manel Abdellatif, Lionel Briand, S Ramesh, and Mojtaba Bagherzadeh. 2023. Black-box testing of deep neural networks through test case diversity. *IEEE Transactions on Software Engineering* (2023).

[3] Marat Akhin and Mikhail Belyaev. 2020. Kotlin language specification: Kotlin/-Core. *JetBrains Research* (2020).

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[5] Theodore W Anderson and Donald A Darling. 1954. A test of goodness of fit. *Journal of the American statistical association* 49, 268 (1954), 765–769.

[6] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.

[7] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

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

[9] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2013. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media.

[10] Abdulazeez S Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Information and software technology* 39, 9 (1997), 617–625.

[11] Georgescu Calin. 2023. *Empirical Study Data for Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler*. https://doi.org/10.5281/zenodo.8221889

[12] Georgescu Calin. 2023. *Fuzzer Implementation for Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler*. https://doi.org/10.5281/zenodo.8222995

[13] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

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

[15] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[16] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[17] William Jay Conover. 1999. *Practical nonparametric statistics*. Vol. 350. john wiley & sons.

[18] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful APIs. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 226–236.

[19] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2022. Generating Class-Level Integration Tests Using Call Site Information. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2069–2087.

[20] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[21] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20 (2015), 611–639.

[22] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

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

[24] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[25] Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. 2020. Language-agnostic generation of compilable test programs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 39–50.

[26] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. 2020. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics* 52, 5 (2020), 3745–3756.

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

[28] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26 (2004), 369–395.

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

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

[31] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

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

[33] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 20–27.

[34] Terence Parr. 2013. The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference* (2013), 1–326.

[35] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375.

[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

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

[38] 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.

[39] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. 2023. Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 36–47.

[40] 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*. 283–294.

Received 2024-02-08; accepted 2024-04-18