Compiler Testing using Template Java Programs

Zhiqiang Zang
The University of Texas at Austin
Austin, Texas, USA
zhiqiang.zang@utexas.edu

Nathan Wiatrek
The University of Texas at Austin
Austin, Texas, USA
nwiatrek@utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, Texas, USA
gligoric@utexas.edu

August Shi
The University of Texas at Austin
Austin, Texas, USA
august@utexas.edu

ABSTRACT

We present JAttack, a framework that enables template-based testing for compilers. Using JAttack, a developer writes a template program that describes a set of programs to be generated and given as test inputs to a compiler. Such a framework enables developers to incorporate their domain knowledge on testing compilers, giving a basic program structure that allows for exploring complex programs that can trigger sophisticated compiler optimizations. A developer writes a template program in the host language (Java) that contains holes to be filled by JAttack. Each hole, written using a domain-specific language, constructs a node within an extended abstract syntax tree (eAST). An eAST node defines the search space for the hole, i.e., a set of expressions and values. JAttack generates programs by executing templates and filling each hole by randomly choosing expressions and values (available within the search space defined by the hole). Additionally, we introduce several optimizations to reduce JAttack's generation cost. While JAttack could be used to test various compiler features, we demonstrate its capabilities in helping test just-in-time (JIT) Java compilers, whose optimizations occur at runtime after a sufficient number of executions. Using JAttack, we have found six critical bugs that were confirmed by Oracle developers. Four of them were previously unknown, including two unknown CVEs (Common Vulnerabilities and Exposures). JAttack shows the power of combining developers' domain knowledge (via templates) with random testing to detect bugs in JIT compilers.

CCS CONCEPTS

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

KEYWORDS

Testing, test generation, program generation, compiler, templates

ACM Reference Format:
Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556958

1 INTRODUCTION

Compilers are among the most critical components in the software development toolchain, and their correctness is of utmost importance. A bug in a compiler might lead to a crash during the translation, an incorrect output (native code does not match the semantics of the program written by developers [4]), or even expose security vulnerabilities in the generated code.

Compiler developers have manually written thousands of tests, i.e., programs in the compiler's target programming language, as to check for correctness [5, 48, 79]. Although manually-written tests nicely capture *developers' intuition* of what programs are expected to trigger corner cases, it is time-consuming to write a large number of such tests. As a result, researchers and practitioners have developed a number of automated techniques for testing compilers [14, 22, 23, 37, 42, 59, 64, 74, 94, 98], namely by generating a large number of programs on which the compiler can run. Existing compiler-testing techniques mainly fall in two categories: grammar-based [42, 64, 94, 98] and mutation-based [22, 23, 59, 74]. The former group generates programs from scratch following the production rules available in the language grammar. The latter group usually starts with some seed programs and then genetically mutates the seeds.

Although existing approaches are valuable, they have shortcomings. First, they provide limited ways for compiler developers to fully embed their knowledge into the testing process. Second, they frequently support only a subset of the language grammar (e.g., only integer values [94]).

We present JAttack, a framework that enables *compiler testing using templates*. Using JAttack, a developer writes a *template program* (template for short) that describes a *set of concrete programs* to be used as inputs to a compiler. Unlike prior work, *our framework enables developers to express richer manual tests for compilers*. Our design of a template captures the developers' intuition in very much the same way as manually-written tests but provides an opportunity to express variants of those tests that can be obtained by *testing* the templates. The goal is similar to parameterized unit testing [89], where developers manually write unit tests that encapsulate some features they want to test in their code but have parameters that a backend framework explores as to obtain deeper testing around the insights the developers initially provide. Unlike with mutation-based fuzzers, compiler developers can use templates to specify exactly how to generate program variants. (Figure 1 shows an example template, which is discussed in detail in Section 2.) JAttack complements existing automated compiler-testing techniques that can provide a structure of a program on which JAttack can further build templates.

In JAttack, a developer writes a template in the host language (Java), which contains *holes* to be filled by JAttack. Each hole is written in a domain-specific language (DSL) embedded in the host language, i.e., we do not change the syntax, compiler, nor runtime environment of the host language. We define the DSL as a set of APIs that allow developers to specify characteristics of the hole they want explored in a template, where each API call produces an instance of an *extended abstract syntax tree* (eAST) node; an eAST node bounds the search space for the hole, i.e., defines a set of possible statements, expressions, and values. As an example, consider the following API call that defines a hole: `relation(intVal(), intVal(), GT, LT).eval()`, which represents a logical relation between two integer literals (each can take any value between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`) using either > (`GT`) or < (`LT`) relational operators; this hole evaluates to a boolean. Using the JAttack's API, each hole can then be type-checked by the host compiler.

JAttack is useful for augmenting testing for many complex compiler features as it leverages the developer insights from the provided templates. For our evaluation, we focus specifically on testing *just-in-time* (JIT) compilers. Unlike traditional ahead-of-time compilers that translate a program into native code prior to deployment [1], JIT compilers translate the program during execution [7]. Certain optimizations only occur after executing specific program structures a sufficient number of times.

JAttack takes two inputs: (1) a template, and (2) an iteration count (*N*), i.e., the number of times each generated program will be executed in a loop to ensure that JIT compilation is triggered. JAttack generates a program by repeatedly *executing* the template (up to *N* times) and filling each hole, when the hole is reached the first time, by *randomly* choosing expressions and values available within the search space defined by the hole. (In theory, a template can be exhaustively explored, but it is generally not feasible.) Next, to detect any JIT-related bugs, each generated program is executed *N* times using different JIT compilers, potentially detecting bugs via differential testing [66].

We also introduce three optimizations into JAttack to reduce the generation cost. The first optimization, *early stop*, involves stopping after detecting that further generation would not fill any more holes. The second optimization, *hot filling*, dynamically transforms the template when a hole is reached the very first time; the API call is transformed into the concrete expression that the call would produce. The final optimization, *eager pruning*, uses a modern constraint solver (Z3 [29]) to detect holes for conditional statements (e.g., `if`) that always evaluate to a constant value.

To demonstrate JAttack's capabilities in testing JIT compilers, we wrote 23 templates. We focused on interesting Java language features and took inspiration from existing tests for the Java compiler. We report the cost of generation and execution, as well as benefits of our optimizations; our optimizations reduce the *generation time* by 99.50%. We used the generated programs as inputs to multiple commercial JIT compilers, including the Oracle JDK JIT compiler. Using just our own templates, we were able to discover two bugs in the Oracle JDK JIT compiler. These bugs were confirmed and fixed by Oracle developers, and one of the bugs was previously unknown and acknowledged on Oracle's list of CVEs.

We also evaluated how well JAttack can be used for automated compiler testing by *extracting templates from existing Java projects*. This evaluation is inspired by mutation testing [6, 23, 30, 55, 57, 59, 60], where we essentially "mutate" existing code to construct different tests for compilers. Note that, unlike in traditional mutation testing, holes in our case are filled by randomly generating values and expressions. Moreover, mutation testing uses a predefined set of mutation operators while JAttack provides a way for a user to write holes anywhere in their code following their intuition (e.g., the way we wrote templates for JIT compilers). Each hole has its own set of values (and the set is determined by the developer, not by a tool).

