Java JIT Testing with Template Extraction

ZHIQIANG ZANG, The University of Texas at Austin, USA
FU-YAO YU, The University of Texas at Austin, USA
ADITYA THIMMAIAH, The University of Texas at Austin, USA
AUGUST SHI, The University of Texas at Austin, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

We present LeJit, a template-based framework for testing Java just-in-time (JIT) compilers. Like recent template-based frameworks, LeJit executes a template—a program with holes to be filled—to generate concrete programs given as inputs to Java JIT compilers. LeJit automatically generates template programs from existing Java code by converting expressions to holes, as well as generating necessary glue code (i.e., code that generates instances of non-primitive types) to make generated templates executable. We have successfully used LeJit to test a range of popular Java JIT compilers, revealing five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM. All of these bugs have been confirmed by Oracle and IBM developers, and 11 of these bugs were previously unknown, including two CVEs (Common Vulnerabilities and Exposures). Our comparison with several existing approaches shows that LeJit is complementary to them and is a powerful technique for ensuring Java JIT compiler correctness.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: Testing, test generation, compilers, templates, template generation

ACM Reference Format:
Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT Testing with Template Extraction. *Proc. ACM Softw. Eng.* 1, FSE, Article 51 (July 2024), 23 pages. https://doi.org/10.1145/3643777

1 INTRODUCTION

Compilers are the cornerstone of software development, and their correctness is of utmost importance. For years, the compiler testing community has primarily focused on static compilers, such as GCC, LLVM, and javac. Tools like Csmith [46] and Hephaestus [5] have been shown effective in discovering bugs in static compilers [47, 48]. However, these tools are ineffective in discovering bugs in just-in-time (JIT) compilers. JIT compilers, or JIT for short, dynamically (i.e., at runtime) rewrite parts of programs to optimize program execution based on profiling data. Testing such compilers requires carefully crafted inputs that trigger JIT compilation and provide challenging code snippets for the optimizing compilers.

Recently, there has been substantial work on testing JIT compilers, which can be organized in three main categories: grammar-based, mutation-based, and template-based techniques. The first group [1, 35, 50] generates test inputs based on language grammars. The second group [7, 8, 43, 53]

Authors' addresses: Zhiqiang Zang, The University of Texas at Austin, Austin, USA, zhiqiang.zang@utexas.edu; Fu-Yao Yu, The University of Texas at Austin, Austin, USA, fu.yao.yu@utexas.edu; Aditya Thimmaiah, The University of Texas at Austin, Austin, USA, auditt@utexas.edu; August Shi, The University of Texas at Austin, Austin, USA, august@utexas.edu; Milos Gligoric, The University of Texas at Austin, Austin, USA, gligoric@utexas.edu.

ACM 2994-970X/2024/7-ART51
https://doi.org/10.1145/3643777

commonly starts with a set of seed programs that are evolved with a predefined set of mutation operators. The third group [9, 51, 52] combines manual testing (writing templates) with fuzzing, i.e., filling holes in templates based on a set of manually predefined choices that can be different for each hole.

These three groups are complementary and each group provides some benefits over the others. Consider JAttack [51], a template-based technique that recently discovered two CVEs (Common Vulnerabilities and Exposures) in Oracle's JIT. Each *template program* is a valid Java program with holes, and each hole is written in an embedded domain-specific language that specifies the set of expressions that can potentially fill the hole. JAttack generates programs by fuzzing holes during the execution of a given template. The advantage of this technique is that developers have full control of the space that should be tested and the way programs should be modified. At the same time, JAttack requires substantial developers' engagement, as both template program design and hole values are written manually.

In this paper, we present a novel framework, dubbed LeJit, for automatically generating template programs from existing code. LeJit generates templates by rewriting existing expressions to holes, as well as generating necessary glue code (e.g., code that creates instances of non-primitive types on which methods can be invoked) to make those templates executable. Execution of generated templates, which randomly fills the holes, creates concrete programs that are used as inputs for Java JIT testing. As a result, LeJit sits in between mutation-based techniques and template-based techniques. Unlike existing template-based techniques, templates are automatically extracted from any existing code. Unlike mutation-based techniques, each hole has its own set of values and can be filled dynamically (rather than statically), and multiple holes are filled simultaneously during the execution of the template. Subsequently, LeJit is in a way similar to higher order mutation [21], but holes are filled dynamically through executing templates.

LeJit is designed to enable generation of a program template from any existing method. One of the key challenges was to enable templates for methods that accept instances of complex types as arguments, including an instance on which an instance method is to be invoked. Our key insight in this direction is to capture instances of various types during testing of methods from which templates are to be extracted; tests used can be either existing manually-written tests or automatically-generated tests.

Unlike several existing tools for testing Java runtime environments [18], LeJit generates source code rather than bytecode. Some advantages of focusing on source code rather than bytecode include: 1) eliminating the need to worry about invalid classfiles, as those obtained from source files always pass the early check of the class format performed by bytecode verifiers, allowing for "deeper" testing; 2) simplifying every step during the bug reporting phase: a bug reported as a source code snippet instead of bytecode is easier to understand, minimize, and report, and it also facilitates confirmation, fixing, and integrating in test suites by compiler developers; 3) decreasing the likelihood of revealed bugs resulting in false positives, since these programs result in valid bytecode generated via a Java compiler as opposed to random sequences of bytecode instructions.

We used LeJit to test several JIT compilers: Oracle HotSpot, IBM OpenJ9, and Oracle GraalVM. We used differential testing [26] to detect crash and inconsistency between JIT compilers. We extracted templates from ten open-source Java projects available on GitHub, although our technique can extract templates from any other code. Our runs discovered five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM. 11 out of the 15 bugs were previously unknown, including two CVEs. All bugs have been confirmed by compiler developers.

We further compared LeJit with JITfuzz [44] and JavaTailor [53], the state-of-the-art testing tools for Java JIT and JVMs, respectively. Our experiments show that LeJit increased code coverage of C1 compiler by 8.0% and C2 compiler by 8.2% [28] compared to JITfuzz, and increased by 3.3%

```
1 package org.apache.commons.text;...
2 public class StrBuilder implements ... { ...
3   static final int CAPACITY = 32;
4   char[] buffer; private int size;
5   private String newLine, nullText;
6   public StrBuilder(final String str) {
7      if (str❶ == null) { buffer = new char[CAPACITY❷]; } ... }
8   public StrBuilder trim() {
9     if (size == 0❸) { return this; }
10    int len = size❹;
11    final char[] buf = buffer❺;
12    int pos = 0❻;
13    while (pos < len && buf[pos] <= ' '❼) { pos++; }
14    while (pos < len && buf[len - 1] <= ' '❽) { len--; } ...
15    return this; } }
```

Fig. 1. An existing program from the text project [40] used as a source for template extraction.

and 4.0% compared to JavaTailor, when testing OpenJDK HotSpot. Additionally, using JITfuzz and JavaTailor we have not discovered any of the bugs found by LeJit.

The main contributions of this paper include:

- **Framework**. We designed and implemented a framework for extracting templates from existing code by converting expressions into holes and capturing instances of complex types during test execution. Captured instances enable execution of templates that produce concrete programs used as inputs for compiler testing.
- **Implementation**. We have implemented LeJit for Java and built it around a recent publicly available framework (JAttack). We have also developed several variants of LeJit to help us understand the benefits of templates and captured instances used for arguments.
- **Evaluation**. We have performed extensive evaluation of LeJit. We have extracted 143,195 templates from ten open-source Java projects on GitHub. We then used JAttack to generate 886,178 concrete programs. We have used the generated programs to test three compilers – Oracle HotSpot, IBM OpenJ9, and Oracle GraalVM. Additionally, we compare LeJit with JITfuzz and JavaTailor the state-of-the-art tools for testing Java runtime environments.
- **Analysis**. We performed an in-depth analysis of templates and generated programs to understand how the presence of various Java language features, e.g., arrays, conditional statements, loops, etc., affect LeJit's bug detection capabilities. We also studied the impact of various types of templates on the result, and we find types of holes that play an important role in bug detection.
- **Results**. Our results show the effectiveness of LeJit. We have discovered 15 bugs, including five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM. 11 of the bugs are previously unknown, including two CVEs. All bugs have been confirmed by compiler developers. Our results also show that LeJit is complementary to the state-of-the-art techniques, which did not discover any of the bugs found by LeJit.

Our tool is publicly available at https://github.com/EngineeringSoftware/lejit.

