UDC 004.05
doi:10.31799/1684-8853-2022-6-31-40
EDN: OOCYPQ

Articles

Backend Bug Finder — a platform for effective compiler fuzzing

***D. S. Stepanov*[a]**, *Senior Lecturer, orcid.org/0000-0003-1719-0325*
***V. M. Itsykson*[a]**, *PhD, Tech., Professor, orcid.org/0000-0003-0276-4517, vlad@icc.spbstu.ru*
[a]*Peter the Great St. Petersburg Polytechnic University, 29, Politekhnicheskaia St., 195251, Saint-Petersburg, Russian Federation*

Introduction: *The standard way to check the quality of a compiler is manual testing. However, it does not allow to cover a vast diversity of programs that can be written in a target programming language. Today, in addition to manual written tests there are many automated compiler testing methods, among which fuzzing is one of the most powerful and useful. A compiler fuzzer is a tool that generates a random program in a target language and checks how the compiler works in this language.* ***Purpose:*** *To develop a platform for compiler fuzzing and, based on it, to develop a tool for Kotlin compiler testing.* ***Results:*** *We have developed Backend Bug Finder which is a platform for compiler fuzzing is. We have chosen a mutation-based approach as a method for generating random programs. First, an existing program is entered to the mutator as the input to be then transformed in some way. Mutations can be both trivial, for example, replacing arithmetic operators with others, and complex, changing the structure of the program. Next, the resulting program is fed to the input of the compiler with the following check of its operation. The developed test oracle can detect three types of errors: crashes, miscompilations, and performance degradations. If an error is detected, the test case is fed into the post-processing module, where reduction and deduplication algorithms are applied. We have developed a tool for fuzzing the Kotlin language compiler based on the platform for its approbation, which showed the applicability of the proposed approach for finding errors in modern compilers.* ***Practical relevance:*** *Over a year and a half of work, our tool has found thousands of different Kotlin compiler bugs, more than 200 of which were sent to the developers, and more than 80 have been fixed.*

Keywords *— fuzzing, compiler testing, compiler test generation, Kotlin, mutational fuzzing.*

For citation: Stepanov D. S., Itsykson V. M. Backend Bug Finder — a platform for effective compiler fuzzing. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 6, pp. 31–40. doi:10.31799/1684-8853-2022-6-31-40, EDN: OOCYPQ

Introduction

Often, software developers are faced with situations when their programs do not compile, run abnormally slowly, or not as they should. Sometimes these problems are results of compiler errors: bug trackers of popular compilers (clang, gcc, javac, etc.) contain tens of thousands of bugs found by users. The quality and reliability of compilers is the primary target of language development teams. For example, in the Kotlin (https://kotlinlang.org/) compiler project bug tracker at the beginning of February 2022 there are 30,256 issues (https://youtrack.jetbrains.com/issues?q=%23Kotlin). Most of them were found by users of the language, who, with each error they find, feel less and less desire to work with the compiler. Despite the huge number of manually written tests, compiler developers and testers cannot cover all the use cases of the language, since the number of possible programs is infinite.

How can we try to reduce the number of errors faced by compiler users? This is where the methods of automatic testing of compilers can help us, one of the most popular of which is fuzzing. Fuzzing was invented in 1988 at a Barton Miller seminar at the University of Wisconsin [1]. Fuzzer was a program that generated random sequences of characters and fed them to the input of UNIX command line utilities and see if the program crashed on the generated data or not. It means that the compiler fuzzer should generate correct random programs in the target language and check whether they will be compiled correctly.

In this paper we are presenting a platform for compiler fuzzing, named Backend Bug Finder (BBF) and the implementation on its basis of the Kotlin compiler fuzzer. This tool is designed to find bugs related to the correct work of the compiler, and to its performance. Our fuzzer is mutational, it takes manually written code as input and tries to modify it in a non-trivial way. In addition to the test generation, BBF implements algorithms for bugs post-processing: deduplication and reduction. The results of the tool are highly appreciated by the language development team and are useful in practice.

Related works

Even before the fuzzing was coined, there were various attempts to automatically test compilers with randomly generated programs as early as 50 years ago. This section discusses the development of compiler fuzzing area, as well as the most popular approaches and tools. Comprehensive overview of the compiler testing area is given in the article by Chen J. et al. [2].

The purpose of any compiler fuzzer is to generate a random, non-trivial, correct program and check the compiler's work on it. Why should generated programs be non-trivial? Because bugs in all trivial programs have long been fixed. And as the compiler develops, the complexity of error-containing programs increases. Why do generated programs need to be correct? A syntactically incorrect program can only test the parser. Semantically incorrect ones can find compiler errors, but in practice such cases will never occur and fixing such bugs will not lead to a significant improvement in the quality of the compiler. So automatic program generation for compiler testing is a big problem. All approaches for production of compiler tests can be divided into two groups: generative and mutational ones. It follows from the titles that generative approaches try to generate programs in some way, while mutational approaches modify existing ones.

At the beginning of the development of the compiler fuzzing area, the most obvious and simple approach was implemented: the generation of code based on grammar. Purdom implemented such a generator in his work [3]. The algorithm consists in iterating by the production rules of grammar, and trying to compile the resulting code. This approach is suitable only for testing parsers, since almost all the code obtained in such a way is semantically incorrect. After that, various researchers introduced improved types of grammars, which allowed to add semantic rules into the generation process (affix, attribute, W-grammars) [4–6]. Unfortunately, for modern programming languages, none of them allows to generate non-trivial, semantically correct programs.