Using 77 open-source Java projects that span a wide variety of domains and therefore use of different Java language features, we automatically extracted 5,419 templates. By running these templates through JAttack, we found four more bugs in the Oracle JDK JIT compiler (out of which only one was previously known) including one previously unknown CVE recently acknowledged by Oracle.

The key contributions of this paper include:

- **Framework**. We introduce JAttack, the framework for templating tests for compilers. JAttack is designed to complement manually-written tests and blend developer's intuition (via templates) and random testing to increase likelihood to detect bugs in Java JIT compilers.
- **Programming and execution models**. We introduce a programming and an execution model to integrate templates entirely in the host language (Java), without changing the syntax or the runtime environment. Templates are like manually-written programs with holes; each hole, expressed using a DSL, builds an eAST node that specifies values that the hole can take (i.e., defines a search space). We introduced three optimizations that are applied when generating programs from templates.
- **Use case**. We implemented JAttack for the Java programming language and applied it to testing Java JIT compilers. We evaluated JAttack by writing 23 template programs. Our results show that the optimizations substantially reduce test generation time, making JAttack practical. Furthermore, we discovered two bugs in the Oracle JDK JIT compiler.
- **Template extraction**. We evaluated JAttack as an automated framework for JIT compiler testing by automatically extracting templates from existing Java projects. Using 5,419 templates from 77 open-source Java projects, JAttack discovered four more bugs in the Oracle JDK JIT compiler.

JAttack is available at https://github.com/EngineeringSoftware/jattack.

2 EXAMPLE

Figure 1a shows a template program that we wrote while developing JAttack for Java. Our motivation for this template was to exercise

```
1  import static jattack.Boom.*;
2  public class C {
3    static int s1;
4    static int s2;
5    @Entry
6    public static int m() {
7      int[] arr1 = { s1++, s2,  intVal().eval() ❶,
8                       intVal().eval() ❷,  intVal().eval() ❸ };
9      for (int i = 0; i < arr1.length; ++i)
10       if ( logic(relation(intId(), intId(), LE),
11                  relation(intId(), intId(), LE),
12                  AND, OR).eval() ❹)
13         arr1[i] &=
14           arithmetic(intId(), intId(), ADD, MUL).eval() ❺;
15     return 0; } }
```

(a) An example of a template.

```
1  import static jattack.Boom.*;
2  public class C {
3    static int s1;
4    static int s2;
5    @Entry
6    public static int m() {
7      int[] arr1 = { s1++, s2,  45350238 ❶,
8                       681339300 ❷,  125652422 ❸ };
9      for (int i = 0; i < arr1.length; ++i)
10       if ( arr1[3] <= s2 || s2 <= arr1[2] ❹)
11         arr1[i] &=  arr1[1] * s1 ❺;
12     return 0; } }
```

(b) An example of a generated program.

Figure 1: An example of a template and one of the generated programs from the template.

Java JIT optimizations for programs that use local arrays and static variables. It is important to note that *every template for JAttack is a valid Java program*. This template uses static methods (e.g., `logic`) that are defined in the `jattack.Boom` class. As such, the Java compiler can also type-check the template.

The template contains five *holes* representing places where JAttack should generate expressions, filling them in to create a concrete *generated program*. Three holes are between lines 7 and 8, one between lines 10 and 12, and one on line 14. The number of holes is equal to the number of `eval` invocations. The `eval` invocation as well as the type information of the expression calling the `eval` allows JAttack to tell a hole from actual code.

The first three holes are defined by the `intVal` method calls; each call to `intVal` represents a hole that will be filled by an integer literal; note that without any arguments, `intVal` produces an integer between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. The next hole (lines 10-12) defines a logical "and" or "or" expression (`logic` with the `AND` and `OR` arguments) between two relational expressions. Each relational expression (`relation`) connects two free integer variables `intId`, which can be `s1`, `s2`, `i`, or any element of `arr1` (the array index is randomly picked between 0 and the size of the array) at this point, using the <= operator (`LE`). The final hole (line 14) is an arithmetic expression (`arithmetic`) of two free integer variables, which are combined using either a + or a $*$ operator (`ADD` or `MUL`).

We describe more details on what type of expressions we support for holes along with our API in Section 3.

JAttack *generates* programs through an *execution-based* model. In other words, JAttack fills the holes in a template after executing the template. (Unlike static generation, an execution-based model prunes the search space by only filling holes reached during execution. See Section 7.1 for details.) A template must have a template entry method, annotated with `@Entry` as shown in the example (method `m`). When the execution reaches any unfilled hole, JAttack generates a valid expression for that hole based on the used API calls. When all reachable holes are filled (see Section 3.3 for how this is determined), JAttack outputs the corresponding generated program. JAttack then calls the template entry method again to generate the next program up to the specified maximum number of programs. An example of a generated program that JAttack outputs for the template in Figure 1a is shown in Figure 1b. The numbered circles in the generated program correspond to the same ones next to holes in Figure 1a.

Finally, to detect any JIT-related bugs, JAttack *executes* each generated program starting from the same entry method a large number of times using different JIT compilers, potentially detecting bugs via differential testing [66]. The large number of re-executions is necessary as to trigger JIT optimizations. In Java, a JVM starts executing a program with an interpreter and monitors the execution for "hot" code sections, i.e., code that is frequently executed. The JIT compiler then optimizes "hot" sections. Through repeated executions of the generated method `m`, the generated program shown in Figure 1b revealed a bug in the Oracle JDK JIT compiler, crashing the JVM.

3 JATTACK FRAMEWORK

JAttack introduces test templating, a way to define a *set of programs* used for testing compilers. Templates written for JAttack could be useful for testing other program analysis tools, but we leave such studies for future work. We designed JAttack guided by the following requirements: (1) developers decide where the holes should be placed and bound the search space of each hole, and (2) the domain-specific language (DSL) for writing holes is non-intrusive, i.e., it requires no changes to the host compiler.

In this section, we describe our programming and execution models (Section 3.1), implementation for Java (Section 3.2), the generation procedure (Section 3.3), the optimizations for generation (3.4) and the overall JIT-testing procedure (Section 3.5).

3.1 Programming and Execution Models

We define the syntax and operational semantics of a simple imperative language with an extension to support templates. Note the language shown here includes only integer type for ease of presentation; we greatly extend the scope in our implementation for Java. The simple imperative language and extensions represent the foundations for supporting templates for general imperative languages, and our implementation in Java, described in later sections, is based on these extensions.