2 EXAMPLE

We demonstrate the capabilities of LeJit, using an example program that involves a bug we detected in the OpenJ9 JIT compiler. Figure 1 shows a snippet of the example program.

```
1 package org.apache.commons.text;...
2 import jattack.annotation.*;
3 import static jattack.Boom.*;
4 public class StrBuilder implements ... { ...
5   static final int CAPACITY = 32;
6   char[] buffer; private int size;
7   private String newLine, nullText;
8   public StrBuilder(final String str) {
9     if (refId(String.class).eval()❶ == null) {
10      buffer = new char[intId().eval()❷]; } ... }

12  @Entry
13  public StrBuilder trim() {
14    if (relation(intId(), intVal()).eval()❸) {
15      return this; }
16    int len = intId().eval()❹;
17    final char[] buf = refId(char[].class).eval()❺;
18    int pos = intVal().eval()❻;
19    int _lim1 = 0;
20    while (logic(
21             relation(intId(), intId()),
22             relation(charArrAcc(
23                        refId(char[].class),
24                        intId()),
25                      charVal()))
26           .eval()❼ && _lim1++ < 1000) {
27      pos++; }
28    int _lim2 = 0;
29    while (logic(
30             relation(intId(), intId()),
31             relation(charArrAcc(
32                        refId(char[].class),
33                        arithmetic(intId(),
34                                   intVal())),
35                      charVal()))
36           .eval()❽ && _lim2++ < 1000) {
37      len--; } ...
38    return this; }

40  @Argument(0)
41  public static StrBuilder _arg0() {
42    StrBuilder sb1 = new StrBuilder("date");
43    sb1.append((Object) 10.0);
44    sb1.appendSeparator("d");
45    Object[] arr = new Object[] { 1.0 };
46    return sb1.append("resourceBundle", arr); } }
```

(a)

```
1 package org.apache.commons.text;...
2 import jattack.annotation.*;
3 import jattack.csutil.Helper;
4 import jattack.csutil.checksum.WrappedChecksum;
5 import jattack.exception.UnfilledHoleException;
6 import static jattack.Boom.*;
7 public class StrBuilder implements ... { ...
8   static final int CAPACITY = 32;
9   char[] buffer; private int size;
10  private String newLine, nullText;
11  public StrBuilder(final String str) {
12    if (nullText❶ == null) {
13      buffer = new char[CAPACITY❷]; } ... }

15  public StrBuilder trim() {
16    if (size <= -18387848538) { return this; }
17    int len = CAPACITY❹;
18    final char[] buf = buffer❺;
19    int pos = 809931165❻;
20    int _lim1 = 0;
21    while ((len > _lim1 && buf[size] != 'Z')❼
22            && _lim1++ < 1000) { pos++; }
23    int _lim2 = 0;
24    while ((pos <= _lim2
25              || buf[size - 1312433786] > '0')❽
26            && _lim2++ < 1000) { len--; } ...
27    return this; }

29  public static StrBuilder _arg0() {
30    StrBuilder sb1 = new StrBuilder("date");
31    sb1.append((Object) 10.0);
32    sb1.appendSeparator("d");
33    Object[] arr = new Object[] { 1.0 };
34    return sb1.append("resourceBundle", arr); }

36  public static void main(String[] args) {
37    WrappedChecksum cs = new WrappedChecksum();
38    StrBuilder rcvr = _arg0();
39    cs.update(rcvr);
40    for (int i = 0; i < 100_000; ++i) {
41      try {
42        cs.update(rcvr.trim());
43      } catch (UnfilledHoleException e) {
44        throw e;
45      } catch(Throwable e) {
46        cs.update(e.getClass().getName()); } }
47    cs.update(StrBuilder.class);
48    Helper.write(cs.getValue()); } }
```

(b)

Fig. 2. An example (a) template extracted from the program in Figure 1, and (b) a concrete program generated from the template by filling in the holes, which crashed OpenJ9 JIT compiler.

LeJit extracts a template from this program by replacing expressions with holes, as shown in Figure 2a. A *hole* is a placeholder to be filled with concrete expressions during program generation. Each hole is expressed as an API call, which defines the type and range of values that can be used to fill the hole, e.g., the first hole `refId(String.class)` (line 9) in the template represents any available variable with type `String` at this execution point [51].

There are eight holes displayed in the template, two in the constructor and six in the method `trim`. Each hole, labeled with a circled number, is converted from the expression in the original program

with the same number. For example, the first hole `refId(String.class)` is converted from the local variable `str` (line 7) in Figure 1. The next hole `intId()` (line 10) in Figure 2a, which represents any available `int` variable, is converted from the `int` field `CAPACITY` (line 7) in Figure 1. The third hole (line 14) in Figure 2a, represents a relational expression that connects an integer variable and an integer literal (between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`) using a relational operator (`<`, `<=`, `>`, `>=`, `==`, `!=`). This hole is converted from the `if` condition `size == 0` (line 9) in Figure 1. Similarly, the next three holes: an integer variable hole (line 16), a char array variable hole (line 17), and an integer literal hole (line 18) in Figure 2a, are converted from `size` (line 10), `buffer` (line 11), and `0` (line 12) in Figure 1, respectively. The last two holes are converted from the `while` condition `pos < len && buf[pos] <= ' '` and `pos < len && buf[len - 1] <= ' '`, respectively. The hole `7` (line 26) in Figure 2a represents a logical relational expression that connects two relational expressions using a logical operator (`&&`, `||`). The first relational expression connects two integer variables using one of the relational operators. The second relational expression connects a char array access expression and an integer variable. The char array access expression selects an available variable of type `char[]` as the array and utilizes an integer variable as the index value to retrieve the corresponding element from the array. The last hole (line 36) in Figure 2a represents a similar expression but it uses an arithmetic expression as the index of the char array access expression. The arithmetic expression applies one of the arithmetic operators (`+`, `-`, `*`, `/`, `%`) on an integer variable and an integer literal (i.e., constant).

A template must have an entry method that is the start of the execution, annotated with `@Entry` as shown in the template (method `trim`). One of the key challenges is to obtain an argument for the method `trim`, i.e., an instance on which the method is to be invoked, so that the template can be executed. In our example, since the entry method `trim` is an instance method, the only required input is an instance of the template class `StrBuilder` that declares the method. To provide inputs to the entry method, LeJit inserts a public static *argument method*, annotated with `@Argument` (method `_arg0`). Thus, the argument method `_arg0` instantiates a `StrBuilder` and returns the instance after invoking a sequence of methods (line 42–46) in Figure 2a. Our key insight in this direction is to capture the sequence of methods during testing of the entry method `trim`; tests used can be either existing manually-written tests or automatically-generated tests. The sequence of methods to return an instance of class `StrBuilder` (line 42–46) in Figure 2a is obtained from a generated test.

Following JAttack, LeJit generates programs by executing the template from the entry method defined in the template. When LeJit reaches an unfilled hole the first time, it randomly picks a valid expression within the bounded search space defined by the hole. Once LeJit has filled all reachable holes, it outputs a generated program. Figure 2b shows an example generated program from the template in Figure 2a. In the figure, the hole API and the concrete expression generated to fill it share the same circled number, indicating a match between them. The generated program can be executed directly, as LeJit also generates a `main` method (line 36) in Figure 2b, that invokes the entry method using an instantiation of the template class returned from `_arg0` (line 38) in Figure 2b. The `main` method repeatedly invokes the entry method in a `for` loop (line 40–46) in Figure 2b. The large number of iterations is necessary to trigger JIT optimizations in Java since the JIT compiler triggers and starts to optimize code only when a method becomes "hot", i.e., frequently executed. To encode the program behavior during execution, the `main` method hashes and saves the argument values, return values, or any thrown exceptions from each iteration, and the final class state (i.e., static fields) of the template. These hashes are used to generate a checksum, which is the final output of the execution.

To perform differential testing [26], LeJit executes every generated program using JIT compilers in various JVM implementations and compare their outputs. The program in Figure 2b gave the same output using HotSpot and GraalVM, but it crashed the OpenJ9 JIT compiler. The IBM developers

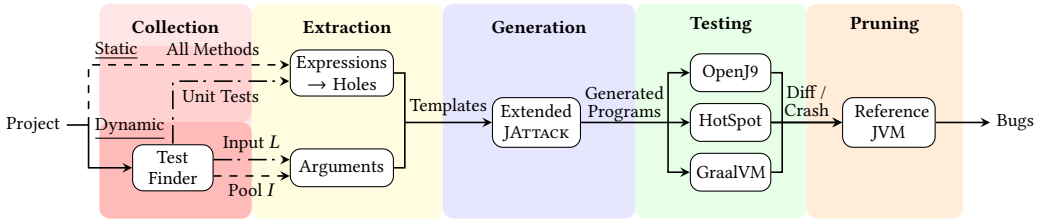

Fig. 3. The overview of LeJit. Dotted-dashed lines: test-based approach; dashed lines: pool-based approach.

confirmed that the crash is due to a bug in the OpenJ9 JIT compiler on handling array index out-of-bounds.

3 LEJIT FRAMEWORK

The LeJit framework has five key phases: (a) *collection*, (b) *extraction*, (c) *generation*, (d) *testing*, and (e) *pruning*, as illustrated in Figure 3. First (Section 3.1), LeJit collects a list of methods from the given code and obtains tests that can be used to create meaningful inputs to these methods. Then (Section 3.2), treating each method in the list as an entry method, LeJit selects the input that can be used to invoke the method, and extracts a template from the Java class that defines the method. Next (Section 3.3), LeJit executes each template with the selected inputs to the entry method to generate concrete Java programs. After that (Section 3.4), LeJit executes the generated programs through the entry method with the same selected inputs, using different Java JIT compilers for differential testing [26]. Finally (Section 3.5), LeJit prunes the detected crash and/or consistency as to minimize false positives, and then reports detected bugs.

3.1 Collection

In the collection phase, LeJit collects a list of methods from the given code to be used as template entry methods. LeJit then obtains tests for each of the methods, which will be used to obtain objects that can be used as arguments to the entry method.