The work of Kreutzer P. et al. [7] can be considered as one of the modern grammar-based approaches. The authors presented a language for describing attribute grammars that contain both syntactic and semantic rules of the target programming language in a declarative way. The disadvantage of this approach is that in practice, it is hardly possible to write such grammar covering all modern language features and ways of their interaction. Therefore, in practice, it is limited to a small subset of the language.

In addition to grammar-based approaches, there are methods that somehow use grammar to improve the generation process (grammar-aided approaches). The most famous grammar-aided tool is CSmith [8], designed for testing C/C++ compilers. The difference between this tool and the approaches described above is that the program generation rules are described manually in the source code. The generation process consists of creating a set of data structures and the main function, which contains various scenarios of their use. Each generation step uses a grammar to understand which production rule can be applied and performs a set of checks to ensure that the resulting program is correct and free from undefined behavior. Over the years, the tool has found a large number of bugs in the GCC and Clang compilers, which greatly helped the developers.

Besides of the mentioned, there are a huge number of generative approaches that implement various ideas, for example, deriving generation rules from existing code in the target programming language [9], using machine learning [10], testing certain compiler features in programming language compilers by writing a generator of the necessary expressions [11], etc.

As was written earlier, in addition to generative approaches, there are also mutational approaches. They differ from generative ones in that they use an existing program in the target programming language as an initial seed and try to somehow modify part of it to get a new one. Mutational approaches are divided into two groups: preserving and not preserving the original semantics.

Semantic-preserving mutational approaches are based on the equivalence modulo inputs [12]. The main idea of equivalence modulo inputs is that two programs are equivalent to each other under a set of input data if for each input of the set their behavior is the same. It means that mutations by construction should produce code that is semantically equivalent to the original one. Next, we check that programs that are syntactically different but semantically equivalent actually work the same way. Examples of such mutations can be different: changes to dead code that are not affected during program execution, or, conversely, its insertion. A more difficult example can be various options for replacing random expressions in a program with equivalent ones. Despite the shown efficiency and usefulness, this type of mutation is not intended for full-fledged testing of compilers, because it is very limited.

Non-semantic-preserving mutations are those that change the semantics of the program. It can be anything: replacing the signs of arithmetic expressions, variables renaming, changing the class hierarchy, etc. Such mutations do not guarantee the semantic correctness of the resulting program, so after each mutation it should be checked, and, in case of incorrectness, it is necessary to roll back to the previous correct state. Non-semantic-preserving mutations can be performed on different types of code representation: text, abstract syntax tree (AST), control flow graph, etc. For example, Chen et al. in their work [13] successfully used 129 non-semantic-preserving mutations to test various Java Virtual Machine (JVM) implementations and found more than 50 bugs in it. Another good example is Holler's work on finding vulnerabilities in the JavaScript (JS) interpreter of Mozilla Firefox [14]. The main idea is to replace non-terminal symbols with non-terminals of the same type, but from other programs. In order to increase the probability of producing correct code, the authors implemented a set of heuristics, such as changing variable names, etc.

As can be seen from this section, compiler testing is a very big and practically useful scientific area separate from classic fuzzing. Each of the tools given as an example found a big number of errors. Ideally, a compiler fuzzer should be used in the development of any programming language, regardless of the approach that it implements, as practice shows, it will find some errors and help users to use the programming language more safely. Unfortunately, almost all tools are designed to test a specific programming language or even specific compiler, so creating one for Kotlin is an actual task.

BBF in detail

Backend Bug Finder is a platform for finding bugs in the programming languages compilers. Its development began in 2019 and is still ongoing. Project is open source and available on Github (https://github.com/DaniilStepanov/bbfgradle). In this section we will consider in detail its structure and operation principle: from methods for producing test cases to post-processing of results.

BBF overview

Our platform for compiler fuzzing (Fig. 1) consists of a number of components. In this section, we will consider the purpose and algorithm of each of them.

Test generator is responsible for test generation. From all the existing approaches, we decided to implement non-semantic-preserving mutations that are applied in random order to a random test from the initial test suite. The output of the test generator is a mutant program to check the correctness of the compiler. The mutation process is directed using the information received from the compilation checker, which allows one to understand that the current mutant is incorrect or meaningless and do not work with it further.

A component called *compilation checker*, also known as a test oracle, is responsible for exchanging information with the compiler. The input for this component is a mutant program, which it sends to the compiler for execution and analyzes the output, which is then passed to the test generator to direct its work. If a bug was found, then it is transferred to the bug manager for post-processing.

Bug manager is the component responsible for the found bugs post-processing. Usually, it includes algorithms for deduplication, reduction and isolation of errors. The output of this component is a well-formatted and minimized test containing a bug previously not found in the compiler, ready to be automatically or manually submitted to the appropriate bug tracker.