Syntax. Figure 2 defines the syntax of the simple language. A program is a sequence of zero or more statements. Each statement is either an assignment, conditional, goto, or halt. An expression in a

```
Stmts := (LABEL:)? Stmt ";" Stmts | ϵ
Stmt := AssignStmt | IfStmt | GotoStmt | halt
AssignStmt := ID "=" Exp
IfStmt := "if" "(" Exp ")" LABEL
GotoStmt := goto LABEL

Exp := ExpBasic | "[[" ExpAlt "]]"
ExpBasic := ExpBasic Op Operand | Operand
ExpAlt := ExpAlt "," ExpBasic | ExpBasic

Operand := ID | NUM | "[[c]]" | "[[v]]"
Op := "+" | "-" | "||" | "&&" | "==" | "!=" | "<"
```

Figure 2: Syntax for an imperative language with holes.

$$\text{ASSIGN\_EXP: } \frac{\langle pc+1, I, M, L\rangle exp \Rightarrow \langle pc', I', M, L\rangle v}{\langle pc, I, M, L\rangle id = exp \Rightarrow \langle pc', I', M, L\rangle id = v}$$

$$\text{ASSIGN\_VAL: } \frac{M' = M[v/id]}{\langle pc, I, M, L\rangle id = v \Rightarrow \langle pc+1, I, M', L\rangle I(pc+1)}$$

$$\text{GOTO: } \frac{}{\langle pc, I, M, L\rangle \text{goto } l \Rightarrow \langle L(l), I, M, L\rangle I(L(l))}$$

$$\text{IF\_EXP: } \frac{\langle pc+1, I, M, L\rangle exp \Rightarrow \langle pc', I', M, L\rangle v}{\langle pc, I, M, L\rangle \text{if}(exp)\ N \Rightarrow \langle pc', I', M, L\rangle \text{if}(v)\ N}$$

$$\text{IF\_VAL: } \frac{pc' = (v == 0)\ ?\ pc+1 : L(l)}{\langle pc, I, M, L\rangle \text{if}(v)\ l \Rightarrow \langle pc', I', M, L\rangle I(pc')}$$

$$\text{C\_HOLE: } \frac{val = alt(range(MIN, MAX)) \qquad I' = I[val/pc]}{\langle pc, I, M, L\rangle [[c]] \Rightarrow \langle pc, I', M, L\rangle val}$$

$$\text{V\_HOLE: } \frac{id = alt(identifiers(M)) \qquad I' = I[id/pc]}{\langle pc, I, M, L\rangle [[v]] \Rightarrow \langle pc, I', M, L\rangle id}$$

$$\text{E\_HOLE: } \frac{e = alt([e_1, e_2, ..., e_n]) \qquad I' = I[e/pc]}{\langle pc, I, M, L\rangle [[e_1, e_2, ..., e_n]] \Rightarrow \langle pc, I', M, L\rangle e}$$

Figure 3: Semantics for the simple language from Figure 2.

program can combine relational, arithmetic, and logical operators. On top of these basic imperative features, the language also introduces the concept of a hole, denoted with [[ ]]. These holes can be used around a sequence of comma-separated expressions, or they can be around individual operands, where [[c]] represents a hole for a literal/constant and [[v]] represents a hole for a variable.

Semantics (core language). Valid programs can only use integer literals. We define the state of a program with the following configuration: $\langle pc, I, M, L\rangle$, where pc is the program counter (initially 0), I is the instruction memory (i.e., mapping from program counter to statements or expressions), M is the main memory (i.e., mapping from identifiers to integer values), and L is a map from labels to indices in I. Prior to the execution, statements and expression indices are placed into I by performing a pre-order traversal of the program's abstract syntax tree (first statement is at index 0). Also, L is initialized to map each label to the appropriate index in I. We also use the following operations: (1) map lookup $\_(val)$, and (2) map update $\_[val/\_]$.

Figure 3 shows the operational semantics of the language. For simplicity, the rules do not include error handling. The assignment statement simply updates the value of a variable in memory. The conditional statement evaluates the expression and then jumps if the expression evaluates to true ($val \neq 0$). The goto statement unconditionally jumps to the statement with the specified label. We do not show the rules for computing basic expressions, as we assume the same semantics as in the C programming language. The halt statement terminates the execution.

Semantics (template language). We define several utility functions: (1) `identifiers`(M) - returns a list of available variable names in M at the point of an invocation, (2) `range`(x, y) - returns a list of integers between x and y, and (3) `alt`([...]) - takes a sequence as input and outputs one of its elements.

A hole for an integer literal (C_HOLE) evaluates to an integer literal and rewrites itself to that literal. A hole for a variable (V_HOLE) evaluates to an available identifier and rewrites itself to that identifier. Finally, a hole for an expression (E_HOLE) evaluates to one of the given expressions (and rewrites itself to that expression). Note that the rewrite rules are such that the entire hole is replaced with the choice of a concrete expression upon execution, so the hole no longer exists after the first evaluation, ensuring that each hole evaluates only to a single expression per execution. The program in I at the end of execution is the generated program in the same language as the original template.

Filling a hole. Given a list of candidates for a hole, we need to explore different candidates every time we execute the program, which would in turn rewrite the template into new concrete programs that we can later use for testing compilers. While one can try and systematically explore all the possible candidates, the search space can be incredibly large (e.g., for [[c]] the range of possible integers go from `MIN` to `MAX`), especially when considering combinations of candidates chosen across all holes in the program.

For this work, we choose candidates for a hole randomly. Random exploration has been found effective in prior work [64, 74, 94, 98]. We keep re-executing the template to rewrite into concrete programs up until we reach a specified limit for number of generated programs. Note that each execution of a template is independent of other executions, i.e., any modifications to the template during one run is *not* observable in another run.

Example. Consider the following example in our language: `s1 = [[c]]; s2 = [[c]]; if ([[v]] < [[v]]) l9; l9: halt;`. Executing this template once might generate: `s1 = 45350238; s2 = 681339300; if (s1 < s2) l9; l9: halt;`. Another execution can lead to a different generated program: `s1 = 125652422; s2 = 23297; if (s2 < s2) l9; l9: halt;`.

3.2 JAttack Implementation for Java

We implement the semantics of JAttack for the host Java programming language. To support the concept of holes while integrating it into Java, we introduce a set of API methods that construct holes.

Figure 4 shows a subset of the full API we provide. This API represents a DSL that maps to our simple imperative language. All methods in the API return an instance of a node rooting an *extended abstract syntax tree* (eAST). An `Exp<Integer>` node corresponds to an expression (evaluating to integer due to Java typing, so `Exp<Boolean>` is the same for boolean type). An `IntVal` (extends `Exp<Integer>`) node and an `IntId` (extends `Exp<Integer>`) node correspond to an integer literal and variable, respectively. We define `BoolVal` and `BoolId` to correspond to the boolean type for Java. `BAriExp<Integer>` (extends `Exp<Integer>`) is for binary arithmetic expressions, while `RelExp` (extends `Exp<Boolean>`) and `LogExp` (extends `Exp<Boolean>`) are for relational and logical expressions. These nodes are therefore

```
BoolVal boolVal();
IntVal intVal(int min, int max);
IntVal intVal();
BoolId boolId(String... names);
IntId intId(String... names);
...
<T> RefId<T> refId(Class<T> type, String... names);
<T extends Number> BAriExp<T> arithmetic(
    Exp<T> left, Exp<T> right, Op... ops);
<T extends Number> RelExp<T> relation(
    Exp<T> left, Exp<T> right, Op... ops);
LogExp logic(
    Exp<Boolean> left, Exp<Boolean> right, Op... ops);
...
<T> ExprStmt exprStmt(Exp<T> exp);
IfStmt ifStmt(
    Exp<Boolean> cond, Stmt thenStmt, Stmt elseStmt);
WhileStmt whileStmt(Exp<Boolean> cond, Stmt body);
...
<T> Exp<T> alt(Exp<T>... exps);
Stmt alt(Stmt... stmts);
```

Figure 4: API for writing holes; a call to any of the methods in the API instantiates an eAST node.

placeholders for the actual, concrete expressions to be generated at runtime, so they represent holes to be filled. For example, an `IntVal` node created using the method `intVal` represents [[c]], a hole that can evaluate to any integer from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`. We also provide an `intVal` API that can specify the range of integer values, and an `intId` API that can enumerate available variables (at any point) by analyzing bytecode, when no variable is specified.