We developed two approaches to collect a list of entry methods and to obtain code sequences that create arguments for entry methods.

Test-based. We use automated test generation to generate a large number of unit tests for all the classes in the given code. We utilize *the last method call* in the unit test as the entry method. As such, we can save the code sequence leading up to the method call as a way to construct arguments for that method. This approach ideally results in the same number of entry methods as the number of unit tests generated, and each entry method is associated with the saved code sequence as the input to the method.

Pool-based. Instead of using generated unit tests as in the previous approach, we save all prefixes of generated tests in the pool-based approach; each prefix creates an object that we add to an object pool. This object pool stores all code sequences produced during a test generation run, where each code sequence ultimately returns an instance of a class defined within the project (the object returned by the final method call in the sequence). We organize the pool using a mapping that associates each class with all the code sequences that can instantiate that class. This approach parses all the Java classes in the given project and obtains all the methods from these classes, and then uses all of the methods as entry methods.

Note that the pool-based approach creates a superset of objects created by the test-based approach, but the entry methods are different (as described above). We compare these two approaches in

```
1:  Input: M: the entry method
2:  Input: L: the collected input to the entry method (test-based only)
3:  Input: I: the pool of inputs to all methods (pool-based only)
4:  Output: the extracted template
5:  function Extract(M, L, I)
6:    C ← GetClassDeclaring(M)
7:    t̄ ← Clone(C)
8:    for all e in GetAllExprs(t̄) do
9:      e′ ← Convert(e, 0)
10:     Replace(e, e′)
11:   for all l in GetAllLoops(t̄) do
12:     InsertLoopLimiter(l)
13:   if L then
14:     m ← CreateArgumentsMethod(L)
15:     InsertMethod(m, t̄)
16:   if I then
17:     for all p in GetAllParams(M) do    ▷ including the receiver
18:       τ ← ResolveType(p)
19:       if IsPrimitive(τ) then
20:         i ← "<τ>Val()"
21:       else if I.contains(τ) then
22:         i ← RandomInputOfTypeFromPool(τ, I)
23:       else
24:         i ← "null"
25:       m ← CreateArgumentMethod(p, i)
26:       InsertMethod(m, t̄)
27:   return t̄

28: Input: ē: the original expression
29: Input: d̄: the depth of ē
30: Output: the hole API
31: function Convert(ē, d̄)
32:   τ ← ResolveType(ē)
33:   switch GetCategoryOfExpr(ē) do
34:     case Identifier:
35:       h ← "<τ>Id()"
36:     case Literal:
37:       h ← "<τ>Val()"
38:     case Relation:
39:       l̄ ← Convert(ē.left, d + 1)
40:       r̄ ← Convert(ē.right, d + 1)
41:       h ← "relation(<l̄>, <r̄>)"
42:     ...
43:   if d̄ = 0 then
44:     return "<h>.eval()"
45:   else
46:     return "<h>"
```

Fig. 4. Template extraction algorithm.

our evaluation to discover if they are complementary and if each leads to valuable inputs during compiler testing.

3.2 Extraction

For every entry method in the list provided by the collection phase, LeJit creates a template from the class that declares the method. Figure 4 shows the overall algorithm for LeJit to extract a template from a given entry method. The input to the function `Extract` is the entry method M, and either the collected inputs to M, if using the test-based approach, or the pool of inputs for all methods, if using the pool-based approach, is represented by L or I, respectively. The output is the extracted template $\bar{t}$.

The function `Extract` starts by finding the original class C that declares the entry method in the given Java code (line 6) and initializes template $\bar{t}$ as a clone of C (line 7). Next, for the class, LeJit recursively converts every expression in every method (obtained from `GetAllExprs(`$\bar{t}$`)`) into a hole. Next, `Extract` replaces each expression $\bar{e}$ in $\bar{t}$ with a hole API call (i.e., Java method that represents a hole) by calling the function `Convert` (line 9) and then replacing the expression into the hole in place (line 10). Rather than convert each expression into a hole, LeJit can also selectively create holes for some types of holes. Although we empirically evaluate impact of various types of holes, we assume in this algorithm that we convert each expression into a hole w.l.o.g.

The function `Convert` takes an expression $\bar{e}$ and its depth $\bar{d}$ as input and returns a hole API. It resolves the type of $\bar{e}$ as τ, then converts $\bar{e}$ into a hole API by recursively replacing each sub-expression of $\bar{e}$ with the proper hole API call. If $\bar{e}$ is an identifier, it is converted into a `<`τ`>Id` hole API call.

Example. The variable `str` of `String` type in the class from Figure 1 (line 7) is converted into `refId(String.class)` in the template from Figure 2a (line 9). Another `int` variable `size` (line 10) is converted into `intId()` in the same template (line 16).

If $\overline{e}$ is a literal, it is converted into a `<`τ`>Val` hole API call.

Example. The integer number `0` in the class from Figure 1 (line 12) is converted into `intVal()` in the template from Figure 2a (line 18).

If $\overline{e}$ is not a terminal, its sub-expressions are recursively converted into hole API calls. For a relational expression $\overline{e}$, the left and right sub-expressions are converted into the hole API call $\overline{l}$ (line 39 in Figure 4) and $\overline{r}$ (line 40), respectively, before $\overline{l}$ and $\overline{r}$ are combined using the `relation` hole API (line 41). The operator of the relational expression is ignored because a hole API uses all available operators by default if no operator argument is provided.

Example. Consider the relational expression `size == 0` in the class from Figure 1 (line 9). The left sub-expression `size` is converted into `intId()`, and the right sub-expression is converted into `intVal()`. Then the two results are combined as `relation(intId(), intVal())` in the template from Figure 2a (line 14).

Other expressions that are non-terminals, e.g., arithmetic, logical, array access, etc., are converted in a similar way as a relational expression; we do not list all of them in the algorithm. Once the given expression $\overline{e}$ is converted into a hole API call, `Convert` checks if the current depth is 0. If so, it appends the `eval()` call to the hole API call and returns the resulting call as the output of the function (line 43–46). Note that `eval()` first triggers the hole API call, which returns an expression that fills the hole. Then `eval()` is called on the returned expression that evaluates to the type that the hole represents (e.g., `int`). Thus, there is only one `eval()` for the outermost hole API call.

A hole as a loop condition might introduce an infinite loop in the template class $\overline{t}$ if the hole is filled in with some expression that is always evaluated to `true` at the generation phase. Therefore, `Extract` inserts a loop limiter to restrict the maximum iterations that one loop can be executed (line 11–12 in Figure 4).

Example. Consider hole 7 in the template from Figure 2a (line 26), which is a loop condition. To prevent the infinite loop that may occur due to filling in random values, a loop limiter `_lim1++ < 1000` is appended to the `logic` hole to restrict the maximum iterations to one thousand times.

Once the holes are created, `Extract` then creates and inserts argument methods into $\overline{t}$, according to the selected approach in the collection phase.

Test-based. A public static `@Arguments` method is added to the template class $\overline{t}$ (line 14–15 in Figure 4). The method returns an array of all the inputs to the entry method in the order of method parameters. Consider the entry method with signature:

```
public Quaternion multiply(final double alpha)
```

in the class `org.apache.commons.math4.complex.Quaternion` from open-source project math [39]. The following `@Arguments` method is generated:

```
@Arguments
public static Object[] _args() throws Throwable {
  Quaternion quaternion = new Quaternion(
      35.0, (double) 0, 57.29577951308232, -1.0);
  return new Object[] { quaternion, (double) 17 };
}
```

where `quaternion` and `17` are the inputs collected in the collection phase, i.e., extracted from a generated unit test with the entry method `multiply` as the last method call.

Pool-based. A public static `@Argument` method is created to provide an input for each parameter (including the receiver) of the entry method according to the type τ of the parameter (line 17–26 in Figure 4). If a primitive input is required, then a `<`τ`>Val` hole API will be used (line 20); otherwise a reference input with the required type is randomly picked from the object pool provided from the collection phase (line 22). If the pool does not contain the type, `null` is used (line 24). Consider the entry method `trim` in the template from Figure 2a, since it is an instance method without any parameter, only a single `@Argument` method `_arg0` (line 42–46) is created. The method `_arg0` uses a randomly picked code sequence from the object pool and returns an instance of `StrBuilder` that can be used to invoke the instance entry method `trim`.

Finally, `Extract` returns the template $\bar{t}$ (line 27 in Figure 4). LeJit repeats the procedure to extract a template for every entry method provided in the list from the collection phase.

3.3 Generation

In the generation phase, LeJit obtains concrete programs from every template extracted from the previous phase. LeJit generates programs through an execution-based model. Given a template $\bar{t}$, the initial global state is captured first. Then, the entry method of the template is repeatedly executed, stopping when all holes are filled or the maximum iterations N has been reached. Next, the technique outputs a generated program by filling every hole with corresponding concrete code. This process repeats M times to generate M programs, with the template state reset after each program generation. LeJit builds on JAttack to support the generation phase.

