Testing the Compiler for a New-Born Programming Language: An Industrial Case Study (Experience Paper)

Yingquan Zhao
College of Intelligence and Computing, Tianjin University
Tianjin, China
zhaoyingquan@tju.edu.cn

Junjie Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Ruifeng Fu
College of Intelligence and Computing, Tianjin University
Tianjin, China
frf2000@tju.edu.cn

Haojie Ye
Programming Language Lab, Huawei
Hangzhou, China
yehaojie@huawei.com

Zan Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
wangzan@tju.edu.cn

ABSTRACT

Due to the critical role of compilers, many compiler testing techniques have been proposed, two most notable categories among which are grammar-based and metamorphic-based techniques. All of them have been extensively studied for testing mature compilers. However, it is typical to develop a new compiler for a new-born programming language in practice. In this scenario, the existing techniques are hardly applicable due to some major reasons: (1) no reference compilers to support differential testing, (2) lack of program analysis tools to support most of metamorphic-based compiler testing, (3) substantial implementation effort incurred by different programming language features. Hence, it is unknown how the existing techniques perform in this new scenario.

In this work, we conduct the first exploration (i.e., an industrial case study) to investigate the performance of the existing techniques in this new scenario with substantial adaptations. We adapted grammar-based compiler testing to this scenario by synthesizing new test programs based on code snippets and using compilation crash as test oracle due to the lack of reference compilers for differential testing. We also adapted metamorphic-based compiler testing to this scenario by constructing equivalent test programs under any inputs to relieve the dependence on program analysis tools. We call the adapted techniques **SynFuzz** and **MetaFuzz**, respectively.

We evaluated both SynFuzz and MetaFuzz on two versions of a new compiler for a new-born programming language in a global IT company. By comparing with the testing practice adopted by the testing team and the general fuzzer (AFL), SynFuzz can detect more bugs during the same testing time, and both SynFuzz and MetaFuzz can complement the other two techniques. In particular, SynFuzz and MetaFuzz have detected 11 previously unknown bugs, all of which have been fixed by the developers. From the industrial case study, we summarized a series of lessons and suggestions for practical use and future research.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**.

KEYWORDS

Compiler Testing, Program Synthesis, Metamorphic Testing

ACM Reference Format:
Yingquan Zhao, Junjie Chen, Ruifeng Fu, Haojie Ye, and Zan Wang. 2023. Testing the Compiler for a New-Born Programming Language: An Industrial Case Study (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598077

1 INTRODUCTION

Compilers are one of the most fundamental software systems since almost all software is built on top of them. Like other software, compilers also contain bugs [8, 9, 28, 34, 43], which could make any software processed by the buggy compiler produce unexpected behaviors (even disasters in safety-critical domains). That is, compiler bugs could produce a wider influence than the bugs in an application. Therefore, ensuring the quality of compilers is quite important.

In the literature, many compiler testing techniques have been proposed to guarantee the quality of compilers [6, 7, 10, 18, 25, 32, 36, 37, 43]. According to the state-of-the-art survey of compiler testing [9], these techniques can be classified into two main categories: (1) *grammar-based techniques*, which construct test programs based on the programming language grammar and determines whether a bug is detected through comparing the results produced by several comparable compilers (or one compiler under several optimization levels) on the given test program or compilation crash [10, 43, 47]. (2) *metamorphic-based techniques*, which perform mutation on the given test program to produce equivalent test programs (under the

*Junjie Chen is the corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598077

given inputs of the test program) and then compare the results of the compiler under test on these equivalent test programs to detect compiler bugs [20, 29, 41]. Indeed, these techniques have achieved great effectiveness on mature compilers (e.g., GCC [1] and LLVM [2]) [9].

Besides mature compilers, it is also typical to develop a new compiler for a new-born programming language in practice. However, it is still unknown how the existing techniques perform in this practical scenario. Moreover, these existing techniques cannot be directly applied to test such a new compiler due to the following reasons: (1) For a new-born programming language, there is a lack of reference compilers for supporting differential testing of the new compiler, which hinders the use of the grammar-based techniques taking differential testing as the test oracle. (2) There is a lack of accompanying analysis tools for a new programming language (e.g., coverage analysis tools), which hinders the use of the metamorphic-based techniques constructing equivalent test programs under a given set of inputs. (3) The significantly different programming language features (e.g., grammar) can incur substantial implementation effort. In this work, we make the first exploration to investigate the performance of the existing techniques (that have been evaluated on mature compilers) on a new compiler for a new-born programming language through an industrial case study. To achieve this study, we substantially adapted the existing techniques according to the above-mentioned characteristics of this practical scenario.

For the category of grammar-based techniques, the mainstream methods of constructing test programs are based on grammar by either generating test programs from scratch [21, 32, 43] or synthesizing test programs by learning code snippets from existing test programs [24, 40, 47]. For ease of presentation, we call the former *generation-based techniques* and the latter *synthesis-based techniques*. In this work, we just adapted the latter to the new scenario, since there is an in-house testing tool following the idea of generation-based techniques for the new compiler used in our industrial partner, which will be also evaluated in our study (to be presented in Section 3). Specifically, inspired by the idea of synthesis-based compiler testing [24, 47], we designed **SynFuzz**, which first extracts four kinds of code snippets from existing test programs (such as developer-written test programs) and then constructs a new and valid test program by synthesizing a set of code snippets according to the grammar of the new-born programming language. As mentioned above, the widely-used differential-testing mechanism cannot be used as the test oracle for testing such a new compiler. Hence, SynFuzz determines the bug detection by checking whether the new compiler crashes when compiling the constructed test programs.

For the category of metamorphic-based techniques, the most widely-studied ones are EMI-based (equivalence modulo inputs) compiler testing, which constructs equivalent test programs *under a given set of inputs* via program analysis (especially coverage analysis) [28, 29, 40]. Due to the lack of accompanying analysis tools for a new programming language, EMI-based techniques are hardly applicable in this new scenario. To apply the idea of metamorphic testing to this scenario, we design **MetaFuzz** inspired by the existing work [18, 30], which constructs equivalent test programs *under any inputs*. In this way, we can get rid of the dependence on coverage analysis tools. Specifically, we implement six identically equivalent mutation rules in MetaFuzz according to the new programming language features, which can construct the test programs producing the same output as the given test program under any same inputs. If the outputs of the given test program and the mutated programs are different, MetaFuzz detects a compiler bug.

By adapting synthesis-based and metamorphic-based compiler testing to the practical scenario, we implemented both SynFuzz and MetaFuzz and then applied them to test a new compiler CompX (i.e., two relatively stable versions) for a new-born programming language in the global IT company Huawei. Since the new compiler is still under development and not publicly available, we hide the name of the compiler. The results show that SynFuzz outperforms MetaFuzz, the state-of-the-art testing practice adopted by the CompX team (i.e., an in-house generation-based compiler testing tool, called CompXFuzz), and the general fuzzer (i.e., AFL [46]). The improvements of the former over the latter three are 66, 21, and 78 in terms of the number of detected bugs across all the experiments on both versions. Moreover, both SynFuzz and MetaFuzz can complement each other and the other two techniques. For example, SynFuzz and MetaFuzz can detect 61 and 22 unique bugs across all the experiments on both versions. In particular, SynFuzz and MetaFuzz detected 11 previously unknown bugs in total, all of which have been confirmed and fixed by developers. Finally, we deliver a series of lessons learned from the industrial case study and suggestions for practical use and future research. Due to the effectiveness of SynFuzz and MetaFuzz, they have been deployed to test CompX in Huawei.