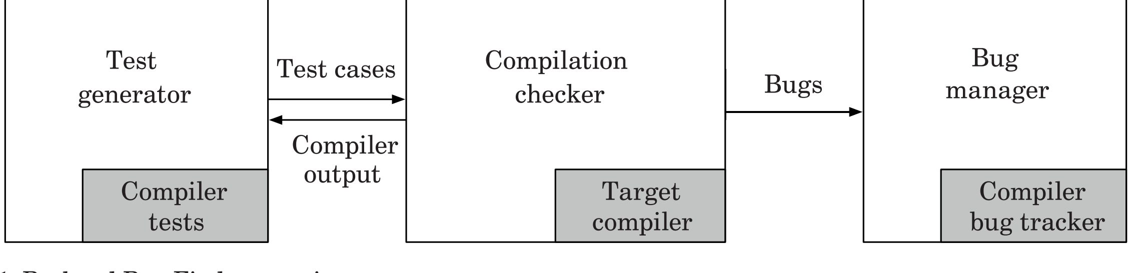
Test generator

As mentioned earlier, non-semantics-preserving mutations were chosen for implementation in the BBF. The grammar-based approach is not very suitable for testing common language compilers due to their complexity. A large number of different features are implemented in the language, and manual description of the rules for its generation requires too much time with no guarantee of results.

Mutations in the BBF can be as diverse as possible: from simple ones like swap random lines or replace plus to minus to complicated mutations with class hierarchy modifications. In addition to such mutations, the platform also provides the ability to write partial generation: for example, random functions or classes.

All mutations are applied in random order and with different probabilities to provide a variety of resulting mutants. It is important to note that since our mutations do not guarantee the preservation of the semantics and correctness of the test, after each of them a check is necessary: if the resulting mutant is incorrect, a rollback to the previous state is performed. We get such information from the oracle after compilation and execution of the obtained mutant.

As input, the test generator takes a random test from the compiler's test seed, which is usually manually written by developers or testers. To improve



■ ***Fig. 1.*** Backend Bug Finder overview the quality of fuzzing, they should be diverse and cover all language features and constructions. All tests must have a single structure: at the beginning, a set of various data structures, functions, and other top-level objects is defined, after which a function is declared that contains different scenarios for using the previously described components.

Mutations can be performed over two types of source code representation: textual and abstract syntax tree (AST). The textual representation can be used for simple mutations using regular expressions or line-level modifications. The AST is used for more complex transformations, such as changing the structure and architecture of tests. For example, when modifying a function argument, you have to modify all the call places. To improve the mutation process, the AST can be extended with various semantic information obtained from code analysis: types, descriptors, and so on.

Compilation checker

The compilation checker is used to determine that the compiler is not working correctly. To do this, the program obtained from the test generator is compiled and executed, after which the result of compiler work is analyzed. In terms of fuzzing, the compiler is a program under test (PUT) and the compilation checker is a test oracle. According to the type of information received from the compiler, test oracles are divided into black-box, gray-box and white-box. In the case of black-box oracle, we only get information from the program I/O stream, which does not allow us to direct the fuzzing process in any way. The gray-box oracle receives additional information from the PUT, usually its coverage, which is then used to target the fuzzing process to cover necessary source code. In white-box testing, a PUT model is built, on which symbolic execution methods are used to generate new input test data.

In terms of getting information from the compiler, BBF is a black-box fuzzer — we work only with the information received from compiler I/O. The generation of tests for compiler testing is very different from generation of not structured input data for other programs, and in the case of target compiler testing, using coverage information as in general fuzzing will not give an increase in testing quality. Also, practice shows that prioritization of various mutations also does not increase the quality of common compiler fuzzing. Work on using compiler coverage information to improve the quality of target fuzzing is one of the directions for future work.

Regarding the types of bugs we are searching for, compilers are large and diverse software projects, so errors are not limited to crashes. This section discusses the types of bugs we are looking for and how we detect them.

Compiler crashes are the most common and easily detected kind of errors. To find bugs of this type, we just need to parse the message that the compiler ends with. This type of error occurs most often. With such an error, you will not be able to run the resulting code, because the compiler for some reason could not compile it.

The next type of bug is much more dangerous and is called “miscompilation”. This situation occurs when the compiler works correctly, but the behavior of the program does not correspond to the semantics of the language. To search for this type of errors we need to compile the program with different compilers (or different versions or modes of the same compiler), instrument the compiled code and check its behavior at runtime. If the behavior of the program compiled in one of the modes or by one of the compilers differs from the others, then at least we have found an interesting situation, and as a maximum this situation turns out to be a bug.

Last but not least are performance bugs. This type of error is associated either with too long compilation time, or, more seriously, with abnormally high execution time. Such errors are also detected using the “voting” system, but the whole difficulty lies in measuring the program running time, since the measurement error usually makes it difficult to separate bugs from false positives. BBF deals with such situations by adjusting the number of test runs until the measurement error becomes negligible.

Of course, considered bug types list is not final. The types of bugs to be found are limited only by the developer’s imagination. As a part of the project, work was done to find errors in the Kotlin language compiler plugin, in application binary interface, certain components of the compiler, etc.

Bug manager

Practice shows that without this component it is impossible to effectively test the compiler. For example, usually, in a few hours of work, fuzzer finds more than a hundred bugs, almost all of which are duplicates. In addition to a large number of duplicates, test cases that lead to an error contain a large amount of irrelevant information, the manual removal of which takes a lot of time. It means that any fuzzing platform should contain post-processing algorithms. They typically include reduction, deduplication, and isolation algorithms.