Extending template support. LeJit enhances JAttack in several aspects. (1) JAttack allows only a static method as the entry method. On the other hand, LeJit introduces the receiver object for the entry method, which allows an instance method as the entry method via passing the receiver's value from `@Argument` or `@Arguments` methods. (2) JAttack does not support non-primitive static fields in templates, as it resets a static field by saving and recovering the value. To resolve this, LeJit resets states of template classes by re-invoking static initializers (`clinit`) [2], thus allowing non-primitive static fields to be reset in templates. (3) JAttack crashes due to `UnfilledHoleException` immediately when encountering holes in static initializers, while LeJit adds extra logic to handle those exceptions when loading (including re-initializing) template classes. (4) Certain holes in constructors are not supported well by JAttack. For a `<`τ`>Id` hole inside `super()` or `this()` calls from a constructor, JAttack can fill the hole with a field accessed from `uninitializedThis`, which fails bytecode verification. LeJit overcomes the limitation by tracking at which execution point in a constructor (when all `INVOKESPECIAL` and `NEW` bytecode instructions are paired up) `uninitializedThis` gets initialized and can be used. (5) LeJit introduces a number of new hole APIs for type casting and improves the checksum utility of JAttack to avoid hash collisions when hashing an object graph.

Improving generation procedure. LeJit makes two changes to the original generation procedure of JAttack. (1) One of the advantages of JAttack's execution-based generation over static generation is that it knows exactly what gets executed in a generated program and such information can help generate better programs. However, JAttack does not leverage the information in its implementation. It simply outputs any generated program as long as the program compiles. Instead, LeJit skips certain generated programs that are less likely to trigger JIT optimizations. For instance, LeJit will skip a generated program if the execution stops even before entering the entry method due to an exception thrown from argument methods. (2) JAttack renames the class with a unique suffix in every generated program, e.g., Gen1, Gen2, etc. However, such renaming breaks circular dependencies between the template class and other classes in the same project, which makes many generated programs not compilable. LeJit disables renaming and keeps the original class name of the template for all generated programs. When executing a generated program in the testing phase,

LeJit ensures that the compiled classfile of the generated program appears in the classpath before all the other classes of the original Java source, such that the generated program, rather than the original class with the same name in the project, will be used.

3.4 Testing

For differential testing, LeJit executes each generated program with various implementations and levels of JIT, i.e., different *JIT configurations*. We define a JIT configuration as a tuple (vendor name, compiler name, version number, JVM options), for example: (Oracle, HotSpot, 20, `-XX:TieredStopAtLevel=1`) and (IBM, OpenJ9, 17.0.6, `-Xjit:optlevel=hot`). Each generated program is executed repeatedly with a large number of iterations (to trigger JIT compilation) and outputs a checksum value in the end. This checksum value is calculated by hashing the arguments provided to the entry method, the output of the return value from the entry method in each iteration, and the final state (i.e., static fields) of the entire class [51]. Then, LeJit compares the checksum values from different JIT configurations and reports a failure if it observes any difference. Additionally, LeJit reports a failure if the program crashes on some JIT configurations.

3.5 Pruning

Not every failure indicates a real issue with JIT. During our experiments, we find that most of the failures reported due to observed inconsistent checksum values across JIT configurations were caused by either (1) non-deterministic features of the generated program itself, e.g., random numbers, current timestamps, hashcode, etc., (2) the inconsistency between JIT configurations themselves, e.g., system property `java.vm.name` and `java.vm.info`, which contain the JVM's version information and Java options used, or (3) discrepancies between JVM implementations from different vendors, such as HotSpot and OpenJ9, which disagree on the maximum array size. To alleviate this problem, JAttack reruns the failing program twice using the interpreter mode (`-Xint`) of a single JIT configuration, while keeping the rest of the JIT configuration intact, and reports the failure only when the two reruns using interpreter mode give the exact same outputs. However, this solution can only filter out false positive JIT bugs caused by (1) but not (2) or (3). LeJit improves the filtering by a) using various JVMs (e.g., HotSpot and OpenJ9) and b) increasing the number of reruns of the failing program from one to three times. If any of the reruns using interpreter mode still shows inconsistent checksum values across various JVMs, LeJit considers the failure to be not related to JIT and thus ignore it as a false positive.

When a generated program crashes while being executed using a particular JIT configuration, JAttack always labels it as a bug. However, not all crashes are caused by issues with the JVM and therefore not all are worth reporting to developers. Some crashes are `UnfilledHoleException`, which occur due to unfilled holes in the program, which are left as API method calls during the generation phase but are reached during execution in the testing phase (due to non-determinism). Such cases may be caused by incorrect JIT compilation that leads to a mismatch in program behavior between the generation and testing phases, which we want to report as a bug. However, many of these cases result from the aforementioned three reasons that cause inconsistent checksum values between JIT configurations. For example, non-deterministic features such as current timestamps may have inconsistent values between the generation phase and testing phase. This inconsistency can cause disagreement in code paths taken between the generation phase and the testing phase, e.g., when evaluating `if` conditions on timestamps, which can result in unfilled holes that were not reached during generation but were reached during testing. To address this issue, LeJit reruns the generated program with various JVMs using interpreter mode if the program reports a crash due to `UnfilledHoleException`. If the program does not crash during the rerun, then LeJit reports

a bug. However, if the program still crashes during the rerun, then LeJit considers the crash as a false positive and skips reporting the failure.

While the pruning approach is simple, with manual inspection on a number of cases, we found it sufficiently useful. We also compared our pruning with original JAttack's pruning. Our pruning filtered around 96%, while JAttack filters out around 60%, out of total failures.

3.6 Implementation

We implement collection of entry methods, extraction, and pruning as standalone tools. We extend Randoop [33] to obtain objects used as arguments for non-primitive types. Finally, we extend JAttack [51] to support generation and testing phases.

4 EVALUATION

We assess the value of LeJit by answering the following research questions:

RQ1: What are the contributions of the major components of LeJit?
RQ2: How effective is LeJit compared with the state-of-the-art techniques?
RQ3: What is the impact of holes in various Java language features on LeJit's bug detection?
RQ4: What is the impact of different types of templates on LeJit's bug detection?
RQ5: What critical bugs does LeJit detect in Java JIT compilers?

We first describe the experiment setup (Section 4.1) and then answer each of the research questions (sections 4.2-4.6).

4.1 Setup

Collection. We use open-source projects as the main input to LeJit for extracting templates. An alternative was to generate Java programs using one of the techniques for testing traditional Java compilers [11, 17], but open-source projects cover a much broader range of Java features. We search GitHub [16] for 1,000 Java open-source projects with the most stars, and we also include projects with at least 20 stars that belong to several popular organizations, e.g., Apache, Google, etc. In total, we collected 1,793 projects. We further filter by keeping the projects that (1) use the Maven [41] build system; (2) have a license that permits our use; and (3) have tests. After this step, there were 161 projects. Then, we attempt to build each project from its source and create a fat jar [34] for each project. We filter out any projects that cannot be packaged this way. Lastly, we exclude some projects that are not compatible with LeJit's toolchain, e.g., ASM [32], JavaParser [19], and Randoop [33]. Eventually, there are 62 projects for use.

We run LeJit once using the pool-based approach with all the 62 projects but stop LeJit early, before the generation phase, in order to extract templates and collect holes. We next select the top ten projects with the most holes and loop limiters (used to avoid introducing infinite loops; see Section 3.2) in the extracted templates. Table 1 shows the ten open-source Java projects and associated numbers of holes and loop limiters; we show the number of holes for each *hole type*.

In the test-based approach, we configure the test generation to obtain 5,000 unit tests for each project or for 30 minutes, whichever comes earlier. In the pool-based approach, we always run test generation for 30 minutes. Lastly, we use Eclipse Temurin 11.0.18 (Adoptium OpenJDK build) in the collection phase, including building open-source projects, test generation, and running LeJit itself. We select this lower version of Java in order to maximize compatibility with open-source projects and LeJit's toolchain such as Randoop, i.e., being able to compile most projects into fat jars and to run Randoop with the projects, with a Java version.

Generation. We generate ten programs from each template, with a three-minute timeout. We also set a one-minute timeout in the testing phase for executing each generated program. (We

Table 1. Project information and number of holes per hole type. PrimitiveId contains all the <τ>Id holes, and PrimitiveVal contains all the <τ>Val holes, where τ is one of the primitive types in Java. # Loops is the number of loop limiters.

Project	# Holes							# Loops	$\sum$
	PrimitiveId	PrimitiveVal	Array	Arithmetic	Shift	Relation	Logic		
vectorz	1,585,341	498,470	187,333	468,965	3,115	207,971	7,960	112,319	2,959,155
math	969,150	391,390	83,093	275,034	6,392	114,672	10,547	67,306	1,850,278
lang	984,373	464,883	73,260	93,359	1,318	190,413	15,680	77,790	1,823,286
text	246,909	86,419	8,648	34,378	0	48,976	3,462	11,943	428,792
compress	178,754	89,706	9,296	17,012	2,668	25,710	3,325	10,163	326,471
zxing	93,637	74,610	10,368	23,204	1,282	17,740	3,364	6,937	224,205
codec	35,414	36,528	5,420	8,103	1,597	3,831	373	1,335	91,266
statistics	31,753	7,271	141	7,825	0	5,110	507	235	52,607
jfreechart	6,253	2,920	287	1,190	12	539	78	182	11,279
numbers	6,441	2,082	22	891	140	1,065	154	123	10,795
$\sum$	4,138,025	1,654,279	377,868	929,961	16,524	616,027	45,450	288,333	7,778,134

change the value to 50 seconds when later comparing against JITfuzz for fair comparison.) We use Oracle JDK 17.0.6 to execute templates in the generation phase. We select a different JDK version for additional differential testing between the generation and testing phases.