To sum up, our work makes the following contributions:

- We identify a new but indeed practical scenario of compiler testing, i.e., testing a new compiler of a new-born programming language.
- We designed and implemented SynFuzz and MetaFuzz by adapting synthesis-based and metamorphic-based compiler testing proposed in the scenario of testing mature compilers to the new scenario.
- We deployed both SynFuzz and MetaFuzz to test the new compiler corresponding to a new-born programming language in a global IT company Huawei, demonstrating the effectiveness of SynFuzz and the complementary of SynFuzz and MetaFuzz.
- We deliver a series of lessons learned from the industrial case study and suggestions for future research and better practice in the scenario of testing a new compiler corresponding to a new-born programming language.

2 TECHNOLOGY

2.1 SynFuzz

Following the general idea of grammar-based compiler testing (especially synthesis-based compiler testing), there are some specific techniques in the scenario of testing mature compilers. For example, LangFuzz is a grammar-based technique for testing JavaScript engines, which extracts all code snippets corresponding to each non-terminal in the grammar and generates new test programs through code snippet replacement [24]. JavaTailor is the most recent synthesis-based technique, which is proposed for testing JVMs [47].

Table 1: Summary of code snippet types in SynFuzz

ID	Type	Description
1	Sequential Snippet (SEQ)	A sub-tree containing child nodes that can form a sequence of statements without any conditional branches;
2	Loop Snippet (LOOP)	A sub-tree of `while`, `do-while`, or `for` LOOP, including the loop condition node and the corresponding body node;
3	Conditional Snippet (COND)	A sub-tree of `if` node with its branches (i.e., `else if`, and `else`) or a `switch` node with all the cases, including the conditions of all the branches/cases and the corresponding bodies;
4	Try-Catch Snippet (Try-Catch)	A sub-tree of `try` node, including the `try` body and the statement nodes used for handling the caught exception;

It extracts several pre-defined types of code snippets at the CFG (control flow graphs) level in order to balance extraction efficiency and effectiveness, and then synthesizes new test programs by inserting code snippets into seed programs. Due to the significantly different programming language features and lack of accompanying analysis tools for a new programming language (e.g., CFG analysis tools), these specific techniques cannot be directly applied to the new scenario. Hence, we carefully design and implement SynFuzz with substantial adaptations following the general idea of synthesis-based compiler testing.

As shown in Figure 1, SynFuzz contains four main steps: code snippet extraction (that extracts code snippets from all the given seed programs), code snippet selection (that selects a code snippet from all the extracted code snippets), seed program selection (that selects a seed program from all the given seed programs), and test program synthesis (that inserts the selected code snippet into the selected seed program). Same as the practice of synthesis-based compiler testing [24, 47], SynFuzz conducts random selection for both code snippet selection and seed program selection. The major technical challenges in SynFuzz lie in code snippet extraction and test program synthesis, since the former should determine which kinds of code snippets are helpful to generate effective test programs while the latter should guarantee the validity of a synthesized test program. To balance the testing efficiency and effectiveness, SynFuzz also designs several types of code snippets following the practice of JavaTailor, but extracts and synthesizes them at the AST (Abstract Syntax Tree) level (instead of the CFG level used in JavaTailor). This is because AST is a fundamental and common part of syntactic analysis for most programming languages, which get rids of the dependence on more advanced program analysis tools (e.g., CFG analysis tools) and can also increase the generality of SynFuzz to some degree. In the following, we introduce the two steps in detail.

After constructing a test program, SynFuzz runs it to test the new compiler. If the compiler crashes, it means that it detects a compiler bug; otherwise, the test program will be put into the pool of seed programs for supporting further synthesis.

2.1.1 Code Snippet Extraction. In SynFuzz, we systematically define four types of code snippets at the AST level, which are shown in Table 1. Each code snippet is a sub-tree in the AST, which helps make a trade-off between code-snippet extraction efficiency and effectiveness. Specifically, if SynFuzz measures code snippets at a too fine granularity (e.g, line granularity), which may increase

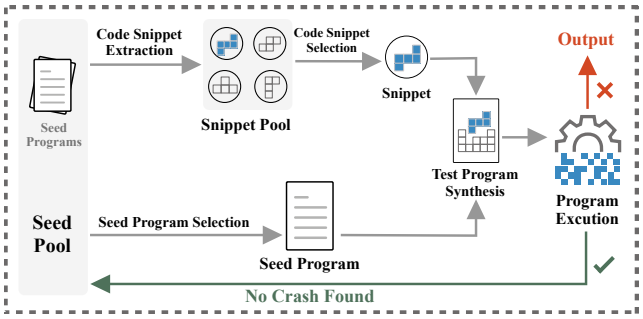


Figure 1: Overview of SynFuzz

the overhead of code snippet extraction and make the diversity of generated test programs small. If SynFuzz measures code snippets at a too coarse granularity (e.g., file granularity), which may lead to a weak dependency between the code snippet and the seed program and thus negatively affect the bug-revealing capability of the synthesized test program.

To extract a code snippet, SynFuzz performs a depth-first search from the root node of the AST to identify the starting node of the sub-tree for a code snippet. The attributes in each node are used to identify the starting node of a sub-tree as well as to determine the type of the corresponding code snippet. For example, a node containing an `if` statement can be identified as the starting node of a sub-tree for a COND snippet (short for Conditional Snippet, as listed in Table 1). Thanks to the tree structure that maintains all the dependencies between nodes, SynFuzz can efficiently identify the starting node together with all its child nodes as a COND snippet. Then, SynFuzz recursively searches the sub-tree of the COND snippet in order to further determine whether it still contains other code snippets. In particular, the method of extracting SEQ snippets is slightly different from the above one extracting the other three types of code snippets. Specifically, since a SEQ snippet is a combination of consecutive leaf nodes and the parents of these nodes may have been identified as the starting nodes of the other three types of code snippets, SynFuzz creates a virtual starting node for these nodes to form a sub-tree and then identifies the sub-tree as a SEQ snippet.

We further illustrate the code snippet extraction process with an example shown in Figure 2. This is a general AST for a function body, where the node labeled as 1 (i.e., node 1) is the root node of the AST. For each node on the search path, SynFuzz checks whether

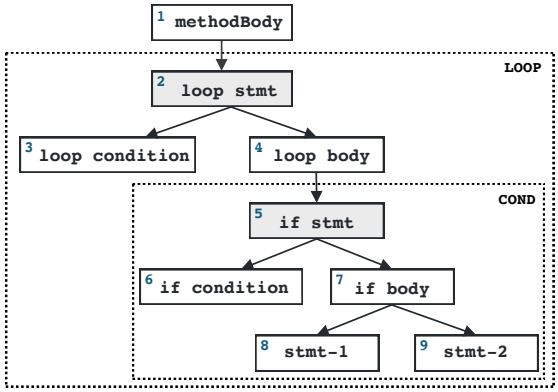


Figure 2: Snippet extraction on an AST of a function body

it contains the statements that can be used to identify the starting node of one of the defined code-snippet types. For example, node 2 contains a loop statement, indicating that it is the starting node of a LOOP snippet; SynFuzz then extracts this node and all its child nodes as a LOOP snippet, corresponding to the part labeled as LOOP in this figure. After identifying the LOOP snippet, SynFuzz continues traversing down from the starting node of the sub-tree of the LOOP snippet, since the loop body may contain other code snippets (e.g., the COND snippet, labeled as COND in the figure). When it comes to node 7, its two child nodes are both leaf nodes and exist in a continuous manner (i.e., there are no other nodes with branches between node 8 and node 9), and thus node 8 and node 9, together with their virtual root node are identified as a SEQ snippet. Finally, SynFuzz can extract 3 snippets in this example, as shown at the bottom of Figure 2.