While specific API methods can create corresponding nodes, e.g., `arithmetic` for `BAriExp`, we also provide method `alt` that can choose from a provided list of `Exp<Integer>` or `Exp<Boolean>` nodes, like the semantics for an expression hole (E_HOLE) in our simple imperative language.

Although we illustrate the implementation of JAttack using `int` and `boolean`, we support any other primitive type, e.g., `long` and `double`, or any reference type. For instance, `refId` can enumerate available variables (at any point) with type `T` that can be specified using argument `Class<T> type`, e.g., `refId(String.class)` returns a `RefId<String>` node corresponding to any available `String` variables at this execution point. We provide API methods to create statements as well, such as `exprStmt`, `ifStmt` and `whileStmt`. Furthermore, one can extend our implementation to include more language constructs in Java, as along as they can be represented in an eAST.

As an alternative, we originally designed our API to use a list of concrete Java expressions to choose from, e.g., `alt(i++, j++)`. However, these expressions would get executed and result in side-effects, and the final execution would not match executing the corresponding generated program with the concrete expressions substituting for the hole, so we abandoned that direction.

Instead, when using eAST nodes, we do not actually generate an expression to fill a hole until the `eval` method is invoked on the node, e.g., `intVal().eval()`. Only after calling `eval` does a concrete node get generated for that hole. Once generated, the node is interpreted to compute the result of the expression. Furthermore, all subsequent calls to the same API method (from the same location) will always return the same node. For our Java implementation, we define a hole to be where the developer calls `eval` for a built eAST. The eAST constructed for an API call represents a range of candidates to fill the hole. As an example, consider the hole specified by the `logic` call (lines 10-12 in Figure 1a). Executing the `logic` method returns a root node of an eAST, illustrated in Figure 5. Candidates for the hole are obtained by recursively obtaining candidates for nodes in subtrees and combining them together.

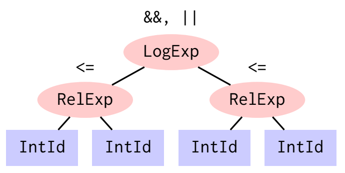

Figure 5: eAST corresponding to `logic` hole from Figure 1a.

In this example, the `RelExp` nodes would result in candidates that combine choice of integer variables combined with the specified operators (just `LE` in the example); the top-level `LogExp` node would use the returned candidates and combine with the specified `AND` or `OR` to create the final candidates. This eAST structure corresponds to an expression hole in our imperative language, namely the following:
[[[[v]] <= [[v]] && [[v]] <= [[v]], [[v]] <= [[v]] || [[v]] <= [[v]]]].
Unlike our simple imperative language, our API provides syntactic sugars to describe a large set of similar candidates without having to enumerate all of them by specifying multiple operators at once (see `Op... ops` in Figure 4).

3.3 Generation Procedure

Figure 6 shows the overall algorithm for JAttack's `Generate` function that executes a template repeatedly to generate concrete program instances. The input to `Generate` is a template T and the number of programs to generate N. The output is a set of generated programs G.

Function `Generate` starts by initializing the empty set of generated programs G and then capturing the initial global state of the template T into variable S (line 6). We currently support capturing static fields with primitive and array types as the global state; future work could also capture reference types [9, 10]. We capture the global state to be used later when generating programs, ensuring the generation of each (out of N) program is done from the clean state. (We use the Java reflection mechanism to capture the state.) Additionally, `Generate` finds the template entry method (line 7), which is the entry point for executing T (in our Java implementation, this is the method annotated with `@Entry`), and also counts the total number of holes that should be filled in the template (line 8).

Next, `Generate` repeatedly calls `RunTemplate`, which executes the template, resulting in a generated program that is added to G. Assume that a template always terminates, which can be guaranteed through carefully specifying the search space for the holes in conditions, the overall loop in `Generate` ends when the number of uniquely generated programs has reached the maximum necessary number N. We set a timeout, for `RunTemplate`, as it might not be feasible to generate the specified number of unique programs.

Before calling `RunTemplate` in each iteration, `Generate` sets the global state to be the same as the initial global state S (line 10). Setting the initial state to S ensures that subsequent runs of `RunTemplate` always start the generation process, which executes the same template entry method, in the clean state.

Example. Consider the template from Figure 1a. The template has a static variable `s1` that is modified (line 7). Subsequent executions should make sure `s1` starts at `0` again, otherwise they would not be starting at the same state and would not generate programs that are even possible.

Function `RunTemplate` is responsible for generating a single concrete program from the given template T. First, it initializes H as an empty mapping from holes to their filled expressions (line 21). `RunTemplate` then sets an intermediate program Q to be the input template program T to start with (line 23), and then it repeatedly executes the entry method *entryMeth* on Q (line 25). The `ExecEntryMethod` returns a mapping H' of holes it filled to the actual expressions.

Example. In Figure 1a, executing the hole on line 7 would result in a mapping of that hole to concrete value 45350238 (line 7 in Figure 1b).

The overall mapping H gets updated with H'. If all holes have been filled, then the loop terminates (line 27). The reason for executing the template entry method *entryMeth* many times is to ensure all holes that can be reached get filled. Eventually, our goal is to execute a corresponding generated entry method up to `MAX_NUM_ITERATIONS` times as to trigger JIT optimizations (Section 3.5). Some holes may only be reachable after multiple iterations, so executing just once would not fill those holes.

Example. Consider the template from Figure 1a. The last hole (line 14) could be skipped in the first run because the condition (line 10-12) is evaluated to `false`. However, the hole could be filled later when static variable `s1` gets updated (line 7), making the condition `true`.

Some choice of candidate for a hole may possibly make another, later hole unreachable, putting it in dead code. JAttack may fill a hole in a condition, such as for an `if` statement, that always evaluates to `false`, and therefore any holes within the block of these conditional statements cannot be reached. To prevent the execution from `RunTemplate` from continuously executing while being unable to fill those unreachable holes, `RunTemplate` stops after the `MAX_NUM_ITERATIONS` maximum number of iterations. Having unfilled dead-code holes in a generated program is fine because such code should never even be executed within the maximum number of iterations later (and if it is executed, that would indicate a bug in the JIT compiler). `RunTemplate` does stop earlier when all holes are filled (line 27).