Reduction is designed to remove information irrelevant to error from the test. Practice shows that most often the minimized test takes less than 10 lines, while the generated one is many times larger. To reduce tests in BBF, we implemented a hybrid approach: several language-agnostic methods and infrastructure to implement language-specific transformations. Language-specific transformations are intended for complex changes in the structure of the code that cannot be done using automatic methods, for example, when removing a function argument, you must remove it in all call sites. Transformations run one after another until the test is no longer minimized. Reducer overview is shown in Fig. 2.

Hierarchical delta debugging [15] and static backward slicing [16] were chosen as language-independent methods that remove most of information irrelevant to error. The hierarchical delta debugging algorithm is based on the idea that on each level of the AST, representing the code with a bug, we are looking for the minimum set of vertices on which the desired error continues to be reproduced, and the vertices that do not affect this are deleted.

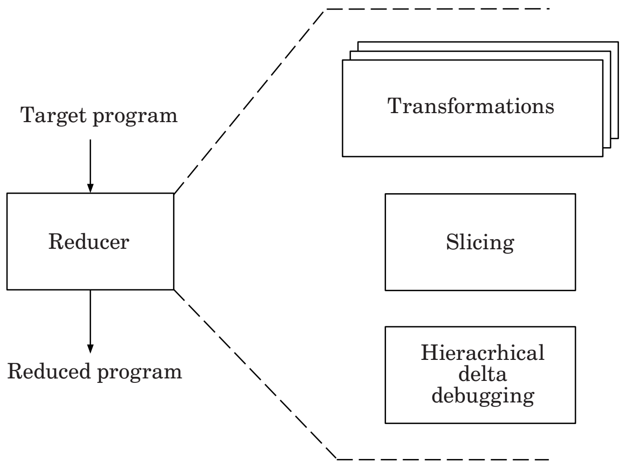
Slicing creates a program slice that contains only code that affects a slicing criterion. In our case, the slicing criteria are the variables found in the line of the test that contains the error. There are several types of slicing with different complexity and quality of work, we chose static backward slicing, which works at the intraprocedural, function and class levels.

As a result, the reduction algorithm removes significant amount of information irrelevant to the bug and greatly improves the manual processing of fuzzing results.

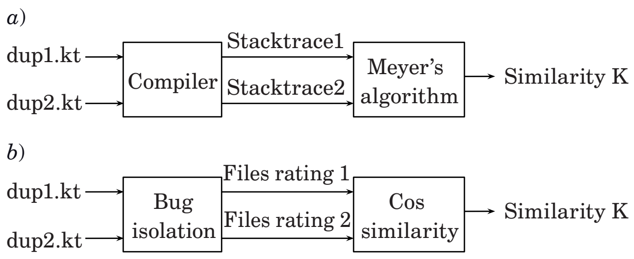
Practice of applying any compiler fuzzing shows that without deduplication, manual processing of found bugs takes too much time. It should be noted that the algorithms described in this section do not completely solve the problem of duplicates, allowing both false-positive and false-negative situations, but still significantly reduce the time of manual processing of results.

The first idea that comes to mind for filtering duplicates is that the same bugs have similar stacktraces. So, we decided to compare them. For this purpose, we are using Myer's difference algorithm [17]. The result of the algorithm is a similarity coefficient (0 <= k <= 1). If the coefficient exceeds a previously empirically selected threshold, then the bugs are considered duplicates. Algorithm is shown in Fig. 3, *a*.

As is clear from the description, stacktraces are required for this algorithm to work, but they are available only in case of a compiler crash. What about other types of bugs when we don't have anything to compare? The solution to this problem lies in the scientific field called "Bug Isolation" [18]. The task of bug isolation algorithms is to determine the place in the source code of the PUT, which is the cause of the error. The result of the algorithm is a set of files sorted by the probability of containing bug in it. The main idea of our method is the hypothesis that bugs are duplicates if they are in the same place in the compiler source code. To determine the source of the error, the method of generating witness programs described in the article by Chen et al. [19] is used. Scheme of the algorithm is shown in Fig. 3, *b*.



■ ***Fig. 2.*** Reducer overview



■ ***Fig. 3.*** Bug deduplication algorithms: *a* — using stacktraces comparison; *b* — using bug isolation techniques

Kotlin compiler fuzzing tool implementation

Based on the platform described in the previous section, we have implemented a tool for the Kotlin compiler fuzzing. In this section we would like to discuss some of the more interesting implementation details.

Working with Kotlin

To implement our tool, we need to be able to build a structural representation of Kotlin code, be able to modify it, and also effectively test the compiler on it. All the necessary tools are already implemented inside the Kotlin compiler, so we decided to build a tool on top of it. In order not to be part of a huge project, we decided to use the necessary compiler modules as libraries. However, this solution also has a drawback –when updating the compiler version, the versions of the libraries do not always remain compatible, which makes the update impossible.

The compiler infrastructure also allows us to use the Program Structure Interface (PSI) trees (https://plugins.jetbrains.com/docs/intellij/psi.html) — traditional concrete syntax tree implementation used in JetBrains IntelliJ Platform. PSI is a standard way of representing structured code inside a Kotlin compiler. In addition to syntactic information, the tree contains information obtained from code analysis: types, descriptors, and so on. PSI makes it easier to perform the modification of the code and allows you to make changes that cannot be made with only syntactic information.