Testing. We test a wide range of JDKs with different vendors and versions during our experiments. When evaluating LeJit alone including its variants, we use Oracle JDK 20 (HotSpot default and level 1), IBM Semeru 17.0.6.0 (OpenJ9 default and hot level), and GraalVM Enterprise Edition 22.3.1 (GraalVM default) for differential testing. When comparing LeJit with JITfuzz and JavaTailor, we also include a custom build of OpenJDK jdk-17.0.6+10 (HotSpot default and level 1) (see Section 4.3). We collect code coverage over the JVM code using the custom build of OpenJDK. We separately re-run generated programs to collect code coverage when evaluating LeJit alone including its variants. We collect code coverage on the fly when comparing LeJit with JITfuzz and JavaTailor. JITfuzz uses coverage, so we use the same setup for all the tools.

Pruning. We use reference JIT configurations to rerun three times every failing program, i.e., a generated program that either has different outputs or has crashed JVM in the testing phase. We report such a failing program as a bug if the rerun using reference JIT configurations does not show any difference or crash (see Section 3.5). We use HotSpot with `-XX:TieredStopAtLevel=0` and OpenJ9 with `-Xnojit` as reference JIT configurations when pruning failures.

Machine. We run all experiments on a 64-bit Ubuntu 18.04.1 desktop with an Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 64GB RAM.

4.2 Contribution of Major Components

We evaluate LeJit using both test-based and pool-based approaches (Section 3), named LeJit$_t$ and LeJit$_p$, respectively. Additionally, we define two baselines (LeJit$_{\text{NoTmpl}}$ and LeJit$_{\text{NoPool}}$) to help us understand the benefit of using templates and creating instances for entry methods.

The variant LeJit$_{\text{NoTmpl}}$ follows the same collection phase as LeJit$_t$ that collects the last method call as the entry method from every unit test generated. However, LeJit$_{\text{NoTmpl}}$ does not extract any templates, and thus not generate any programs from templates. Instead, it uses existing code and directly goes to the testing phase and executes the entry method a large number of times, using just the arguments in the test. This baseline (indirectly) shows the power of automatically generated tests, obtained on a randomly selected set of projects, for discovering Java JIT compiler bugs.

Table 2. Comparison of LeJit variants. # Tmpl. is the number of templates, # Gen. is the number of generated programs, # Fail. is the number of failures. All the numbers are averages.

	# Tmpl.	# Gen.	# Fail.	# Avg. Bugs	Coverage (%)		
					C1	C2	HotSpot
LeJit$_t$	10,714	90,916	66	3.3	83.8	79.0	49.5
LeJit$_{\text{NoTmpl}}$	14,854	14,854	24	0.7	83.6	78.4	47.9
LeJit$_p$	11,921	99,644	129	5.3	84.4	79.6	50.0
LeJit$_{\text{NoPool}}$	10,185	89,977	56	4.0	84.0	78.7	49.2

We design the variant LeJit$_{\text{NoPool}}$, which extracts a template for every method in a given project. The only difference (compared to LeJit$_p$) is that LeJit$_{\text{NoPool}}$ does not collect the object pool, when extracting templates, but it rather searches for public constructors, or static methods without parameters or with only primitive parameters, or `null` to construct reference arguments. LeJit$_{\text{NoPool}}$ is a superset of the original template extraction technique presented in JAttack [51]; LeJit$_{\text{NoPool}}$ supports more types of holes and more entry methods than JAttack. LeJit$_{\text{NoPool}}$ shares the same generation and testing phases with LeJit$_p$.

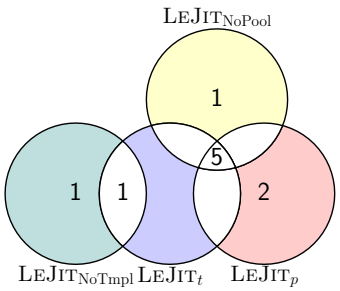

Fig. 5. The overlap of bugs detected by LeJit variants. LeJit$_{\text{NoTmpl}}$: no templates/generated programs; LeJit$_t$: LeJit with Test-based approach; LeJit$_p$: LeJit with Pool-based approach; LeJit$_{\text{NoPool}}$: enhanced JAttack.

Table 2 compares the numbers of generated programs, failures, and unique bugs reported. Note that all the numbers in the table represent averages from three runs. The various LeJit variants exhibit differences in their running times. Specifically, the slowest variant (LeJit$_p$) required around six days for a single run, on average. The first two rows compare LeJit$_t$ and LeJit$_{\text{NoTmpl}}$. LeJit$_t$ executes much more programs than LeJit$_{\text{NoTmpl}}$, since LeJit$_{\text{NoTmpl}}$ does not extract templates to generate programs. LeJit$_t$ also slightly increases code coverage in C1 and C2 (separate optimizing compilers within HotSpot), as well as in the entire HotSpot. However, it was interesting to observe that LeJit$_{\text{NoTmpl}}$ can even find an average of 0.7 bugs per run. As seen from the second two rows, LeJit$_p$ and LeJit$_{\text{NoPool}}$ execute a comparable number of programs and both find a few bugs. LeJit$_p$ achieves both higher coverage and higher number of bugs on average.

Figure 5 shows all the bugs we found from the variants and the overlap of different variants. We do not include two bugs found in our preliminary experiments and three bugs found during experimenting with various template types (Section 4.5). We can see that both automated generation of instances and template extraction contribute to detecting the bugs. LeJit$_t$ and LeJit$_p$ together miss two bugs that LeJit$_{\text{NoTmpl}}$ and LeJit$_{\text{NoPool}}$ find. Interestingly, LeJit$_{\text{NoTmpl}}$ without any template or holes finds two bugs, which shows the effectiveness of traditional automated test generation even for domain that is not originally targeted.

4.3 Comparison with State-of-the-art

We compare LeJit with JITfuzz [44] (version `3dc8f91`), a state-of-the-art technique for automated Java JIT testing. Additionally, we compare LeJit with JavaTailor [53] (version `bf9421f`), a history-driven test program synthesis for testing JVMs. Although JavaTailor does not target JIT per se, it is worth learning about the relation and potential overlap between LeJit and JavaTailor.

Table 3. Comparison of JITfuzz and LeJit. *The bug was already found before using JITfuzz so we did not report it again.

	# Programs	# Fail.	# Bugs	Coverage (%)					
				C1		C2		HotSpot	
				Func.	Line	Func.	Line	Func.	Line
JITfuzz	45,352	2,115	*1	70.8	69.7	67.2	64.3	36.6	44.6
LeJit$_t$	96,626	97	0	78.9	77.7	74.6	72.5	39.4	48.0

JITfuzz vs. LeJit. To compare against JITfuzz, for a given project, we need to provide JITfuzz an *initial class* as the seed, as well as a test class as a starting point to execute the mutated programs from those seeds. Following the same methodology as in the previous work [44], we first identify ten classes in the given project with the highest cyclomatic complexity, and we pick the initial class randomly from these ten classes. Since the original work did not mention how the test class should be selected, we randomly pick a test class that imports and instantiates the initial class. To ensure a fair comparison, we run JITfuzz with the same ten projects (Section 4.1). During our preliminary experiment, we found that JITfuzz does not support tests using JUnit 5 [42], so we manually migrated the picked test classes to JUnit 4 [22] in five projects (other projects already used JUnit 4).

We use LeJit$_t$, i.e., with the test-based approach, in order to better control end-to-end running time by specifying the number of generated tests. (The pool-based approach uses all the available methods in the projects, which makes it hard to estimate the time needed.) We run both tools for the same length of time, around six days, which is longer than used in the JITfuzz evaluation [44] and in recommended practice [23], while also matching the end-to-end running time of LeJit$_t$. We use the same timeout, 50 seconds, which is the default setting of JITfuzz, for executing each single generated program. JITfuzz requires custom debug builds of OpenJDK with AFL++ toolchain to work, because it needs to collect runtime coverage of JIT source code [44]. Thus, we build OpenJDK jdk-17.0.6+10 from source [31] with `-enable-debug` and `-enable-native-coverage` and use the debug build as JIT under test. Note that LeJit works on both debug and release builds. We use a debug build for fair comparison (we already ran LeJit on multiple released binaries in Section 4.2). Also, JITfuzz does not use differential testing but detects only crashes, so we do not use OpenJ9 and GraalVM for LeJit for a fair comparison; instead we use only default level and level 1 of HotSpot from the custom debug build of OpenJDK for differential testing required by LeJit. We collect code coverage of C1 (`src/hotspot/share/c1/*`), C2 (`src/hotspot/share/opto/*`), and the entire HotSpot (`src/hotspot/*`).