Three optimizations are introduced to reduce generation cost (line 28-32) and we describe the optimizations in detail in Section 3.4. Note that Q is an intermediate program, and we do not directly return Q. As such, we can optimize and make extra changes in Q to speed up generation, and these changes do *not* belong in a final generated program P.

The final returned program P is then the original template T with all its holes filled using the mapping H (computed using function `ApplyFilledHoles` in line 33, which is not shown). Essentially, each node corresponding to a filled hole can output the concrete code snippet for the expression it currently holds, and the hole expression in the template T gets replaced with this concrete code snippet. `Generate` then takes the returned program P and adds it to the running set of generated programs G. Note that `Generate` will keep calling `RunTemplate` until obtaining a sufficient number of programs; each time, `Generate` will use the fresh template program T, which has no filled holes, as to create a brand new generated program.

```
 1: Input: template program $T$
 2: Input: number of programs to generate $N$
 3: Output: set of generated programs $G$
 4: function GENERATE($T$, $N$)
 5:     $G \leftarrow \emptyset$
 6:     $S \leftarrow$ CAPTUREGLOBALSTATE($T$)
 7:     $entryMeth \leftarrow$ FINDENTRYMETHOD($T$)
 8:     $num \leftarrow$ COUNTHOLES($T$)
 9:     repeat
10:         RESETGLOBALSTATE($S$)
11:         $P \leftarrow$ RUNTEMPLATE($T$, $entryMeth$, $num$)
12:         $G \leftarrow G \cup \{P\}$
13:     until $|G| = N$
14:     return $G$
15:
16: Input: template program $T$
17: Input: entry method $entryMeth$
18: Input: number of holes $num$
19: Output: generated program $P$
20: function RUNTEMPLATE($T$, $entryMeth$, $num$)
21:     $H \leftarrow \{\}$
22:     $seenStates \leftarrow \emptyset$
23:     $Q \leftarrow T$
24:     for $i \leftarrow 1$ to $MAX\_NUM\_ITERATIONS$ do
25:         $H' \leftarrow$ EXECENTRYMETHOD($entryMeth$, $Q$)
26:         $H \leftarrow H \cup H'$
27:         if $|H| = num$ then break
28:         $R \leftarrow$ CAPTUREGLOBALSTATE($Q$)
29:         if $R \in seenStates$ then break
30:         $seenStates \leftarrow seenStates \cup \{R\}$
31:         $Q \leftarrow$ HOTFILL($Q$, $H$)
32:         $Q \leftarrow$ REMOVEDEADCODE($Q$, $H$)
33:     return APPLYFILLEDHOLES($T$, $H$)
```

Figure 6: Generation algorithm.

3.4 Optimizations for Generation

We develop three optimizations to speed up the generation process. Note that these optimizations all apply only for a single run of `RunTemplate`, to just a single generated program at a time. Also, these optimizations do *not* impact the generated programs, they only speed up the generation process.

Early stop. We can have an even earlier stopping condition based on the insight that if the global state after execution is the same as an already seen state, then any future run would lead to the same behavior (as a previous execution). Starting execution in the same global state cannot lead to new executions that fill new holes. In `RunTemplate`, we keep track of the seen global states in `seenStates` and check the global state after each execution (line 29 in Figure 6). This type of program state hashing has been extensively used in software model checking [50].

Hot filling. In our preliminary experiments, we found that executing a template entry method many times is time-consuming, especially compared to executing the generated entry method as part of our evaluation. The extra overhead comes from repeated executions of our Java API methods that build and evaluate eAST nodes. Recall during generation the filled hole is not rewritten into the concrete expression, but just evaluated to produce the same value as the concrete expression. The filled holes get replaced with the actual code only when the entire template gets translated into a new generated program (line 33 in Figure 6). Thus, while our implementation ensures that repeated execution of the same API method returns the same eAST node, invoking the `eval` method to evaluate the node is still expensive compared to evaluating the concrete code that replaces the hole in the generated program.

The hot filling optimization replaces the hole at runtime (during generation) with the concrete expression when the hole is evaluated for the first time, so that execution in the following iterations can use concrete code rather than invoking our Java API methods, including the `eval` method that evaluates the filled hole. In `RunTemplate`, we invoke the method `HotFill` (line 31 in Figure 6) on the resulting Q after execution that finds all API calls with set nodes and, using the mapping of holes to expressions H, replaces those calls with the concrete expressions. Then, using interfaces provided from package `javax.tools`, e.g., `javax.tools.JavaCompiler`, we implement an in-memory Java compiler, file manager, and associated class loader to dynamically compile Q and then reload this modified template's class, resulting in a new Q. The next iteration starts from the new Q as the template (line 25 in Figure 6). This technique is conceptually similar to "quickening" optimization implemented in self-optimizing interpreters [16, 45].

Eager pruning. In our preliminary experiments, we also noticed a significant number of generated programs with conditional expressions that are trivially false, e.g., `(var1 > var1)`. The body of such conditional statements would never be executed, so it is unnecessary to execute any further to fill holes within statements guarded by that condition. After executing the template entry method and obtaining filled holes in H, we invoke function `RemoveDeadCode` to eliminate any such dead code in the program Q (line 32 in Figure 6), completely rewriting the body into an empty statement. This technique is conceptually similar to partial evaluation [56]. We leverage a modern SMT solver (Z3 [29]) in our implementation to determine whether any conditional expression is satisfiable or not, eliminating code in case the expression is unsatisfiable. Note that we only temporarily remove the code as a means to speed up generating a single program. The returned generated program does not have any unreachable code removed. Later calls to `RunTemplate` always start with the same template T that has all the code still there.

3.5 JIT-Testing Procedure

Revealing JIT-related bugs requires not just programs but also executing those programs many times. For each generated program, we iterate though different JIT compilers. For each JIT compiler, we repeatedly execute the generated entry method, hashing the output of each execution (a generated entry method's return value is always encoded into an integer) into a running total. After executing `MAX_NUM_ITERATIONS` times (the same limit in Figure 6), we capture the global state of generated program (values of all static fields) and encode it within a checksum value, adding this to the running total. The final total representing the combination of all the executions.

For a given generated program, we use differential testing [66] to check if the running totals computed from all JIT compilers are all the same. Any difference should indicate that the generated program detected a bug within some JIT compiler. However, the program may itself be non-deterministic, i.e., having different outputs when run multiple times on the same JIT compiler. Note that non-determinism was only observed from the templates extracted from existing Java projects, e.g., involving randomness, while our manually created templates are guaranteed to be deterministic. To avoid being misled by non-determinism, when there are differences in output across different JIT compilers, we choose a JIT compiler as a reference point and run the program twice using that same compiler. If the outputs from running on the same JIT compiler differ, then output differences between JIT compilers do not indicate a bug. While this step may potentially miss detecting some bugs, it gives higher guarantees that reported bugs are true bugs.