Also, using the compiler as a library makes it possible to compile code much more efficiently due to the lack of overhead for running compilation externally. But it is worth noting that in the case of the Kotlin compiler, after a few hours it starts to work more and more slowly, so sometimes the fuzzing process needs to be restarted.

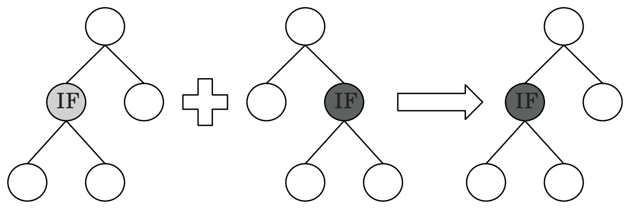
Test generator implementation

For the Kotlin language we have implemented more than 50 mutations that are performed over two types of code representation: text and PSI. In this section, examples of interesting and diverse mutations will be given; a complete list can be found in the source code of the tool. To improve the mutation process, a random type generator and a generator of random expressions of the given types were implemented. It is also possible to work with the Kotlin standard library or use external ones.

The initial seed for the test generator is tests for the Kotlin compiler, written manually by the developers (https://github.com/JetBrains/kotlin). There are quite a lot of them (6000+), they are diverse and cover all language features and constructions. Tests have a single structure: at the beginning, a set of various data structures, functions, and other top-level objects is defined, after which a function named *box* is declared that contains different scenarios for using the previously described components.

PSI modifier

This mutation is designed to modify the PSI without using any semantic information. Nodes in the tree can be swapped, added, removed, or changed to nodes of the same type from other tests. Let's consider the last item in more details. An example of such an operation is shown in Fig. 4. For this mutation to work, first of all, it is necessary to collect a database of trees and their node types. After that, we take a random test case, select a random node and search for nodes of the same type from other tests, take a random one and try to make a replacement: if the resulting test turns out to be correct, then we accept the transformation and continue to



■ ***Fig. 4.*** Example replacing of PSI node to the node from another tree

mutate the new test, if not, we roll back to the previous state.

Expression replacer

This transformation selects a random expression and replaces it with another of the same type that is available in the current scope. The expression we are replacing can be a property, a parameter, a local variable, or a function call that returns a value of the compatible type. An example of applying this transformation is shown in Fig. 5. The original program P1 is shown in Fig. 5, *a*, suppose we choose to replace the `b` property initializer of type `Int` shown in Fig. 5, *b*. Further, for the place where the change will be made, we calculate the scope, for example, this will be `a` and result of calling the `getSum` function. After that, we select a random element of the same type from the scope and make a replacement. However, when choosing `getSum`, we need to generate a call to this function. And when generating a call, variables from the current scope can also be used. The generation result is shown in Fig. 5, *c*.

a)
```
fun getSum(a: Int, b: Int) = a + b

fun box(): String {
       val a = 1
       val b = 1
       return if (a == b) "OK" else "ERROR"
}
```

b)
```
fun getSum(a: Int, b: Int) = a + b

fun box(): String {
       val a = 1
       val b = [Int]
       return if (a == b) "OK" else "ERROR"
}
```

c)
```
fun getSum(a: Int, b: Int) = a + b

fun box(): String {
       val a = 1
       val b = getSum(a, 9)
       return if (a == b) "OK" else "ERROR"
}
```

■ ***Fig. 5.*** Expression replacer mutation example: *a* — P1; *b* — P2; *c* — P3

Random component generator

This transformation represents the generative part of the tool. To increase the diversity and complexity of tests, a generator of random data structures was implemented. This generator allows you to create interfaces, abstract classes, objects and standard classes of all kinds. A class can contain properties, functions, or nested classes. They can also be inherited from others while maintaining code correctness. To implement this generator, a random type generator was also implemented, which takes into account the context and the Kotlin standard library. Generation of non-trivial function bodies is a complex and time-consuming task and one of the primary directions of future work. An example of the generated class is shown in Listing 1. In addition to adding classes, the generator can insert various properties and functions into existing classes.

Listing 1. Example of generated class

```
open class A {
    private val a: Int = 1
    public inline val b: List<Int>
              get() = listOf()
    open fun c(): A = TODO()
    inner class B : A() {
              val a: Double = 0.0
              override fun c(): A = A()
    }
}
```