After completing the code snippet extraction process on the given test programs, a code-snippet pool can be constructed by SynFuzz, which will be used to support the synthesis of new test programs conforming to the grammar.

2.1.2 Test Program Synthesis. Following the practice of synthesis-based compiler testing, SynFuzz synthesizes a new test program by randomly selecting a code snippet and a seed program. While it is effective to ensure the diversity of synthesized test programs, it also suffers from the problem of synthesizing invalid test programs due to the broken syntactic/semantic constraints (e.g., missing the definitions of the variables in the code snippet). Invalid test programs can cause them to be directly rejected at the very beginning compilation stage, which negatively affects the overall testing effectiveness within the given testing time. This conclusion is also confirmed by the result of AFL in our study (to be presented in Section 3.2.1). Therefore, guaranteeing the validity of synthesized test programs is very important.

To solve this problem, SynFuzz provides two strategies to guarantee syntactic and semantic correctness, respectively. The first one is that the synthesis point (i.e., the place where the selected code snippet is inserted) selected by SynFuzz must be valid, which helps guarantee the syntactic correctness of synthesized test programs. A valid synthesis point refers to a node in the AST that can accept child nodes (e.g., the body of `if` statement node). Before synthesis,

```
1  func main(){
2
3    let var1 : Bool <- "A" is Bool
4    if (var1) {
5        Result<Bool>.Ok(var1)
6    } else {
7        Result<Bool>.Err(Exception())
8    } ?? try {
9        func func1<T0>(param1 : Bool, param2 : Int64) : Int64 {
10           param2
11       };
12       var var2 : Int64 <- 1 #constructing new definition
13       func1<Int8>(var1, var2) #newly inserted code ingredient
14       true
15   } catch (e: Exception) { false }
16   Int64(1)
17 }
```

Figure 3: Bug #I4Q8OG

SynFuzz traverses the AST of the seed program to find all the valid synthesis points for the selected code snippet and then randomly selects one as the target synthesis point.

The second one is that the broken constraints must be fixed during the synthesis process, which can help guarantee the semantic correctness of synthesized test programs. There are three types of broken constraints that need to be fixed: 1) *missing types' references*, 2) *missing functions' definitions*, and 3) *missing variables' definitions*. We design the corresponding methods to fix each type of broken constraint, respectively. To complete the references of the missing types, SynFuzz adds the test program from which the selected code snippet is extracted as a dependent module for the seed program. To complete the definitions of the missing functions, SynFuzz checks whether each missing function is an internal function (that can only be accessed in a specific scope). If so, the internal function is also inserted into the synthesis point in the front of the code snippet. Otherwise, SynFuzz checks whether the modifier of the missing function is `public`, and changes it to `public` if not.

The fixing method for the missing variables' definitions is different since a variable (i.e., local variable) cannot be accessed outside its scope even if we change its modifiers. Specifically, SynFuzz first searches for the variables in the seed program that are type-compatible with the undefined variable in the code snippet. If such variables are found, SynFuzz randomly selects one to replace the undefined variable in the code snippet. Otherwise, SynFuzz constructs a new definition for the undefined variable. The insight behind reusing existing variables is that it can strengthen the interaction between the code snippet and the new context provided by the seed program, which facilitates the exploration of new compiler paths and thus increases the probability of revealing new bugs.

We further illustrate the synthesis process with an example shown in Figure 3. This is the pseudo-code of a test program generated by SynFuzz, which makes the new compiler crash (more details can be found in Section 3.2.3). In this example, SynFuzz inserts a SEQ snippet into the seed program (shown at Line 13), where the two arguments of `func1` are actually undefined (we do not show the original variables here). Since the invoked function (i.e., `func1`) in this code snippet is an internal function, and there is no definition of `func1` in the seed program, SynFuzz also inserts the corresponding function definition into the seed program (shown at Lines 9-11). Through searching, we find that there is an existing variable `var1` that has the same type (i.e., `Bool`) as the first argument in the seed program, and thus SynFuzz directly uses `var1` to replace the first argument. However, there is no variable type compatible with the

Table 2: Summary of equivalent mutation rules in MetaFuzz

ID	Type	Description	Example
1	Insert a new function	Insert a function that returns the parameter directly as return value (i.e. f(p) == p), and the corresponding invocation to the newly inserted function;	let var : Int64 <- 5 => func f (param : Int64) : Int64 { return param } let var : Int64 <-f(5)

2	Invert if condition	Invert the condition of if statement as well as the code body corresponding to different conditions;	let var : Int <- 5
			if (var >0) { print("Y") } else { print("N") }
			=>
			if (var <= 0) { print("N") } else { print("Y") }

3	Insert an if statement	Insert an if statement whose condition is an expression that is guaranteed to evaluate to be true/false, and the corresponding `if` body is a code snippet that will always be executed/never be executed;	print("Y") => let var : Bool <- expression if (var) { print("Y") }

4	Insert a loop statement	Insert a loop statement, i.e., `for`, `while`, `do-while`, and the loop count is guaranteed to evaluate to 0/1, and the choice of `loop` body is the same as for `if`;	for (i in []) { # [] refers to an empty list
			#any dead code snippets
			}

5	Variable Equivalence Conversion	Add equivalent operations to variables, e.g., ∗1 for numeric variables, or || `false` for boolean variables;	let var : Bool <- "A" is Bool => let var : Bool <- "A" is Bool || false
6	Expression Equivalence Conversion	Split a numeric expression into multiple equivalent expressions.	let var : Int <- 5 + 1 => let var : Int <- 5 var <- var + 1

second argument (i.e., `var2`) in the seed program, and thus SynFuzz creates a new definition for `var2` with the `Int64` type at Line 12.

Regarding constructing new definitions for variables, SynFuzz constructs the corresponding types of variables with random initialization if they are primitive types. However, for reference types (e.g., the `Object` type), it may be costly to construct their definitions due to the complex inheritance dependencies, and thus SynFuzz extracts the variable definitions with dependencies from the test program providing the code snippet and then also inserts them into the seed program.

2.2 MetaFuzz

MetaFuzz aims to bring the idea of metamorphic-based compiler testing to the new scenario. That is, MetaFuzz constructs a new test program that is equivalent to the given seed program through mutations. Then, it uses these equivalent test programs to test the new compiler. If their outputs under the same inputs are different, a compiler bug is detected. Hence, it can help detect miscompilation bugs without crashes (that cannot be detected by SynFuzz).

To get rid of the dependence on coverage analysis tools, MetaFuzz does not construct equivalent test programs under a given set of inputs like EMI-based techniques [28, 40], but constructs equivalent test programs under any inputs inspired by the existing technique (i.e., GLFuzz) [18] in the scenario of testing mature compilers. GLFuzz is an identical-equivalence-based technique, which was proposed to test graphic shader compilers by designing a set of identically equivalent mutation rules (also called semantic-preserving transformations). MetaFuzz substantially adapted GLFuzz to the new scenario according to the characteristics of the new compiler and the new-born programming languages. MetaFuzz removes the identically equivalent mutation rules designed in GLFuzz specific to graphic shader compilers, and systematically enriches the mutation rules for the new compiler.