Besides checking for differences in final running totals, we also report a bug if the execution crashes on some JIT compiler. Executing any unfilled hole (left as the API method call in the generated program) would also trigger a crash, because an unfilled hole should not be reachable. The ultimate output of the entire JIT-testing procedure is a subset of generated programs that expose a bug in one of the input JIT compilers.

4 EXTRACTING TEMPLATES

We also evaluate JATTACK for automated end-to-end compiler testing. Namely, we provide an approach to automatically extract templates from existing Java projects. The code written for these projects are naturally representative of Java language features and can be used as the foundation for templates that can find bugs in Java JIT compilers. We can then also easily and automatically scale up the number of templates to run through JATTACK.

Given a Java class, we first parse all the available methods in the class to detect potential holes. For each statement, we recursively convert each subexpression into the corresponding hole, starting from the leaves of the expression tree. For example, the expression `a + b` would be converted into `arithmetic(intId(), intId()).eval()` (specifying no operator argument means using all valid operators), which matches the expression structure. Note that the final call to `eval` is on the outermost API call, allowing for the greatest space of combination of values that JATTACK can explore.

After inserting holes into the Java class, we then scan the class for available static methods, which are the candidate template entry methods. If the static method takes any parameters, we insert additional *parameter methods*, one for each parameter; a parameter method returns a concrete value for the corresponding parameter type upon execution. For primitive values, we leverage JATTACK to provide a possible value, e.g., if the parameter is an `int` type we simply use `intVal` to represent an integer value. For non-primitive types, i.e., classes, we search if such classes have default constructors or constructors with primitive arguments that we can simply use to create an instance of that class. If there are no such constructors, we search from other classes for a public static method that returns an instance of the class. If none of the above cases applies, we then use `null` as the concrete value.

Note that our approach for extracting templates and then using them to generate concrete programs is similar in nature to concepts in mutation testing [30], where existing programs are mutated into other similar programs through syntactic mutation operators [23, 59, 60]. Conceptually, converting a program into a template program and then generating additional programs through JAttack is like mutating the original program. However, our way of generating programs leverages the capabilities of JAttack and its DSL to allow more expressive transformations that are beyond traditional mutation operators, and more similar to what can be generated using higher-order mutation operators [54].

5 EXPERIMENTAL SETUP

We briefly describe our evaluation setup.

5.1 Evaluation Subjects

We wrote 13 templates that exercise Java language features. We also studied the available optimizations used in the Oracle JDK JIT compiler, creating six templates whose basic structure would trigger those optimizations while including holes for JAttack to explore. Finally, we studied existing bug reports for JIT-related bugs, creating four templates by modifying the programs attached to bug reports to include holes. In our evaluation, we refer to the templates based on our own understanding of Java and the compiler developers' intuition of optimizations using prefix "M". We refer to the templates based on bug reports using prefix "B". Overall, we created 23 templates, with the goal to evaluate the effectiveness of our optimizations.

To evaluate JAttack in the context of automated test generation, we collect templates automatically from existing Java code. We use 77 open-source Java Maven projects from GitHub to extract templates from their classes. Given a project or a module of a multi-module Maven project, we find classes defined in all ".java" files. We extract templates from these classes following the procedures described in Section 4.

5.2 Configuring JAttack

For each template we created ourselves, we configure JAttack to generate 1,000 concrete programs (`N` in Figure 6). While generation is fastest when we turn on all three generation optimizations (Section 3.4), we also evaluate running generation without any optimization and with each optimization separately, measuring each one's effectiveness. For each of the generated programs, we execute it 100,000 times (`MAX_NUM_ITERATIONS` in Figure 6) on different JIT compilers. The JIT compilers we evaluate on are Oracle JDK, OpenJDK, and OpenJ9, all based on JDK 11.0.8.

For templates extracted from existing Java projects, we follow the same approach, except we configure JAttack to generate only 10 concrete programs from each template program, because of the large number of templates, and we test only on the latest Oracle JDK, which was 16.0.2 at that time.

In our evaluation, we configure Oracle JDK to restrict the specific tiers, L1 and L4, using the option `-XX:TieredStopAtLevel`, in order to test C1 and C2 compilers [3], respectively. We treat each restricted tier configuration for Oracle JDK as conceptually a new JIT compiler for use in our JIT-testing procedure (Section 3.5).

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. For all time measurements, we run the evaluation five times and report the average of those times.

6 EVALUATION

We evaluate JAttack by asking:

RQ1: How efficient is JAttack at generating programs and executing those generated programs with different JIT compilers?

RQ2: How well can JAttack be used for *automated* compiler testing via extracted templates from a large number of existing Java programs, and how does it compare with state-of-the-art automated JIT compiler testing?

RQ3: What critical bugs does JAttack detect in JIT compilers?

We address RQ1 as to better understand how efficient JAttack is at generating programs from templates, as well as the impact of our optimizations for generation, and to understand the efficiency of JAttack's testing procedure. We address RQ2 to understand how well JAttack can be used for automated compiler testing and compare the effectiveness with tools used in industry. We address RQ3 to understand the bugs that we exposed. JAttack and all JIT-bugs we detected, including associated templates and generated programs, are available at https://github.com/EngineeringSoftware/jattack.

6.1 Performance and Optimizations

Table 1 shows the time for JAttack to generate 1,000 programs for each of our manually created 23 templates. The different columns show the total time to generate all 1,000 programs when using different generation optimizations (Section 3.4). Namely, "Non-opt." means no optimizations, "Early Stop" means using only early stop, "Hot Filling" means using only hot filling, and "Eager Pruning" means using only eager pruning. The final column for "Full Opt." is the time when using all optimizations. In addition to time, for each optimization column, we also show the percentage of time reduced relative to "Non-opt." time (the higher the reduction the better). The final row shows the sum of generation time across all templates and the overall reduction over this total time.

Without any optimizations, the total time for generation across all templates (essentially $1{,}000 * 23 = 23{,}000$ programs total) is over two days. We find that the overall time drops tremendously after the optimizations are in place. When all optimizations are enabled ("Full Opt."), the overall time to generate all programs for all templates is around 20 minutes, which is a 99.50% reduction over the time it takes to generate all programs without any optimization.

Breaking down the effectiveness of our optimizations even further, we find that the hot filling optimization is in general the most effective, with hot filling reducing the generation time by 99.34% versus 46.34% for early stop and 20.79% for eager pruning. Furthermore, we also see that early stop and eager pruning have cases where they result in taking more time to generate programs than without any optimization (the negative percentages in the table), which suggests the extra checks required by early stop and the time to invoke Z3 to solve constraints end up introducing more overhead than actually helping. (We did not set a timeout for Z3, because we did not observe Z3 getting stuck; however, setting a timeout could

Table 1: Time ((dd:)hh:mm:ss) and relative reduction to Non-opt. (%) to generate 1,000 programs in various configurations. Tmpl. is Templates; Rdct. is the relative reduction.