Compilation checker

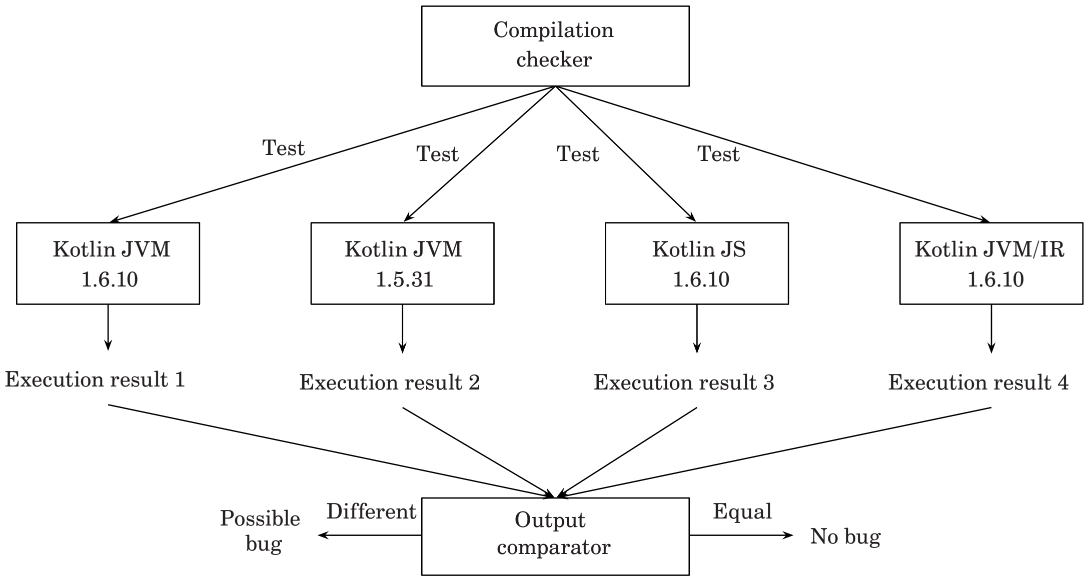
To implement an oracle for the Kotlin compiler, first we need to understand what we are testing, what kind of errors we can look for and how to detect them. The main platform for Kotlin compiler is the JVM, but there are also backends that allow compiling Kotlin to JS and to native binaries, which can run without a virtual machine. At this moment our tool supports JVM and JS backends. Adding a native backend is considered future work. In addition to the release versions of the compiler, we are testing various dev-builds for pre-release testing of new functionality.

The first type of errors we look for are crashes, and it's quite easy to define them: their messages have a specific format that is easy enough to parse. Listing 2 shows an example of the crash message for the Kotlin compiler.

Listing 2. Example of Kotlin compiler crash message

```
  Exception in thread "main" org.jetbrains.kotlin.codegen.CompilationException: Back-end (JVM) Internal error: wrong bytecode generated
    L0
      LINENUMBER 6 L0
      NOP
  File being compiled: (1,1) in /File.kt The root cause
  …
```

Detecting miscompilations is a task of a different complexity. To search for miscompilations, we need to compile the instrumented program with different backends or in different modes of the same backend and compare their behavior. This is done using the voting method, the scheme of which is shown in Fig. 6. And it is important to take into account the specifics of the backend, for example, in JS division by 0 is undefined, and in the JVM it is an exception. For such situations, we have implemented a set of filters that help save time on manual processing of such cases.



■ ***Fig. 6.*** Voting system for the Kotlin compiler

Bug manager implementation

The bug manager is used to reduce the time for manual processing of found bugs. All algorithms and approaches described in the previous section are suitable for Kotlin compiler bugs post-processing. To improve the quality of reduction algorithms, more than thirty Kotlin specific transformations were implemented on two types of code representation: text and PSI. Examples of text transformations are: deleting text inside a balanced pair of brackets, deleting text between two dots, replacing *while* with *if*, etc. PSI based transformations, which are intended for complex modifications can be divided into the following groups:

— expression simplification (replace expression to constant of same type, simplification of *if* statements, loops, elvis operator, etc.);

— removal of unneeded components (imports, function arguments, etc.);

— simplification of interdependencies (removal of inherited properties and functions, replacement of function bodies with TODO(), etc.);

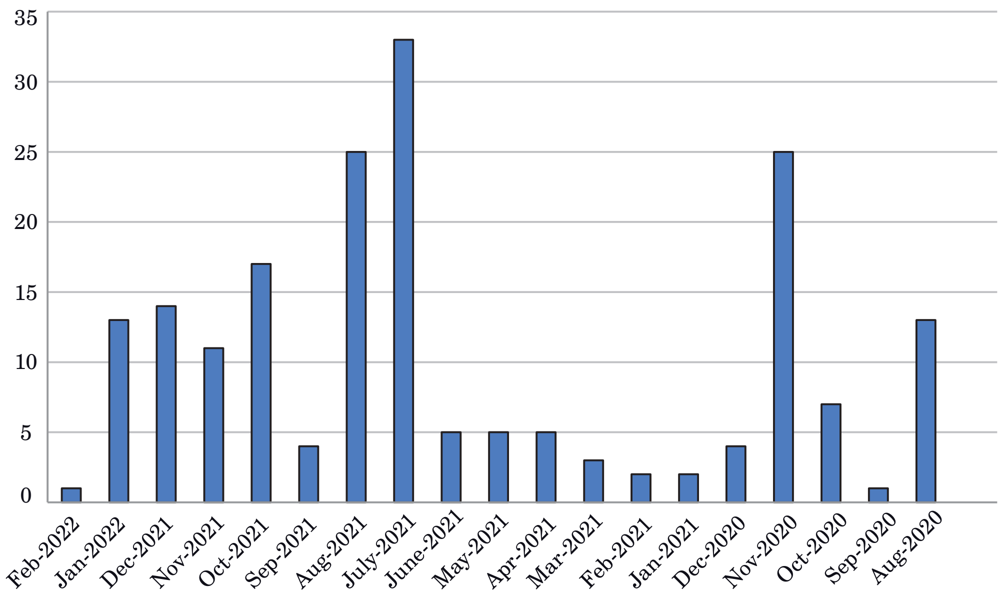
— miscellaneous (comment deletion, formatting, etc.).