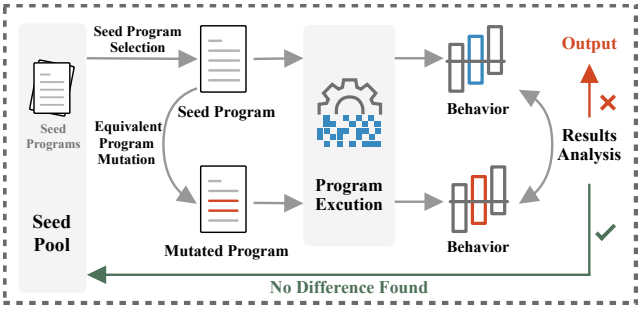


Figure 4: Overview of MetaFuzz

As shown in Figure 4, MetaFuzz contains three main steps: seed program selection (that selects a seed program from all the given seed programs), equivalent program mutation (that mutates the selected seed program to generate a set of equivalent test programs based on identically equivalent mutation rules), and program execution for testing (that compares the execution results of these equivalent test programs under the same inputs for testing the new compilers). Similar to SynFuzz, MetaFuzz randomly selects a seed program for the follow-up steps. In the following, we first introduce the identically equivalent mutation rules implemented

in MetaFuzz (Section 2.2.1), and then present the testing process based on equivalent test programs (Section 2.2.2).

2.2.1 Equivalent Program Mutation. Inspired by existing work [18], we systematically design and implement six identically equivalent mutation rules in MetaFuzz, which is also the main technical challenge of MetaFuzz. Table 2 shows the details of the six mutation rules, including a brief description and an illustrating example for each rule. The six mutation rules involve three different granularities, i.e., function granularity (Rule 1), block granularity (Rules 2-4), and expression granularity (Rules 5-6). All these mutation rules are applied to the AST corresponding to the selected seed program, which can not only alleviate the dependence on various program analysis tools (e.g., CFG analysis tools) but also increase the generality of these mutation rules since AST is fundamental and common for most of programming languages. In particular, Rules 1-2 are newly enriched by MetaFuzz while Rules 3-6 are adapted from the existing work [18].

During the mutation process, MetaFuzz randomly selects mutation rules from the whole set and applies them to the selected seed program together. Then, it generates an equivalent test program. For each equivalent mutation rule, not all the locations in the seed program are applicable. Hence, MetaFuzz first searches for all the applicable locations in the seed program for each selected mutation rule and then randomly selects a location to which the mutation rule is applied. For example, if a selected mutation rule is to invert an `if` condition (i.e., Rule 2), MetaFuzz first searches for all the `if` statements in the seed program and then randomly selects an `if` statement for mutation. If there is no `if` statement in the seed program, MetaFuzz has to skip this mutation rule.

In particular, the application of inserting an `if` or `loop` statement (Rules 3-4) is different, since MetaFuzz requires further selecting the code body for the `if` or `loop` statement (besides the mutation location). Following GLFuzz, if the condition of the `if` statement is guaranteed to evaluate to `false` at runtime or the `loop` times of the `loop` statement is guaranteed to evaluate to 0, it means that the code body will never be executed. Hence, MetaFuzz randomly selects a code snippet from the code snippet pool (built by SynFuzz presented in Section 2.1.1) as the code body for insertion, and ensures the validity of the mutated test program same as the practice of SynFuzz (presented in Section 2.1.2). If the condition of the `if` statement is guaranteed to evaluate to `true` or the `loop` times of the `loop` statement is guaranteed to evaluate to 1, it means that the code body will be executed definitely. Hence, MetaFuzz randomly selects a code snippet extracted from the selected seed program and then inserts the `if` or `loop` statement in the front of the code snippet by treating the code snippet as the code body. Please note that after mutation, the variables' definitions in the selected code snippet are changed to a new scope, resulting in no definitions for the subsequent variables' use in the seed program. Hence, for the variables' definitions in the code snippet, MetaFuzz takes them out of the new scope to maintain its original scope. For example, Figure 5 shows the pseudo-code of applying Rule 3 for mutation, where the selected `if` body is a SEQ snippet consisting of Lines 1-2 in Figure 5a. To ensure the validity of the mutated program, MetaFuzz takes the definition of `var1` outside of the `if` body, as shown in Figure 5b.

```
1 var1 <- "Init"        #define var1
2 println(var1)
3 if (var1 == null) { var1 <- "Init" } #access var1
```

(a) Source code of a selected SEQ snippet

```
1 var1 <- "Init"        #define var1
2 var2 <- expression
3 if (var2) {
4     println(var1)
5 }
6 if (var1 == null) { var1 <- "Init" } #access var1
```

(b) Mutant after applying Rule 3

Figure 5: Example of applying Rule 3

According to the above mutation process, MetaFuzz can apply several equivalent mutation rules together to construct high-order mutated test programs. For example, Rule 3 can be combined with Rule 5 to make the `if` condition more complicated. After the mutation process is completed on the selected seed program, a mutated test program equivalent to the seed program can be generated for testing the new compiler.

2.2.2 Program Execution for Compiler Testing. Based on a pair of equivalent test programs, MetaFuzz uses the new compiler to compile them and then executes them under the same inputs (that exist in the seed program). If they produce different outputs, it indicates that a potential compiler bug has been detected. To reduce false positives, MetaFuzz filters out the non-deterministic outputs (such as timestamps and random numbers) by identifying the corresponding keywords in both test programs and the produced output messages. Then, we manually check the remaining inconsistencies and report the potential bugs to developers. When identifying a false positive, we design a rule about it to help reduce this kind of false positive during subsequent testing. If the equivalent test programs produce the same outputs, MetaFuzz puts the mutated test program into the seed program pool for supporting more high-order mutations, which can also help improve the complexity of the generated test programs and thus boost the bug-revealing capability.

3 INDUSTRIAL EVALUATION

We conducted an industrial case study to investigate whether the compiler testing techniques (i.e., SynFuzz and MetaFuzz) adapted from the existing compiler testing practice in the scenario of testing mature compilers, can work well in the new scenario. Here, we deployed the two techniques to test the new compiler for a newborn programming language (a new multi-paradigm programming language) developed by a global IT company Huawei. Due to the company policy, we hide the names of the new-born programming language and the new compiler. For ease of presentation, we call the new compiler CompX in this paper.

CompX evolves monthly, and we used two relatively stable versions (i.e., versions 0.24.5 and 0.26.1) as the subjects in our study. Both versions contain many known bugs, which can help obtain the results with statistical significance. We also deployed both techniques to test the trunk of CompX in order to investigate whether it can detect previously unknown bugs, which can help complement the results obtained from the two historical versions.

Specifically, our industrial case study aims to address the following research questions:

Table 3: Summary of seed programs and extracted snippets

Seed Programs	LOC	Variables	Functions	Files	Size	Extracted snippet types				
						SEQ	LOOP	COND	Try-Catch	Total
Seeds-24	409,179	13,158	7,288	6,627	3,239	10,946	416	1,568	1,066	13,996
Seeds-26	428,575	12,422	7,650	6,867	3,407	11,683	434	1,156	1,709	14,982

- **RQ1**: How do SynFuzz and MetaFuzz perform compared with the industrial practice in terms of the number of detected bugs?
- **RQ2**: What is the overlap of the detected bugs by each studied technique?
- **RQ3**: Can SynFuzz and MetaFuzz detect previously unknown bugs in CompX?