Tmpl.	Non-opt.	Early Stop		Hot Filling		Eager Pruning		Full Opt.	
	Time	Time	Rdct.	Time	Rdct.	Time	Rdct.	Time	Rdct.
B1	05:00:36	00:14	99.92	01:31	99.50	02:55:07	41.74	00:56	99.69
B2	12:51:25	00:14	99.97	04:14	99.45	12:53:50	-0.31	01:03	99.86
B3	01:19	00:12	84.33	00:33	58.57	02:01	-53.59	00:13	84.08
B4	01:43	01:41	2.37	01:41	2.03	01:41	2.45	01:43	0.29
M1	11:32	00:10	98.60	00:42	93.96	10:17	10.82	00:33	95.22
M2	12:57	00:12	98.49	00:44	94.29	14:40	-13.22	00:34	95.59
M3	11:00	02:19	78.90	01:05	90.14	13:04	-18.88	00:56	91.49
M4	19:44	04:31	77.12	00:28	97.60	20:35	-4.26	00:31	97.39
M5	05:14:14	46:01	85.36	01:04	99.66	04:20:07	17.22	00:46	99.76
M6	03:05	00:09	95.08	00:45	75.79	04:21	-41.40	00:38	79.43
M7	05:24	05:31	-2.40	00:38	88.19	05:41	-5.39	00:49	84.74
M8	19:19	12:26	35.62	01:08	94.14	21:10	-9.62	01:16	93.43
M9	02:25	00:14	90.42	00:28	80.59	02:49	-16.18	00:20	85.98
M10	09:07	02:55	67.96	00:41	92.50	10:32	-15.63	00:39	92.90
M11	10:58	00:10	98.48	02:15	79.44	08:35	21.79	01:02	90.62
M12	04:23	04:40	-6.38	00:36	86.51	05:24	-22.88	00:43	83.64
M13	11:35:19	11:26:37	1.25	01:06	99.84	00:57	99.86	00:51	99.88
M14	11:40:46	05:43:54	50.93	01:38	99.77	11:58:25	-2.52	01:17	99.82
M15	03:55:35	00:09	99.94	00:45	99.68	02:49:16	28.15	00:14	99.90
M16	07:38:42	07:47:57	-2.02	01:02	99.77	07:57:36	-4.12	01:09	99.75
M17	10:38:24	11:28:05	-7.78	03:12	99.50	10:51:56	-2.12	03:32	99.45
M18	04:57	00:09	96.81	00:41	86.07	05:43	-15.45	00:32	89.08
M19	05:47	05:53	-1.82	01:01	82.34	03:51	33.41	01:02	82.16
Σ	2:22:38:42	1:13:54:24	46.34	27:59	99.34	2:07:57:38	20.79	21:20	99.50

impact the performance of the eager pruning optimization.) We see just one case for hot filling (and ultimately for when all optimizations are on) where there is little reduction in time. However, this one case (B4) takes very little time even without any optimizations, and the difference in time is seemingly just noise. Ultimately, all optimizations do help overall, with the reduction in time when using all optimizations still higher than each individually.

We also measure the time to execute the generated programs from each of the 23 manually created templates. The total time across all generated programs is around two hours on L4 and around two and a half hours on L1.

6.2 Template Extraction

We extract 5,419 templates from 16,309 methods in 15,325 classes, resulting in 50,609 generated programs. Recall that we let JAttack generate 10 programs from every template (Section 5), but not every template includes sufficient number of holes from which 10 programs can be generated (JAttack only explores the reachable holes), which is why the total number of generated programs is less than $10 * 5{,}419 = 54{,}190$. We found 137 out of 50,609 generated programs failed during our JIT-testing procedure. We inspected all these 137 programs and discovered four unique bugs (Section 6.3).

Table 2: Comparing results of JAttack and Java* Fuzzer.

	#Generated	#Timeout	#Failures	Coverage(%)	
				C1	C2
JAttack	50609	1243	137	84.3	80.3
Java* Fuzzer	15931	2336	0	80.6	67.5

In addition, we compare JAttack against an existing automated compiler testing tool, Java* Fuzzer [27], which is a fuzzer tool Oracle has been using daily for years and has been successful at detecting bugs in the Oracle JDK (JIT) compiler. Guided by grammar rules and pre-defined heuristics on program structures, Java* Fuzzer generates hundreds of thousands of small, random Java programs as tests, and it then performs differential testing between a JVM under test and a reference JVM. In contrast, JAttack is primarily developed for developers to embed their knowledge into program generation by specifying holes in templates with automated template extraction from existing Java programs. Although JAttack and Java* Fuzzer have similar intentions, they work quite differently, which is why the comparison results should be taken with a grain of salt. We run Java* Fuzzer using the same resources (CPU/RAM) for the same amount of time (which matches the total execution time for JAttack in Section 6.2). We perform differential testing by comparing outputs from executions across different JIT tiers, same as for JAttack. We also collect code coverage of both the C1 (`src/hotspot/share/c1/`) and C2 (`src/hotspot/share/opto/`) compilers from executing the programs generated by both tools separately. Table 2 compares the results of JAttack and Java* Fuzzer. Java* Fuzzer did not generate any program that would expose a bug in the Oracle JDK JIT compiler in this time frame. The code coverage achieved using both tools are close to each other, though JAttack achieves slightly higher code coverage on both C1 and C2.

6.3 Detected JIT-Bugs

Bug m12gen61[1], from template M12, showed mismatching outputs on different tiers because C2's range-check elimination leads to incorrect loop executions. The Oracle JDK developers labeled the bug we reported as a CVE (Common Vulnerabilities and Exposures)[2], and they fixed the bug in a recent Oracle Critical Patch Update[3]. The JDK developers also confirmed Bug m4gen152[4], where a crash occurred from C2, as a P3[5] bug; this bug was discovered in parallel by others and was fixed in JDK 16. Our template that exposed this bug is shown in Figure 1a.

Additionally, we discovered four bugs using extracted templates from existing Java projects. Bug math182[6] crashed on tiers L1 and L4 because an array store in C1 compiled code writes to an arbitrary location due to index overflow. The JDK developers labeled the bug

[1]https://bugs.openjdk.java.net/browse/JDK-8239244 (Login required)
[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14792
[3]https://www.oracle.com/security-alerts/cpuoct2020.html
[4]https://bugs.openjdk.java.net/browse/JDK-8258981
[5]P3: Major loss of function.
[6]https://bugs.openjdk.java.net/browse/JDK-8271130 (Login required)

as a CVE[7], and they fixed the bug in another recent Oracle Critical Patch Update[8]. Bug checkstyle106[9] was confirmed as a crash bug, with priority P2[10], related to wrong JVM state used for a receiver null check, and it was fixed in JDK 17. Bug codec292[11] missed throwing some `NegativeArraySizeException` on tier L4 caused by C2 optimizations; the JDK developers labeled it as a P2 bug and fixed it in JDK 18. Bug compress208[12] crashed due to incorrect C2 loop optimizations before calling `Arrays.copyOf` with a negative parameter; this bug was confirmed with priority P3 and was also discovered in parallel by others. The bug was fixed in JDK 18.

7 DISCUSSION

In this section, we contrast JAttack's execution-based generation to static generation, describe limitations of JAttack, and provide directions for future work.