Table 3 compares the results from both tools. Note that all the numbers in the table represent averages from three runs. JITfuzz reports 2,115 failures out of 45,352 programs that have been generated and executed. On the other hand, LeJit executes 96,626 programs and reports 97 failures. We then analyze and inspect the failures from both tools. Both tools do not detect new bugs. All the 2,115 failures reported by JITfuzz are assertion failures (which are checked on debug builds only). We group the assertion failures by stack traces and error lines in source code within HotSpot, and there are only two unique failures. Both are duplicates of an existing bug JDK-8280126 [29] on optimizing irreducible loops. We do not find any bug from LeJit's failures. We believe the reason for this finding is that we collect code coverage on the fly for the debug build, which impacts the way JIT optimizes generated programs. LeJit detects a number of HotSpot bugs in other experiments we perform using non-debug builds (Section 4.2). LeJit increases line coverage of C1 by 8.0%, C2 by 8.2%, and HotSpot by 3.4% compared to JITfuzz.

Table 4. Number of cases of each group reported from JavaTailor.

Group	NoRep.	NonDet.	DiffText	VerifyError	DiffException	NoException	Misc.
Number	5	18	39	2	24	7	7

JavaTailor vs. LeJit. JavaTailor [53] performs history-driven test program synthesis to test JVM implementations. More precisely, JavaTailor uses previously reported bugs as seeds to synthesize diverse test programs by combining ingredients from historical bug-revealing programs. JavaTailor was shown efficient for testing JVM implementations and here we explore if it can also discover JIT-related bugs.

We ran JavaTailor three times in the default configuration until completion (~8h each run). We inspected the three runs in detail and concluded that findings are similar across runs, thus no further runs were warranted. We used two versions of Java, as JavaTailor also performs differential testing: IBM Semeru 17.0.6.0 (OpenJ9 default level) and a custom build of OpenJDK jdk-17.0.6+10 (HotSpot default level) like in the previous section. We pick these two versions because HotSpot and OpenJ9 were used by JavaTailor's authors in their evaluation, and we use the custom build of OpenJDK because we need to collect code coverage of JIT compilers and compare with LeJit.

As a result of each run, JavaTailor outputs a diff log. We could not find any existing scripts for processing the diff logs, so we wrote our own to help us classify failures and perform inspection. Additionally, we wrote scripts to help us try to reproduce each of the reported failures.

JavaTailor reported 102 differences in the diff log (and each diff corresponds to one class file that is executed with two JVMs and produces different results). We semi-automatically classified the reported cases into 7 groups. Table 4 shows number of cases of each group. NoRep. includes cases that show no differences when we tried to reproduce the difference. NonDet. includes cases that non-deterministically pass or fail (e.g., due to elapsed time being in the output) and are not revealing any bug. DiffText includes cases that are only reported with different text across JVMs, but the reported issue is actually the same. VerifyError includes cases when bytecode verification failed in both JVMs, but the messages were different. DiffException includes cases when exceptions are printed in a different order across JVMs. NoException includes cases when only one of the JVMs throws an exception, but our further inspection showed that these cases were caused by flags that have different default values across JVMs. Misc. includes single instance failures that do not fit into any other group we defined; we found one bug in this group, but the same bug was previously reported [13].

Regarding code coverage, LeJit increases code coverage of C1 by 3.3%, C2 by 4.0%, and the entire HotSpot by 0.4%, compared to JavaTailor.

In conclusion, JavaTailor can discover JVM bugs, but none were related to JIT. We also found that the default reporting has many false positives. We find LeJit and JavaTailor complementary, and each could potentially benefit from the other; we leave the combination of the two for future work.

4.4 Impact of Holes in Various Language Features on LeJit's Bug Detection

In order to understand how holes in different language features contribute to bug detection of LeJit, we perform in-depth analysis on the features within extracted templates and generated programs.

We analyze three language constructs, i.e., arrays, conditional statements, and loops, and one other language feature, i.e., reference arguments. If a filled hole is inside a language construct (e.g., a hole is inside a loop), then we say the generated program that contains the filled hole *has* the language construct, and we also say the associated template from which the generated program is generated *has* the language construct. Similarly, we also measure how many templates and

Table 5. Number of templates and programs with different language features. Cond. is Conditional Statements. R.A. is Reference Arguments.

Project	# Template					# Generated Programs				
	Total	Arrays	Cond.	Loops	R.A.	Total	Arrays	Cond.	Loops	R.A.
codec	11,098	2,476	2,719	4,102	10,414	43,147	19,760	23,244	23,474	39,529
compress	8,269	1,036	4,133	1,818	7,794	62,431	7,585	35,774	15,713	58,578
jfreechart	11,041	215	2,073	554	10,656	61,883	1,894	13,294	4,117	59,587
lang	18,602	3,491	9,156	6,050	16,136	117,061	28,995	78,002	53,117	100,349
math	20,692	3,964	10,116	5,311	18,097	153,912	33,044	84,054	43,810	139,857
numbers	18,021	826	6,473	2,449	1,971	98,218	8,260	63,496	15,599	9,322
statistics	10,010	11	4,396	159	9,188	44,181	37	34,634	810	38,622
text	11,532	2,423	5,105	3,506	10,752	66,395	18,975	46,965	31,068	61,473
vectorz	23,005	6,031	11,277	6,904	21,718	175,814	49,594	94,726	58,759	165,803
zxing	10,925	1,660	3,767	2,375	10,145	63,136	13,440	30,993	19,417	57,703
$\sum$	143,195	22,133	59,215	33,228	116,871	886,178	181,584	505,182	265,884	730,823

Table 6. Impact of Java language features on bugs. Cond. is Conditional Statements. R.A. is Reference Arguments. *Bugs may repeat across projects, and we show the number of unique bugs across all projects.

Project	# Failures due to Bugs					# Bugs (Unique)				
	Total	Arrays	Cond.	Loops	R.A.	Total	Arrays	Cond.	Loops	R.A.
codec	5	1	3	1	5	3	1	2	1	3
compress	6	1	5	1	6	2	1	1	1	2
jfreechart	0	0	0	0	0	0	0	0	0	0
lang	24	6	18	14	23	2	1	2	2	2
math	21	11	19	10	13	6	4	6	4	4
numbers	0	0	0	0	0	0	0	0	0	0
statistics	1	0	0	0	0	1	0	0	0	0
text	2	1	2	2	2	2	1	2	2	2
vectorz	107	42	95	91	93	5	3	5	5	5
zxing	8	8	2	8	0	1	1	1	1	0
$\sum$	174	70	144	127	142	10*	7*	8*	8*	9*

generated programs use an entry method that needs an argument of non-primitive (reference) type, which means the arguments are obtained by generated tests. We say such templates and generated programs have reference arguments. Table 5 shows the numbers of templates and generated programs that use the four language features. We can see that a substantial number of templates and programs need non-primitive arguments.

Similarly, we say a bug *has* a language feature if any generated program that exposes the bug (i.e., failure due to bug) has the language feature. Note that we do not claim that the presence of a feature implies that the bug is related to the feature or the feature is the root cause of the bug. Table 6 shows the numbers of failures due to bugs and unique bugs that use various language features. In conclusion, LeJit well explores the four language features and holes in each of these features contribute to the unique bugs discovered.

4.5 Impact of Template Types on LeJit's Bug Detection

To explore how different types of templates affect LeJit's bug detection, we extract different sets of templates from the same ten projects (Section 4.1). We modified the template extraction algorithm (Figure 4) so that each extracted set of templates contains a single set of specific types of holes out of (1) <τ>`Id`, (2) <τ>`Val`, (3) `arithmetic()` and `shift()`, (4) <τ>`relation` and <τ>`logic`. We also extract

Table 7. Results when using a specific set of types of holes.

	<τ>Id()	<τ>Val()	arithmetic() & shift()	relation() & logic()	All
#Templates	13,090	12,139	13,606	13,682	12,061
#Programs	105,759	87,230	59,917	64,906	102,670
#Failures	106	29	39	70	131
#Bugs	3	1	3	4	4

Table 8. Detected bugs in HotSpot, OpenJ9 and GraalVM using LeJit; 11 bugs were previously unknown.

JVM	Bug ID	Type	JDK Versions	Status	CVE	Duplicates
GraalVM	GR-45498	Diff	17, 20	Fixed	-	-
HotSpot	JDK-8301663	Diff	18, 19, 19.0.2	Fixed	-	JDK-8288064
	JDK-8303946	Diff	8, 11, 17, 19, 20, 21	Confirmed	-	-
	JDK-8304336	Diff	17, 19, 20, 21	Fixed	CVE-2023-22044	-
	JDK-8305946	Crash	17, 19, 20, 21	Fixed	CVE-2023-22045	-
	JDK-8325216	Crash	17, 18, 19, 20, 21	Fixed	-	JDK-8319793
OpenJ9	17066	Crash	8, 11, 17, 18	Fixed	-	-
	17129	Diff	8, 11, 17, 18	Fixed	-	-
	17139	Diff	8, 11, 17, 18	Fixed	-	-
	17171	Crash	11, 17, 18	Fixed	-	-
	17212	Crash	8, 11, 17, 18	Fixed	-	15363
	17249	Diff	8, 11, 17, 18	Fixed	-	-
	17250	Diff	17, 18	Fixed	-	-
	18802	Crash	8, 11, 17, 21	Fixed	-	17045
	18803	Crash	11, 17, 21	Fixed	-	-

a set of templates with all types of holes, which is the default setting. Other than the extraction phase, we use the same methodology and configuration as described in Section 4.1 to run LeJit with the five sets of templates. Table 7 shows the numbers of templates, generated programs, reported failures and bugs from all five sets of templates. The set of templates with all types of holes reports the most number of bugs. Out of the other four sets of templates with only a single set of holes, the `relation()` and `logic()` holes reports the most number of bugs, but even the simplest set of holes, i.e., constant replacement (<τ>Val), plays an important role. Furthermore, we discovered three additional JIT bugs using these various sets of templates.