3.1 Experimental Design

3.1.1 Studied Techniques. Besides SynFuzz and MetaFuzz, we also studied the current practice for testing CompX in Huawei and the general fuzzer (i.e., American Fuzzy Loop (AFL) [46]) for sufficient comparison in our study. The former is a generation-based compiler testing tool based on the grammar of the new programming language, called CompXFuzz, which is developed by themselves in Huawei. Similar to the idea of Csmith [43], CompXFuzz randomly generates a test program from scratch according to the grammar and treats the `main` function as the entry for generation. The latter (AFL) is a grey-box fuzzer that has been widely used in existing studies [22, 38]. In our scenario, AFL instruments all the branches of CompX to collect its coverage as the guidance of the testing process. Since the source code of CompX is confidential, we used the *qemu* mode of AFL for testing CompX by treating it as a binary program[1]. Same as SynFuzz, if a test program generated by CompXFuzz or AFL makes the compiler crash, it indicates that the test program detects a compiler bug. Overall, we investigated the performance of *synthesis-based*, *generation-based*, and *metamorphic-based* compiler testing through substantial adaptations to the new scenario, indicating the sufficiency of our industrial case study to some degree.

3.1.2 Seed Programs. Both SynFuzz and MetaFuzz rely on seed programs. For sufficient comparison with CompXFuzz, we directly used a set of test programs generated by CompXFuzz as seed programs. Please note that SynFuzz is not specific to the test programs generated by CompXFuzz and can also take other test programs (e.g., developer-written programs) as seeds, as mentioned before. Specifically, we used CompXFuzz to randomly generate a set of test programs, and then filtered out the test programs that were invalid (due to the implementation issue of CompXFuzz) or could trigger bugs on the compiler versions under test. Since the grammar of the new-born programming language evolves, we cannot use the same set of seed programs for both versions of CompX. Hence, we collected seed programs for the two versions, respectively. In total, we first generated 10,000 test programs for each version, and then after filtering, we collected 3,239 and 3,407 test programs as the seed programs of SynFuzz and MetaFuzz for versions 0.24.5 and 0.26.1, respectively. For ease of presentation, we call the two sets of seed programs *Seeds-24* and *Seeds-26*, respectively.

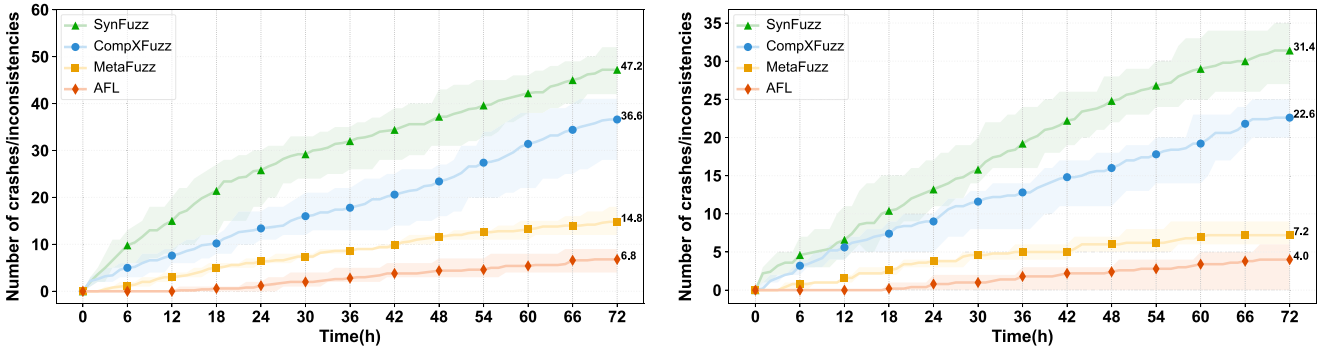
Table 3 presents the basic information of *Seeds-24* and *Seeds-26*, where Column *LOC* refers to the total number of lines of code for all the seed programs, Columns *Variables*, *Functions*, and *Files* refer to the total number of variables, functions, files in seed programs, and Column *Size* refers to the number of collected seed programs. Columns *SEQ*, *LOOP*, *COND*, and *Try-Catch* represent the total number of each type of code snippets extracted from the seed programs.

3.1.3 Metrics. We adopted the number of detected bugs to measure the effectiveness of each studied compiler testing technique. For fair comparison, we ran each studied technique for 3 days on each compiler version. To reduce the influence of randomness, each experiment was repeated 5 times independently with different random seeds. SynFuzz, CompXFuzz, and AFL detect bugs by checking whether the compiler crashes when compiling a test program. Following the existing work [47] and the suggestion from the CompX testing team, we used *crash messages* to determine the number of bugs from a set of bug-triggering test programs. Besides crash bugs, MetaFuzz also detects miscompilation bugs by comparing the outputs of a pair of equivalent test programs. Regarding such output inconsistencies, it is hard to automatically de-duplicate them, and thus the CompX testing team assists us in investigating them manually and determining the number of bugs from a set of test programs triggering output inconsistencies.

Regarding the bugs detected on the trunk of CompX, which tend to be previously unknown bugs, we reported them to the CompX testing team and determined the number of detected bugs according to their feedback.

3.1.4 Implementation and Environment. We implemented SynFuzz and MetaFuzz on the AST parser provided by Huawei. To balance the effectiveness and efficiency of test program generation, SynFuzz inserts five code snippets into a given seed program for generating a new test program, while MetaFuzz mutates the given seed program 20 times based on our designed equivalent mutation rules for generating one equivalent mutated program. All the experiments were conducted on a server with two dodeca-core CPUs Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz and 251GB RAM, running Ubuntu 18.04.4 LTS (64-bit). Due to the company policy, we cannot release our experimental data and the implementations of SynFuzz and MetaFuzz before completing the confidential checking process in Huawei.

[1]Based on the *qemu* mode of AFL, it is possible to obtain binary-level coverage for each test program, which may enable EMI-based compiler testing in the new scenario. However, EMI-based techniques perform mutations on source code, and it is challenging to map binary-level coverage to source code for a new programming language due to incomplete toolchains.



(a) CompX 0.24.5 (b) CompX 0.26.1

Figure 6: Comparison of the number of bugs detected over time

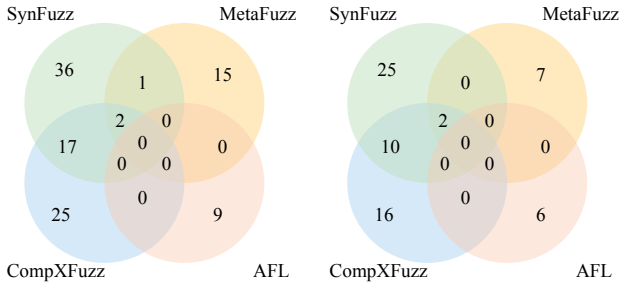
3.2 Results and Analysis

3.2.1 RQ1: Number of Detected Bugs. Figure 6 shows the comparison results among SynFuzz, MetaFuzz, CompXFuzz, and AFL in terms of the number of detected bugs, where the x-axis represents the testing time, and the y-axis represents the number of detected bugs. For each technique, the solid line presents the average result of five repeated experiments, while the shading area shows the overall results between maximum and minimum across the five repeated experiments. From Figure 6, SynFuzz almost always detects more bugs than the other three studied techniques on each compiler version. Specifically, in terms of the total number of bugs across the five repeated experiments on both versions after de-duplication, SynFuzz detects 93 bugs while MetaFuzz, CompXFuzz, and AFL detect 27, 72, and 15 bugs, respectively. The improvements of SynFuzz over CompXFuzz and AFL are 29.17% and 520.00%, respectively. Regarding MetaFuzz, it performs worse than CompXFuzz and SynFuzz in terms of the total number of detected bugs on both versions, but still achieves an 80.00% improvement over AFL. In particular, we performed the Mann-Whitney U-test [3] at the significance level of 0.05 for each pair of techniques to check whether there is a significant difference between them in statistics in terms of the number of detected bugs. We found that all the p-vales are smaller than 0.01, demonstrating the statistical significance of our conclusions.