More details about the Kotlin program reducer and its results can be found in the article [20]. In general, a complex approach using language-agnostic methods and language-specific transformations gives a good result. In practice, more than 90% of irrelevant information is removed from programs that lead to errors.

Kotlin compiler fuzzer evaluation

Active use of the tool has been going on for more than a 1.5 years. To estimate the effectiveness, we have been collecting statistics and logs all this time. To fuzz the Kotlin compiler, we did not use any big computing power, because, despite the efficient work of post-processing algorithms, filtering uninteresting results and sending them to developers takes a lot of time. All interesting bugs were sent to Kotlin bug tracker — YouTrack (https://youtrack.jetbrains.com/issues/KT?q=%23found-by-fuzzer%20). Evaluation was mainly performed on a single computer with Intel(R) Core(TM) i7-8700K CPU@3.70 HGz and 32 GB RAM. The compiler version has always been kept up to date. As an initial seed for mutation, we selected a compiler test suite, consisting of more than 6000 files. Mostly fuzzing was done for the JVM backend, but for several months we were looking for bugs in the JS. It is also worth noting that BBF has been constantly improving during this time: new mutations and oracles have appeared, because without improvements at some point fuzzing stops working.

Figure 7 shows the number of bugs posted per month for all time of fuzzing. The first results began to be transferred to developers in August 2020. At that time, the compiler was not yet as reliable and stable as it is now. In November 2020, the Type Centric Enumeration approach [21] was implemented, which led to a significant increase in bugs found. After that, BBF interested the compiler development team and a lot of effort was spent to make the tool from research to practical. There were few launches during the rework, so there were few bugs before July 2021. In July 2021, the practice-oriented BBF went live and brought a lot of practical value. Developers often asked to test certain components of the compiler that were under development, which also influenced the general



■ ***Fig. 7.*** Number of bugs posted per month

■ Found bugs statistic

Bug status	Critical	Major	Normal	Minor	Not specified
Fixed	2	45	18	5	19
Open	0	11	12	2	16
In progress	0	1	2	0	1
Other	0	2	3	2	0

trend towards an increase in the number of bugs found.

Table shows statistics on the distribution of found bugs. A total of 208 bugs have been submitted as of this writing. Not specified bugs that do not fall into any category are either duplicates or not interesting from the developer's point of view. To better understand our impact, during the experiment, about 13% of the total number of bugs in the JVM backend was found using BBF. The total number of errors found is several thousand, but most of them are caused by semantically incorrect code. These bugs get a low priority and are unlikely to ever be fixed, so the decision was made not to send such bug reports to the developers.

Conclusion

In this work, we have presented a platform for finding bugs in the programming language compilers — Backend Bug Finder. After reviewing the compiler fuzzing area and choosing the methods to be implemented, we considered all the ideas and principles implemented in the tool. On the basis of the developed platform, a fuzzer was implemented for the compiler of the Kotlin programming language. The results of fuzzing shows that the use of the fuzzer greatly improves the quality of the compiler: we published about 13% of the total number of bugs in the JVM backend reported during our experiment.

How difficult is it to adapt the platform to another programming language? It is pretty easy to make a naive fuzzer, which will probably find some errors. But for deep and complete testing of the compiler, efforts will have to be made to write language-specific transformations, reduction algorithms and infrastructure for the efficient execution of generated programs.

As for future work, the BBF will improve both in practical and theoretical terms. In practical side, we plan implement a parallelling of the fuzzing process. In theoretical side, we will move towards gray-box fuzzing and other methods of program generation for compiler testing purposes.

References

1. Miller B. P., Koski D., Lee C. P., Maganty V., Murthy R., Natarajan A., & Steidl J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. *Computer Sciences Department University of Wisconsin*, 1995. Available at: https://minds.wisconsin.edu/bitstream/handle/1793/59964/TR1268.pdf (accessed 16 August 2022).
2. Chen H., Pendleton M., Njilla L., & Xu S. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 2020, vol. 53, no. 1, pp. 1–36. doi:10.1145/3363562
3. Purdom P. A sentence generator for testing parsers. *BIT Numerical Mathematics,* 1972, vol. 12, no. 3, pp. 366–375. doi:10.1007/BF01932308
4. Hanford K. V. Automatic generation of test cases. *IBM Syst. J.*, 1970, vol. 9, no. 4, pp. 242–257. doi:10.1147/sj.94.0242
5. Duncan A. G., Hutchison J. S. Using attributed grammars to test designs and implementations. *Proc. of the 5th Intern. Conf. on Software Engineering*, 1981, pp. 170–178.
6. A. van Wijngaarden. *Orthogonal Design and Description of a Formal Language.* Stichting Mathematisch Centrum, 1965.
7. Kreutzer P., Kraus S., Philippsen M. Language-agnostic generation of compilable test programs. *2020 IEEE 13th Intern. Conf. on Software Testing, Validation and Verification (ICST)*, 2020, pp. 39–50. doi:10.1109/ICST46399.2020.00015
8. Yang X., Chen Y., Eide E., Regehr J. Finding and understanding bugs in C compilers. *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2011, pp. 283–294. doi.org:10.1145/1993498.1993532
9. Bastani O., Sharma R., Aiken A., Liang P. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 2017, pp. 95–110. doi:10.1145/3140587.3062349
10. Cummins C., Petoumenos P., Murray A., Leather H. Compiler fuzzing through deep learning. *Proc. of the 27th ACM SIGSOFT Intern. Symp. on Software Testing and Analysis*, 2018, pp. 95–105. doi:10.1145/3213846.3213848
11. Nagai E., Awazu H., Ishiura N., Takeda N. Random testing of C compilers targeting arithmetic optimization. *Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2012)*, 2012, pp. 48–53. doi:10.1145/3428264
12. Le V., Afshari M., Su Z. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 2014, pp. 216–226. doi:10.1145/2594291.2594334
13. Chen Y., Su T., Sun C., Su Z., Zhao J. Coverage-directed differential testing of JVM implementations. *Proc. of the 37th ACM SIGPLAN Conf. on Programming*