4.6 Detected Bugs

Table 8 lists the bugs that LeJit detected. So far, we have discovered and reported 15 bugs, 11 of which are previously unknown, including two CVEs. We show (in Figure 6) and describe four bugs that encompass a variety of JIT issues.

Arithmetic mis-compilation. A mis-compilation occurred when the OpenJ9 JIT performed a modular operation with parameter passing (Figure 6a). We discovered the bug using a template created from math [39]. The issue lies in an incorrect reuse of a register whose value changes after a floating-point remainder operation.

Incorrect elimination of range checks. From a template extracted from math [38], we discovered a HotSpot JIT mis-compilation bug where the range check for array accesses was incorrectly eliminated, which missed throwing exceptions and produced incorrect results (Figure 6b). Upon reporting the bug, Oracle developers promptly confirmed the issue. They classified the bug as a CVE and rolled out the fix in the next Critical Patch Update.

```java
public class C {
  double q0, q1, q2, q3;
  C(double a0, double a1, double a2, double a3) {
    q0 = a3; q1 = a1; q2 = 0; q3 = 0; }
  static double m(double d) {
    C c = new C(0, 1.0, 0, d % d);
    return c.q1; }
  public static void main(String[] args) {
    double sum = 0;
    for (int i = 0; i < 100_000; ++i) {
      // m(1.0) expected to be 1.0 returns 0.0
      sum += m(1.0); }
    // expected 100000.0
    System.out.println(sum); } }
```

(a) Arithmetic mis-compilation.

```java
public class C {
  static void m(int n) {
    int[] a = new int[n];
    for (int i = 0; i < 1; i++) {
      int x = a[i % -1]; } }
  public static void main(String[] args) {
    int count = 0;
    for (int i = 0; i < 1000; ++i) {
      try { m(0);
      } catch (ArrayIndexOutOfBoundsException e) {
        count += 1; } }
    System.out.println(count); } } // expect 1000
```

(b) Incorrect elimination of range checks.

```java
public class C {
  static int m(int len) {
    int[] arr = new int[8];
    for (int i = 10000000, j = 0;
         (boolean) (i >= 1) && j < 100; i--, j++) {
      // should not enter inner loop.
      for (int k = 0; len < arr.length; ++k) {
        int x = 1 / 0; }
    } return 0; }
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < 100_000; ++i) {
      try { m(13);
      } catch (ArithmeticException e) { sum += 1; }
    } System.out.println(sum); } } // expected 0
```

(c) Erroneous loop condition evaluation.

```java
static import java.nio.charset.StandardCharsets;
public class C {
  static int m(String s) {
    byte[] arr = s.getBytes(ISO_8859_1);
    return arr[2]; }
  public static void main(String[] args) {
    long sum = 0;
    for (int i = 0; i < 10_000_000; ++i) {
      sum += m("\u8020\000\000\020"); }
    System.out.println(sum); } } // expected 0
```

(d) Standard library mis-compilation.

Fig. 6. Examples of minimized programs of detected bugs.

Erroneous loop condition evaluation. Execution of OpenJ9 JIT-compiled code faced a situation where a loop condition was incorrectly evaluated as true, enabling the loop body to run (Figure 6c). However, the loop body should never execute, and this correct behavior was observed in non-JIT executions. LeJit flagged this issue as a bug using a template from codec [36]. IBM developers confirmed the bug within a day.

Standard library mis-compilation. An incorrect output occurred when using GraalVM JIT compilation with the String `getBytes` method (Figure 6d). The generated program by LeJit emerged from a template based off code from codec [37]. The developers confirmed the bug within one day.

Bugs detected with LeJit are presented in an easily digestible manner. Generated programs are easy to minimize and understand, because LeJit extracts templates from real-world Java programs and the minimum example programs we submitted are Java source code. Developers were able to quickly understand our reports and reproduce or further minimize source code as needed. Many reports were confirmed by the first 48 hours. In contrast, bytecode files generated by some tools require substantial effort to understand by compiler developers [30].

5 LIMITATIONS AND FUTURE WORK

When we compared LeJit with JITfuzz, we used only the test-based approach, and we were unable to successfully run another tool: JOpFuzzer [20]. Also, when we evaluated LeJit variants, we did not attempt to match end-to-end duration of $\text{LeJit}_{\text{NoTmpl}}$, or $\text{LeJit}_{\text{NoPool}}$, and end-to-end duration of

LeJit$_t$, or LeJit$_p$. The randomness could impact the experiment results, so we run each experiment three times, as we already described.

Our current implementation of the LeJit test-based approach collects arguments to templates by extending one tool for test generation. The same methodology can also work for manually-written tests or other tools [14], which we plan to explore in the future. Also, it will be easy to migrate LeJit to test other software systems that also take Java programs as input, such as refactoring tools.

LeJit has enhanced JAttack to support non-primitive static fields in templates by re-initializing the template class, but LeJit is limited on re-initializing other classes in dependencies. While JAttack's exception handling has been enhanced to handle exceptions thrown from class loading and initializing, LeJit does not handle errors directly thrown from JVM in the generation and testing phase, e.g., `StackOverflowError`, `OutOfMemoryError`, etc. These shortcomings sometimes left unfilled holes that were supposed to be filled, leading to false positives in the testing phase. We will further explore re-initializing all classes in dependencies and better handling of JVM errors.

Ethical considerations. To avoid "spamming" open-source community, we submit a bug report only when we can reproduce the bug on the latest release of the affected JDKs. We also tried our best to detect duplicates and minimize the programs that reproduce the bugs.

6 RELATED WORK

We describe closely related work on compiler testing [6] using grammar-based, mutation-based and template-based approaches, and test input generation in general.

Grammar-based. These tools use the grammar production rules of the language to generate test programs. Csmith [46], Orange [27] and YARPGen [25] are grammar-based tools for generating C/C++ test programs. Lava [35] uses grammar production rules to generate Java bytecode test programs for testing the JVM. Java* Fuzzer [1] is a grammar-based tool that generates random Java programs for testing JIT compilers. Yoshikawa et al. [50] proposed an approach for generating test programs following grammar for testing Java JIT compilers. Unlike all these techniques, LeJit generates Java programs by utilizing the structures of real-world Java programs and explores possibilities of expressions by inserting and refilling holes.

Mutation-based. A tool that implements a mutation-based approach generates new test programs by mutating existing programs, with several techniques and tools specifically for Java [5, 7, 8, 15, 43, 53]. Classfuzz [8] and Classming [7] are coverage-guided fuzzers that mutate seeds by modifying their syntactic structures and control/data flows. VECT [15] improves JavaTailor [53] via grouping by code vectorization code ingredients, which are collected from history bug-revealing programs in OpenJDK tests, to insert into seed programs. JOpFuzzer [20] uses JIT optimization options as a new dimension for test input in conjunction with profile data of JIT compilers for guiding the fuzzing process. It relies on Java* Fuzzer to generate seed programs. ComFuzz [49] is a recent compiler fuzzing framework that leverages existing test cases that have generated compiler bugs and deep learning to learn language features that are more likely to generate such bugs. SJFuzz [45] is a JVM fuzzer that aims to address the lack of guidance in fuzzing due to the absence of well-designed seeds and mutator scheduling. It aims to automate the process of JVM differential testing via mutating class files using control flow mutators to identify inter-JVM discrepancies. Artemis [24] explores various JIT compilation traces as differential testing, via diverse combinations of optimized/de-optimized method invocations. Although LeJit is similar in nature to concepts in mutation testing [12], via extracting templates and then using them to generate concrete programs, LeJit has several differences. (1) Mutation testing uses a predefined set of mutation operators. However, each hole in a template has its own set of values (and the set can be dynamically determined; see the next point). (2) LeJit fills holes dynamically (rather than

statically), which brings unique advantages: (a) uses meaningful variables to fill holes; (b) allows hole construction with runtime information, e.g. length of array; (c) allows LeJit to establish which holes are in dead code and, in the generation phase, focus on exploring the reachable holes. (3) LeJit can fill multiple holes simultaneously, which is more similar to higher-order mutation [21], during the execution of the template. (4) LeJit introduces an approach for obtaining objects necessary for running templates.