We then analyzed the reason why SynFuzz outperforms the current practice for testing CompX in Huawei (i.e., CompXFuzz). CompXFuzz constructs test programs from scratch based on the grammar of the new-born programming language, which requires constructing a large number of program elements and maintaining complex dependencies between elements. However, SynFuzz inserts some code snippets into a given seed program, and thus just needs to consider the code snippets and their affected code in the seed program to ensure the validity of the generated test program. By comparing the test program generation process of SynFuzz and CompXFuzz, we can find that the generation efficiency of SynFuzz is higher than that of CompXFuzz. That is, during the same testing time, SynFuzz generates much more test programs for compiler testing than CompXFuzz. For example, on version 0.26.1, SynFuzz generates and runs 59,932 test programs on average across the five repeated experiments, while CompXFuzz just generates and runs 40,699 test programs for testing on average. In fact, SynFuzz takes the test programs generated by CompXFuzz as the seed programs in our study for sufficient comparison with CompXFuzz, and thus SynFuzz and CompXFuzz have almost the same input space. Due to the high efficiency of SynFuzz, it can explore the input space faster, leading to better bug detection effectiveness during the same testing time. This conclusion is also confirmed by the existing study [8], which shows that efficiency is the most important factor that affects the effectiveness of a compiler testing technique.

We further analyzed why MetaFuzz was unable to detect as many bugs as SynFuzz and CompXFuzz did, which also mainly lies in the relatively low efficiency of MetaFuzz. MetaFuzz tests the CompX by mutating a given seed program to an equivalent test program and then running the pair of equivalent test programs. Compared with both SynFuzz and CompXFuzz, MetaFuzz needs to compile and execute a pair of test programs (rather than only one test program) each time. It incurs more costs than SynFuzz and CompXFuzz, leading to generating and running fewer test programs during the same testing time. For example, on version 0.26.1, MetaFuzz just generates 27,955 test programs on average across the five repeated runs. Furthermore, as demonstrated by the existing studies [9, 20, 29, 41], such metamorphic-based compiler testing is good at detecting bugs in compiler optimizations. However, the current core functionality of CompX, as a new compiler for a new-born programming language, still lies in the non-optimization part. The current CompX just implements some simple optimizations, which also limits the effectiveness of MetaFuzz. This is also the reason why SynFuzz does not adopt Different Optimization Levels (DOL, a kind of differential testing that compares the results produced by one compiler under different optimization levels on a given test program) as the test oracle. More specifically, DOL is limited by such simple optimizations in effectiveness but incurs much overhead on testing for each test program. With the functionality of optimizations improving, we can incorporate DOL for effectiveness improvement in the future.

We also analyzed why AFL always detected the fewest bugs among all the studied techniques on both versions. The reasons are twofold. The first one also lies in the relatively low efficiency. AFL collects test coverage of the CompX by instrumenting the corresponding binary program (as presented in Section 3.1.1), which incurs extra overhead during the testing process. For example, on version 0.26.1, AFL only generates and runs 8,159 test programs



(a) CompX 0.24.5 **(b) CompX 0.26.1**

Figure 7: Bug overlap analysis

on average across the five repeated runs. The second reason is that most of the test programs generated by AFL are syntactically invalid. Specifically, the mutation rules in AFL do not conform to the language specifications and can damage the structure of the seed program, causing that most of the generated programs are invalid. It can negatively affect testing efficiency. For example, among the test programs generated by AFL on version 0.26.1, only 812 test programs were free of syntax errors on average, indicating that 90.04% of the test programs generated by AFL are syntactically invalid. This also demonstrates the importance of designing the fixing strategy in SynFuzz to ensure the validity of generated test programs.

3.2.2 RQ2: Bug Overlap Analysis. We further analyzed the overlap of bugs detected by the four studied techniques. Here, for each technique, we put all the detected bugs across the five repeated runs together and then de-duplicated them based on the method mentioned in Section 3.1.3 for bug overlap analysis. The Venn diagrams in Figure 7 show the bug overlap analysis results on each compiler version. From Figure 7, each studied technique can detect some unique bugs, which refer to the bugs that are detected by the technique but are not detected by the other three techniques. In total, on the two versions, SynFuzz, MetaFuzz, CompXFuzz, and AFL detect 61, 22, 41, and 15 unique bugs in total, respectively. That is, *SynFuzz and MetaFuzz are complementary with CompXFuzz and AFL to a large extent for testing CompX.*

We further explained why SynFuzz and MetaFuzz can complement the current practice (i.e., CompXFuzz) for testing CompX. As presented above, the high efficiency of SynFuzz makes a major contribution to its largest number of unique bugs. Through deeper analysis, we found that many of the unique bugs detected by SynFuzz were triggered by complex code structures. For example, SynFuzz inserts an internal function and its invocation into a hard-to-reach code area as shown in Figure 3 (i.e., a combination of `if` and `try-catch` statement). However, generating such code structures is very inefficient for CompXFuzz, since it generates test programs within the huge (even infinite) search space, but SynFuzz just explores the space defined by the prepared seed programs through code snippet synthesis. Actually, the larger search space of CompXFuzz than SynFuzz contributes to the unique bugs detected by CompXFuzz to some degree.

For MetaFuzz, its test oracle based on a pair of equivalent test programs makes the contribution to the unique bugs detected by it. Specifically, all the other three techniques can only detect crash bugs, while MetaFuzz can detect both crash bugs and miscompilation bugs. The reason why MetaFuzz can detect crash bugs lies in that it requires compiling equivalent test programs before checking equivalence during runtime. Although we guarantee all the seed programs can be compiled and executed successfully, the mutated test programs may trigger compilation crashes. The overlapped bugs between SynFuzz, CompXFuzz, and MetaFuzz are exactly crash bugs. Moreover, through further investigation, we found that our designed mutation rules indeed help construct complex code structures in mutated test programs, which is also helpful to improve the bug-triggering capabilities of generated test programs.

In particular, the bugs detected by AFL were not detected by any of the other three techniques. The reason lies in the mutation rules in AFL. They mutate the given seed program by bytes and often produce uncertain characters, causing that all the bugs detected by AFL are triggered by illegal characters. However, the other three techniques cannot introduce illegal characters in the generated test programs, and thus they cannot detect these bugs.

3.2.3 RQ3: Previously Unknown Bugs. In this experiment, we kept almost the same experimental settings as the other experiments, but collected the seed programs on the trunk of CompX due to the grammar update and ran each technique for a longer testing time (i.e., ten days). Since the CompX testing team also uses CompXFuzz to test the trunk of CompX, we just reported the *unique bugs* detected by SynFuzz or MetaFuzz to the CompX testing team after our careful analysis. In total, we submitted 11 bugs, 11 of which have been confirmed and fixed by the developers. In particular, five of these bugs were marked as *critical* defects. Here, we use two previously unknown bugs as examples for further illustration.