Language Design and Implementation, 2016, pp. 85–99. doi:10.1145/2908080.2908095

14. Holler C., Herzig K., and Zeller A. Fuzzing with code fragments. *Proc. of the 21st USENIX Conf. on Security Symp.*, 2012, pp. 445–458. doi:10.5555/2362793.2362831

15. Misherghi G., Su Z. HDD: Hierarchical delta debugging. *Proc. of the 28th Intern. Conf. on Software Engineering*, 2006, pp. 142–151. doi:10.1145/1134285.1134307

16. Weiser M. Program slicing. *IEEE Transactions on Software Engineering*, 1984, vol. SE-10, iss. 4, pp. 352–357. doi:10.1109/TSE.1984.5010248

17. Myers E. W. AnO (ND) difference algorithm and its variations. *Algorithmica*, 1986, no. 1, pp. 251–266. doi:10.1007/BF01840446

18. Wong W. E., Gao R., Li Y., Abreu R., Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 2016, vol. 42, pp. 707–740. doi:10.1109/TSE.2016.2521368

19. Chen J., Han J., Sun P., Zhang L., Hao D., Zhang L. Compiler bug isolation via effective witness test program generation. *Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, 2019, pp. 223–234. doi:10.1145/3338906.3338957

20. Stepanov D., Akhin M., Belyaev M. ReduKtor: How we stopped worrying about bugs in kotlin compiler. *2019 34th IEEE/ACM Intern. Conf. on Automated Software Engineering (ASE)*, 2019, pp. 317–326. doi:10.1109/ASE.2019.00038

21. Stepanov D., Akhin M., Belyaev M. Type-centric Kotlin compiler fuzzing: Preserving test program correctness by preserving types. *14th IEEE Conf. on Software Testing, Verification and Validation (ICST)*, 2021, pp. 318–328.

УДК 004.05
doi:10.31799/1684-8853-2022-6-31-40
EDN: OOCYPQ

Backend Bug Finder — платформа для эффективного фаззинга компиляторов

Д. С. Степанов[a], старший преподаватель, orcid.org/0000-0003-1719-0325
В. М. Ицыксон[a], канд. техн. наук, доцент, orcid.org/0000-0003-0276-4517, vlad@icc.spbstu.ru
[a]Санкт-Петербургский политехнический университет Петра Великого, Политехническая ул., 19, Санкт-Петербург, 195251, РФ

Введение: стандартным способом проверки надежности компилятора является ручное тестирование. Но с его помощью невозможно покрыть множество программ, которые могут быть написаны на целевом языке программирования. В настоящее время в дополнение к ручному тестированию существует множество автоматических методов проверки надежности компиляторов, среди которых фаззинг является одним из самых мощных и полезных. **Цель:** разработать платформу для фаззинга компиляторов и на ее основе инструмент для тестирования компилятора языка Kotlin. **Результаты:** разработана платформа Backend Bug Finder для фаззинга компиляторов. В качестве метода для генерации случайных программ выбран мутационный подход, когда на вход мутатору подается программа, которую он пытается каким-либо образом преобразовать. Мутации могут быть как тривиальными, например замена арифметических операторов другими, так и сложными, меняющими структуру программы. Далее полученная программа подается на вход компилятору, и производится проверка его работы. Разработанный тестовый оракул может детектировать ошибки трех типов: падения, мискомпиляции и деградации производительности. В случае обнаружения ошибки тестовый пример подается в модуль постобработки, где применяются алгоритмы редукции и дедупликации. На основе платформы для апробации подхода разработан инструмент для фаззинга компилятора языка Kotlin, который показал применимость платформы для поиска ошибок в современных компиляторах. **Практическая значимость:** за полтора года функционирования разработанный инструмент обнаружил тысячи различных ошибок компилятора языка Kotlin, из них более 200 отправлено разработчикам, более 80 исправлено.

Ключевые слова — фаззинг, тестирование компилятора, генерация компиляторных тестов, Kotlin, мутационный фаззинг.

Для цитирования: Stepanov D. S., Itsykson V. M. Backend Bug Finder — a platform for effective compiler fuzzing. *Информационно-управляющие системы*, 2022, № 6, с. 31–40. doi:10.31799/1684-8853-2022-6-31-40, EDN: OOCYPQ
For citation: Stepanov D. S., Itsykson V. M. Backend Bug Finder — a platform for effective compiler fuzzing. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 6, pp. 31–40. doi:10.31799/1684-8853-2022-6-31-40, EDN: OOCYPQ