Template-based. Ching and Katz [9] use templates constructed from existing code bases to generate concrete programs for testing the APL-to-C compiler COMPC. Zhang et al. [52] statically mutate `int` variable occurrences in existing programs for testing C/C++ compilers. The original JAttack work presented an automated template extraction technique [51]. LeJit's pool-based extraction is similar to this previous approach, but contains several key differences. (1) JAttack converts expressions with limited types, i.e., `boolean`, `int`, `double`, and skips expressions in constructors, while LeJit converts all expressions to all holes supported; (2) JAttack converts only static methods while LeJit can convert any method into a template; (3) JAttack searches in all classes of the project for a public constructor or static method, with no parameter or only primitive parameters, to create reference argument for the template, and uses `null` if it cannot find such a constructor or static method. In contrast, LeJit uses test generation to generate method sequences that create desired objects.

Test input generation. JUnit tests can be automatically generated using Randoop [33] and EvoSuite [14] for a given set of classes under test. In theory, the tests generated from such tools can be directly used to test Java JIT, and we implemented such a prototype using Randoop and evaluated its bug-discovery effectiveness within our evaluation of LeJit (see Section 4.2). Popularized by QuickCheck [10], another approach is to allow developers to write generators from which valid test inputs can be obtained. Several approaches [3, 4, 11, 17] use the generators to exhaustively enumerate all possible paths through the generators up to a given bound. Theoretically, these tools can be used to test Java JIT as well, but they would require developers to manually write generators for Java programs that include various language features; writing those generators manually can be a tedious process. In contrast, LeJit allows generating programs by concretizing templates extracted from open-source projects without developer intervention.

7 CONCLUSION

We presented a framework, LeJit, which enables fully automated end-to-end template-based testing of JIT compilers. LeJit can create a template from any Java method, and it automatically inserts holes and generates necessary arguments for the template. To obtain instances of complex types needed for extracted templates, LeJit uses novel techniques built on automated test generation. We have extensively evaluated LeJit by generating 90,916 programs and discovered 15 bugs in three popular and widely used compilers. Our findings show the power of automating template extraction via LeJit and the power of scaling experiments without humans in the loop, as well as complementary power compared to the state-of-the-art tools for JIT and JVM testing techniques. We believe that LeJit should become an integral part of continuous testing for any Java JIT compiler.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Yu Liu, Pengyu Nie, Jiyang Zhang, and the anonymous reviewers for their comments and feedback. This work is partially supported by a Google Faculty Research Award, a grant from the Army Research Office, and the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027.

REFERENCES

[1] Azul Systems, Inc. 2018. *AzulSystems/JavaFuzzer: Java* Fuzzer for Android*.* https://github.com/AzulSystems/JavaFuzzer.

[2] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *International Conference on Software Engineering*. ACM, 550–561. https://doi.org/10.1145/2568225.2568248

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis*. ACM, 123–133. https://doi.org/10.1145/566171.566191

[4] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded Exhaustive Test-Input Generation on GPUs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 94:1–94:25. https://doi.org/10.1145/3133918

[5] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Programming Language Design and Implementation*. ACM, 183–198. https://doi.org/10.1145/3519939.3523427

[6] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

[7] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *International Conference on Software Engineering*. IEEE, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Programming Language Design and Implementation*. ACM, 85–99. https://doi.org/10.1145/2908080.2908095

[9] Wai-Mee Ching and Alex Katz. 1993. The Testing of an APL Compiler. In *International Conference on APL*. ACM, 55–62. https://doi.org/10.1145/166197.166205

[10] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming*. ACM, 268–279. https://doi.org/10.1145/351240.351266

[11] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 185–194. https://doi.org/10.1145/1287624.1287651

[12] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[13] Eclipse Foundation, Inc. 2024. *The order of super interface initialization in J9 is strange - Issue #13242 - eclipse-openj9/openj9.* https://github.com/eclipse-openj9/openj9/issues/13242.

[14] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[15] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *International Symposium on Software Testing and Analysis*. ACM, 526–537. https://doi.org/10.1145/3597926.3598075

[16] GitHub, Inc. 2023. *GitHub*. https://github.com.

[17] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test Generation through Programming in UDITA. In *International Conference on Software Engineering*. ACM, 225–234. https://doi.org/10.1145/1806799.1806835

[18] James Gosling and Greg Bollella. 2000. *The Real-Time Specification for Java*. Addison-Wesley.

[19] JavaParser.org. 2024. *JavaParser - Home*. https://javaparser.org.

[20] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *International Conference on Software Engineering*. IEEE, 43–55. https://doi.org/10.1109/ICSE48619.2023.00016

[21] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 249–258. https://doi.org/10.1109/SCAM.2008.36

[22] JUnit. 2022. *JUnit - About*. https://junit.org/junit4/.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Conference on Computer and Communications Security*. ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[24] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *Symposium on Operating Systems Principles*. ACM, 66–79. https://doi.org/10.1145/3600006.3613140

[25] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 196:1–196:25. https://doi.org/10.1145/3428264

[26] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[27] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random Testing of C Compilers Based on Test Program Generation by Equivalence Transformation. In *Asia Pacific Conference on Circuits and Systems*. IEEE, 676–679. https://doi.org/10.1109/APCCAS.2016.7804063

[28] Oracle Corporation and/or its affiliates. 2021. *The Java HotSpot Performance Engine Architecture*. https://www.oracle.com/java/technologies/whitepaper.html.

[29] Oracle Corporation and/or its affiliates. 2023. *[JDK-8280126] C2: detect and remove dead irreducible loops - Java Bug System*. https://bugs.openjdk.java.net/browse/JDK-8280126.

[30] Oracle Corporation and/or its affiliates. 2023. *[JDK-8280126] C2: detect and remove dead irreducible loops - Java Bug System*. https://bugs.openjdk.org/browse/JDK-8280126?focusedCommentId=14476253&page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel#comment-14476253.

[31] Oracle Corporation and/or its affiliates. 2023. *openjdk/jdk: JDK main-line development*. https://github.com/openjdk/jdk.

[32] OW2. 2024. *ASM*. https://asm.ow2.io.

[33] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering*. IEEE, 75–84. https://doi.org/10.1109/ICSE.2007.37

[34] Priya Khaira-Hanks. 2023. *What is a Java Uber-JAR and Why Is It Useful?* https://blog.payara.fish/what-is-a-java-uber-jar.

[35] Emin Gün Sirer and Brian N. Bershad. 2000. Using Production Grammars in Software Testing. In *Conference on Domain-Specific Languages*. ACM, 1–13. https://doi.org/10.1145/331960.331965

[36] The Apache Software Foundation. 2023. *commons-codec/BinaryCodec.java*. https://github.com/apache/commons-codec/blob/4de60e/src/main/java/org/apache/commons/codec/binary/BinaryCodec.java.

[37] The Apache Software Foundation. 2023. *commons-codec/StringUtils.java*. https://github.com/apache/commons-codec/blob/4de60e/src/main/java/org/apache/commons/codec/binary/StringUtils.java.

[38] The Apache Software Foundation. 2023. *commons-math/AdamsNordsieckTransformer.java*. https://github.com/apache/commons-math/blob/dff1a0/src/main/java/org/apache/commons/math4/ode/nonstiff/AdamsNordsieckTransformer.java.

[39] The Apache Software Foundation. 2023. *commons-math/Quaternion.java*. https://github.com/apache/commons-math/blob/dff1a0/src/main/java/org/apache/commons/math4/complex/Quaternion.java.

[40] The Apache Software Foundation. 2023. *commons-text/StrBuilder.java*. https://github.com/apache/commons-text/blob/e62203/src/main/java/org/apache/commons/text/StrBuilder.java.

[41] The Apache Software Foundation. 2023. *Maven - Welcome to Apache Maven*. https://maven.apache.org/.

[42] The JUnit Team. 2023. *JUnit 5*. https://junit.org/junit5/.

[43] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing A Test Corpus with Bonsai Fuzzing. In *International Conference on Software Engineering*. ACM, 723–735. https://doi.org/10.1109/ICSE43902.2021.00072

[44] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *International Conference on Software Engineering*. IEEE, 56–68. https://doi.org/10.1109/ICSE48619.2023.00017

[45] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. SJFuzz: Seed & Mutator Scheduling for JVM Fuzzing. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 1062–1074. https://doi.org/10.1145/3611643.3616277

[46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*. ACM, 283–294. https://doi.org/10.1145/1993316.1993532

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2021. *GCC Bug List Found by Random Testing (Total 79)*. https://embed.cs.utah.edu/csmith/gcc-bugs.html.

[48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2021. *LLVM Bug List Found by Random Testing (Total 203)*. https://embed.cs.utah.edu/csmith/llvm-bugs.html.

[49] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Tan, Hwei, Bo Zhang, Wenxiang Qian, and Wang Zheng. 2023. A Generative and Mutational Approach for Synthesizing Bug-exposing Test Cases to Guide Compiler Fuzzing. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 1127–1139. https://doi.org/10.1145/3611643.3616332

[50] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *International Conference on Quality Software*. IEEE, 20–23. https://doi.org/10.1109/QSIC.2003.1319081

[51] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *International Conference on Automated Software Engineering*. ACM, 23:1–23:13. https://doi.org/10.1145/3551349.3556958

[52] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Programming Language Design and Implementation*. ACM, 347–361. https://doi.org/10.1145/3140587.3062379
[53] 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 *International Conference on Software Engineering*. ACM, 1133–1144. https://doi.org/10.1145/3510003.3510059

Received 2023-09-29; accepted 2024-01-23