Figure 3 shows the pseudo-code of a test program generated by SynFuzz, which makes CompX crash. The synthesis process of this test program has been described in Section 2.1.2, and we further explain how it triggers the compiler bug. In this example, the coalescing operator `??` at Line 8 joins an `if` expression with a `try-catch` expression to form a coalescing expression. In particular, the result of coalescing expression depends on the return type of the `if` expression (each expression in the new language has a return type). If the return type of `if` expression is `Result<Bool>.Ok(var1)` (Line 5), the result of coalescing expression is the same as the result of `if` expression (i.e., `var1`); Otherwise, the result of coalescing expression is the same as the execution results of `try-catch` expression. In the body of the `try-catch` expression, an internal function (i.e., `func1`) is inserted at Lines 9-11 and called at Line 13. This test program is syntactically correct and should be compiled normally, however, CompX failed to generate the internal function in the try body after the coalescing operator `??` due to a code generation bug, and crashed during compilation. Please note that neither the seed program nor the test program from which the code snippet is extracted can detect this bug. We put the foreign code snippet into a new context and make them interact, which helps to trigger a new program execution path and thus makes CompX crash.

Figure 8 shows the pseudo-code of a test program generated by MetaFuzz, which makes CompX throw a code generation error during compilation. In this example, MetaFuzz applies two mutation rules on function `func1`. MetaFuzz first applies the *Insert an if statement* rule (i.e., Rule 3) at Line 6, and the `if` condition was

```
1  func func1(){
2
3    loopCount <- (1)  #insert a loop once statement
4    for (i in loopCount) {
5      let var1 : Bool <- false expression
6      if (var1) {       #insert an if statement
7        (match (i) {
8          case _ =>
9            let var2 : Result<Unit> = Result<Unit>.Ok(());
10           var2
11       } ?? func2())
12     }
13   }
14   ... ...
15 }
```

Figure 8: Bug #I4YN50

evaluated to `false`. Since the `if` code body will not be executed, MetaFuzz randomly selects a code snippet from the snippet pool and inserts it at Line 7. The selected code snippet in this example is a SEQ code snippet (i.e., Lines 7-11), which is a coalescing expression consisting of a `match` expression and a function call (i.e., `func2`, the deceleration of `func2` is omitted due to the limited space). Then MetaFuzz applies the *Insert a loop statement* rule (i.e., Rule 4) on `func1` at Line 4, and selects the inserted `if` statement as the `loop` body, where the `loop` count was evaluated to 1 (indicating that the `loop` body will always be executed). This example is syntactically correct and should be compiled normally, however, CompX failed to compile this example and threw a code generation error. This bug cannot be detected by the other three techniques.

4 DISCUSSION

4.1 Lessons Learned

Validity of test programs matters. As shown in RQ1 and RQ2, AFL detects the fewest compiler bugs compared with the other three techniques. One major reason lies in that most of the test programs generated by AFL are syntactically invalid. This points out the importance of the validity of generated test programs. That is, for structural test inputs, it is very critical to guarantee the generated inputs conform to the corresponding specification (e.g., the programming-language syntactic constraints for test programs). Indeed, the invalid test programs generated by AFL also help detect some unique bugs. By considering the overall testing effectiveness, we could consider constructing invalid test programs for testing as complementary after sufficient testing with valid test programs in the future.

Cost of test program generation matters. SynFuzz is more efficient in generating test programs than the other three techniques, and thus detects more bugs than them during the same testing time. This points out that efficiency is an indispensable factor in achieving better compiler-testing effectiveness. This also demonstrates the contribution of SynFuzz by adapting synthesis-based compiler testing to the new scenario.

Diversity of testing techniques matters. From RQ2, all the studied compiler testing techniques can detect a number of unique bugs, indicating that they are complementary in detecting compiler bugs. This shows that it is useful to integrate these techniques to achieve better testing effectiveness in practice. This also demonstrates the contribution of adapting synthesis-based, generation-based, and metamorphic-based techniques to the new scenario, rather than just considering one kind of testing technique.

4.2 Suggestions

Suggestions for practical use. Based on our industrial evaluation, these studied techniques can complement each other, and thus it is important to use all of them in practice. By considering their testing efficiency and effectiveness (especially the unique value), we suggest to apply these compiler testing techniques to the compiler under test *as the priority of SynFuzz, CompXFuzz, MetaFuzz, and AFL* according to the given testing time.

Although SynFuzz and MetaFuzz target the CompX compiler, the ideas of them are general and can be generalized to other similar programming languages, since the programming language features involved in SynFuzz (i.e., the designed code-snippet types) and MetaFuzz (i.e., the equivalent mutation rules) have similar substitutes in many programming languages. Regarding the unique programming language features, they can be extended by implementing corresponding code-snippet extraction and synthesis strategies as well as mutation rules following the high-level ideas adopted in SynFuzz and MetaFuzz. In particular, our industrial evaluation has demonstrated the effectiveness of them. Therefore, when new compilers for other new-born programming languages are developed, they can also be adapted to test them, especially at the initial stage, since they do not have various dependence on advanced program analysis tools (e.g, coverage analysis tools and CFG analysis tools) and reference compilers. Indeed, both SynFuzz and MetaFuzz have been deployed in Huawei to test CompX in practice due to these advantages of them.

Suggestions for future research. First, although we made the first exploration to adapt advanced compiler testing practices in the scenario of testing mature compilers (i.e., synthesis-based and metamorphic-based compiler testing) to fit the new scenario, there is still room to further improve the effectiveness of them. In particular, both SynFuzz and MetaFuzz involve random selection, such as random code-snippet selection, random seed program selection, and random mutation rule selection. For the enormous search space, the random search method could be inefficient, and meanwhile some existing work has demonstrated the effectiveness of more advanced search methods in the area of software testing [11, 16, 17, 44, 48]. Therefore, it is promising to further improve both SynFuzz and MetaFuzz by integrating more advanced search algorithms (such as reinforcement learning [26] or genetic algorithm [14, 45]).

Besides, both SynFuzz and MetaFuzz are complementary, and thus it could be promising to integrate them into one more effective technique. A simple way is to take the test programs generated by SynFuzz as the seed programs of MetaFuzz. Better integration methods (even with CompXFuzz due to the complementarity) can be explored in the future.

4.3 Threats to Validity

The *internal* threat to validity mainly lies in the implementations of SynFuzz and MetaFuzz. To reduce this threat, three authors carefully checked all the code.

The *external* threat to validity mainly lies in the subjects used in our study. Here, we just evaluated SynFuzz and MetaFuzz on two versions of CompX in one company (i.e., Huawei). Actually, putting

one technique to practice on one product of a company has been very challenging, since it has a very high requirement for effectiveness. Our industrial case study demonstrated the effectiveness and practicability of our adapted techniques. In the future, we will try to extend them to more compilers in other companies.

The *construct* threat to validity mainly lies in the randomness involved in these techniques. To reduce the influence of randomness, we conducted the study on two versions by repeating each experiment five times independently with different random seeds and performed a Mann-Whitney U-test to confirm the statistical significance of our conclusions, as suggested by the existing work [27].

5 RELATED WORK

The goal of our work is to conduct an industrial case study to investigate the effectiveness of the existing compiler testing techniques in the new scenario. Here, we substantially adapted grammar-based compiler testing and metamorphic-based compiler testing in the scenario of testing mature compilers to fit the new scenario. Therefore, our related work consists of two parts: the existing compiler testing techniques and the existing compiler testing studies.

Compiler Testing Techniques. In our work, we designed and implemented SynFuzz and MetaFuzz inspired by grammar-based compiler testing (especially synthesis-based compiler testing) [24, 47] and metamorphic-based compiler testing [18, 20]. We have introduced the representative grammar-based techniques (i.e., LangFuzz [24] and JavaTailor [47]) in Section 2.1 and the representative metamorphic-based technique with identically equivalent mutation (i.e., GLFuzz [18]) in Section 2.2. Besides, there are some other compiler testing techniques [4, 5, 15, 23, 31, 33, 35, 39]. For example, Yang et al. [43] proposed Csmith, which is a generation-based technique that randomly constructs C programs from scratch by considering a number of C features. Chen et al. [10] proposed HiCOND, which boosts generation-based techniques by mining historical data to guide the configurations of test program generation tools. In particular, our industrial case study also investigated the effectiveness of generation-based compiler testing by studying CompXFuzz, which is developed by the CompX testing team following the idea of generation-based compiler testing. EMI-based compiler testing is also widely studied in the literature [12, 13, 41, 42], which belongs to the category of metamorphic-based techniques [9]. Different from GLFuzz and MetaFuzz, EMI-based compiler testing aims to construct equivalent test programs under a given set of inputs of the seed program. Based on the idea of EMI, there are several instantiations, such as Orion [28], Athena [29], and Hermes [40]. However, EMI-based techniques heavily depend on program analysis tools (especially coverage analysis tools), which limits the application of them in the new scenario as explained before.

Our work does not aim to propose novel compiler testing techniques, but adapts the existing ones to the new scenario for empirical investigation, and then obtains insightful findings for future research and practical use. And an implicit goal for our work is to borrow the power from existing researches to test the new compiler well. Hence, in the future, we can improve SynFuzz and MetaFuzz by referring to more advanced strategies in the literature.

Compiler Testing Studies. In the literature, there are some empirical studies on compiler testing. For example, Chen et al. [8] empirically compared three compiler testing techniques, i.e., RDT (randomized differential testing), DOL, and EMI, on two mainstream open-source C compilers (i.e., GCC and LLVM), and delivered a series of findings. Lidbury et al. [30] investigated the effectiveness of RDT and EMI on OpenCL compilers. They lifted RDT by building a tool CLsmith on the basis of Csmith [30] and lifted EMI by injecting dead-by-construction code. Actually, MetaFuzz also includes injecting dead-by-construction code as one of the mutation rules inspired by it and GLFuzz. Donaldson et al. [19] summarized the experience of applying randomized metamorphic compiler testing to the testing of graphic shader compilers in production. They enhanced the quality of Vulkan Conformance Test Suite with the test programs generated by GraphicFuzz (originally called GLFuzz) that expose bugs or provide additional coverage, and also discussed the practical experience and new directions for future research.

Different from them, we conducted an industrial case study to investigate the effectiveness of grammar-based and metamorphic-based compiler testing in the scenario of testing new compilers of new-born programming languages. According to the characteristics of the new scenario, we implemented SynFuzz and MetaFuzz through substantial adaptations inspired by grammar-based and metamorphic-based compiler testing. Further, we summarized lessons and suggestions from the industrial study for promoting future research and practical guidelines in the new scenario.

6 CONCLUSION

In this work, we made the first exploration to test a new compiler for a new-born programming language. Specifically, we substantially adapted the notable grammar-based and metamorphic-based compiler testing in the scenario of testing mature compilers to the new scenario. Inspired by grammar-based compiler testing (especially synthesis-based compiler testing), we design SynFuzz, which generates a new test program by synthesizing a set of code snippets extracted from existing test programs with the assistance of the grammar and adopts compilation crash as test oracle due to lack of reference compiler for differential testing. Inspired by metamorphic-based compiler testing, we design MetaFuzz, which generates a new equivalent test program by performing identically equivalent mutations and thus gets rid of the dependence on coverage analysis tools. We conducted an industrial case study to evaluate both SynFuzz and MetaFuzz on two versions of CompX in Huawei by comparing with the practice adopted by the CompX testing team (that is inspired by generation-based compiler testing) and AFL. The results demonstrate the effectiveness of SynFuzz and the complementary of SynFuzz and MetaFuzz. They have also detected 11 previously unknown bugs in total, all of which have been fixed by developers. Due to their effectiveness, they have been deployed to testing CompX in Huawei. In particular, we delivered a series of lessons and suggestions for practical use and future research from the industrial case study.

ACKNOWLEDGMENT

This work was partially funded by the National Natural Science Foundation of China 62002256, 62232001, and 61872263.

REFERENCES

[1] 2023. GCC. http://gcc.gnu.org/.

[2] 2023. LLVM. https://llvm.org/.

[3] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE'11)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[4] Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: program synthesis by reasoning over graphs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29.

[5] Abdulazeez S. Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Inf. Softw. Technol.* 39, 9 (1997), 617–625.

[6] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 700–711.

[7] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 223–234.

[8] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 180–190.

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36.

[10] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 305–316.

[11] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1198–1209.

[12] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020).

[13] Tsong Yueh Chen, T. H. Tse, and Zhiquan Zhou. 2003. Fault-based testing without the need of oracles. *Inf. Softw. Technol.* 45, 1 (2003), 1–9.

[14] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Inf. Softw. Technol.* 93 (2018), 1–13.

[15] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 197–208.

[16] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*. 1257–1268.

[17] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 482–493.

[18] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29.

[19] Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:29.

[20] Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*. ACM, 44–47.

[21] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, Luca de Alfaro and Jens Palsberg (Eds.). ACM, 255–264.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association.

[23] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 621–631.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458.

[25] Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. 2016. RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Softw. Pract. Exp.* 46, 3 (2016), 405–431.

[26] Junhwi Kim, Minhyuk Kwon, and Shin Yoo. 2018. Generating test input with deep reinforcement learning. In *Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, Juan Pablo Galeotti and Alessandra Gorla (Eds.). ACM, 51–58.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. *CoRR* abs/1808.09700 (2018).

[28] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226.

[29] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 386–399.

[30] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 65–76.

[31] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.

[32] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53.

[33] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 21–30.

[34] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346.

[35] Masataka Sassa and Daijiro Sudosa. 2006. Experience in Testing Compiler Optimizers Using Comparison Checking. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, Hamid R. Arabnia and Hassan Reza (Eds.). CSREA Press, 837–843.

[36] Sabrina Souto and Marcelo d'Amorim. 2018. Time-space efficient regression testing for configurable systems. *J. Syst. Softw.* 137 (2018), 733–746.

[37] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 632–642.

[38] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 244–256.

[39] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 203–213.

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

[41] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, Jun Han and Tran Dan Thu (Eds.). IEEE Computer Society, 270–279.

[42] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1 (2022), 15:1–15:28.

[43] 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.

[44] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated

conformance testing for JavaScript engines via deep compiler fuzzing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 435–450.

[45] Tingting Yu, Witawas Srisa-an, Myra B. Cohen, and Gregg Rothermel. 2018. A hybrid approach to testing for nonfunctional faults in embedded systems using genetic algorithms. *Softw. Test. Verification Reliab.* 28, 7 (2018).

[46] Michal Zalewski. 2023. american fuzzy lop (2.52 b). *Retrieved April* 10 (2023).

[47] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1133–1144.

[48] Yingquan Zhao, Zan Wang, Shuang Liu, Jun Sun, Junjie Chen, and Xiang Chen. 2023. Achieving High MAP-Coverage Through Pattern Constraint Reduction. *IEEE Trans. Software Eng.* 49, 1 (2023), 99–112.