7.1 Execution-Based vs. Static Generation

Recall JAttack generates programs through an execution-based model (Section 3.3) but we could have generated programs statically by processing an entire template and replacing all holes with concrete expressions. Static generation would process the template repeatedly, putting in different concrete expressions per hole to output a new generated program, up to some maximum number. Generating programs statically could be faster, because it would not be executing the program at any point.

However, the execution-based generation provides a number of advantages over static generation. Execution-based generation (1) knows what exactly would be executed in a generated program after being compiled, i.e., which parts are dead code or which parts are executable (such information can be leveraged to guide the exploration of holes instead of relying on randomness, which we leave as future work) and (2) makes it possible to use values available at runtime to construct holes; consider:

```
int m(int[] a) {
  return a[intVal(0, a.length).eval()];
}
```

where the hole would be a random integer between `0` and the length of the array `a`, which depends on the value of `a`, known only when actually executing the template. An execution-based model allows for expressing more complex programs that static generation cannot generate, as it does not have such runtime information.

To compare execution-based and static generation, we create a variant of JAttack that generates programs statically. This variant relies on the same syntax and semantics, but it statically processes the template once to replace all the holes with concrete expressions. Similarly to execution-based generation, we construct eASTs for all the holes. Each eAST per hole contains all the choices for the hole, i.e., concrete expressions that can be filled in the hole. For each hole written in the template, we randomly choose one of the concrete expressions to replace the hole, resulting in a generated program. The generated program has every hole filled, unlike for execution-based generation where some holes may remain unfilled if they are not reached during execution.

For this evaluation, we use the same configuration (1,000 programs for each template) for our static variant of JAttack as to allow for proper comparison against the execution-based model. The total generation time with static generation is around three minutes, which is shorter than execution-based generation (around 20 minutes), but they both generate a large number of programs, and executing all generated programs for both approaches still takes almost four and a half hours. As such, generation time is practically negligible compared against the differential testing part of JAttack. Furthermore, since static generation fills every hole in the template, some generated programs could be syntactically different from each other, but their differences are only for expressions in the unreachable holes, so essentially the same code would be executed. Execution-based generation would skip unreachable holes, ensuring every generated program is not only syntactically different but also executed differently. We collected reachability of the filled holes when executing the generated programs. 78.44% of filled holes are reached during execution of generated programs from static generation while execution-based generation guarantees 100.00% reachability. In terms of detected bugs, compared against execution-based generation, statically generated programs detected only Bug m12gen61 and missed detecting Bug m4gen152.

7.2 Limitations & Future Work

There are two main reasons why a relatively small number (5,419) of templates are extracted from a relatively large number (15,325) of classes in existing projects. First, JAttack currently supports only static Java methods as template entry methods. We leave support of instance methods as template entry methods for future work, e.g., using EvoSuite [33] to create receiver objects and inputs for instance methods. Second, we use a different name for the extracted template class from the original class, which sometimes made the template not pass Java type-checking due to circular dependencies between the template class and other classes. We will explore editing the original class in place instead of creating a renamed template class as future work.

JAttack requires re-executing programs many times just to trigger JIT optimizations for testing. We considered other options such as `-XX:CompileThreshold` that controls the number of interpreted method invocations before optimization. We also considered the option `-XX:Tier4InvocationThreshold` that controls the minimum number of method invocations before transitioning to L4. However, we found these other options also have a big effect on when JIT optimizations occur, so just using these options would not truly reflect actual JIT usage, similar to just enabling C2 from the beginning [40].

Not all bugs we detected are reproducible each time due to the non-deterministic nature of executions and JIT profiling (which is different from non-determinism within programs under execution, discussed in Section 3.5). For example, in one of the generated programs for M4, we could not always observe failure (crashing the JVM) when run on the same JIT compiler multiple times. We plan to investigate such flakiness in the future.

Although we designed and implemented JAttack for Java and JIT compilers, the simple imperative language and extensions, as

[7]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21305
[8]https://www.oracle.com/security-alerts/cpujan2022.html
[9]https://bugs.openjdk.java.net/browse/JDK-8271276
[10]P2: Crashes, loss of data, severe memory leak.
[11]https://bugs.openjdk.java.net/browse/JDK-8271459
[12]https://bugs.openjdk.java.net/browse/JDK-8271926

shown in Section 3.1, represent the foundations for supporting templates for general imperative languages. Thus the ideas of template-based testing and execution-based generation, can be also applied to other languages and compilers, e.g., Scala, C#, etc., or even software systems in general. We leave this as future work.

8 RELATED WORK

Compiler testing. There is a large body of work on compiler testing, systematically reviewed in recent surveys [19, 88]. For example, Csmith [80, 94] is a well-known tool for testing C compilers by randomly generating C programs. It found bugs in mainstream compilers [95, 96] and led to significant attention for compiler testing [2, 20, 42, 64, 67]. Mutation-based fuzzing [62, 65, 75–77, 92, 99] is another approach to testing compilers by mutating existing programs, with several techniques and tools specifically for Java [11, 17, 22, 23, 51, 58, 73, 74, 91, 101]. Concerning JIT compilers, Yoshikawa et al. [98] presented a generation approach that produces random Java bytecode . Java* Fuzzer [27, 47, 49] generates random Java programs to test Java JIT compilers. There is also work on testing the C# JIT compilers [70, 71] and Smalltalk JIT compilers [78]. Unlike all these techniques, JAttack was primarily developed to complement manually-written tests. Developers can embed their knowledge into program generation by specifying holes for exploration, enabling better testing of JIT compilers that require complex structures and execution to reveal bugs. We do compare against Java* Fuzzer as part of our evaluation.

While JAttack relies on differential testing [66] to determine whether a test fails, other means to construct a test oracle include metamorphic testing [21] and specification-based testing [81, 97]. Equivalence Modulo Inputs (EMI) [25, 31, 59–61, 63, 69, 87] is a representative of metamorphic testing technique. EMI produces equivalent but different test programs and compares behaviors across these programs on a single compiler. Chen et al. [18] compared differential testing and EMI techniques.

Template-based program synthesis. There has been work on synthesizing programs given initial templates. These techniques can either synthesize programs using SAT/SMT solvers [32, 39, 52, 53, 83, 85, 86], using combinatorial techniques focusing on just variables [100], or define holes using domain-specific languages [15, 84]. Ching and Katz [24] proposed to generate tests for APL-to-C compiler COMPC through a template that denotes functions and data types, which leverages the dynamic nature of APL to execute programs as soon as holes get filled. CodeHint [34] synthesizes sequences of API method invocations by running code with holes to be filled. EdSketch [44] synthesizes implementations for holes through test executions, aiming to pass the tests it runs on, focusing on exploring field references. In contrast to all of these, JAttack generates concrete programs for testing a JIT compiler by executing templates and allows richer expressions to be generated in holes.

Test input generation. Randoop [72] and EvoSuite [33] automatically generate JUnit tests by incrementally extending sequences of method invocations. Neither is able to effectively explore inside a method, e.g., at the expression level, as JAttack. ASTGen [28] and UDITA [35] bounded-exhaustively generate complex test inputs, including programs, but both require developers to manually write extra predicates or generators to encode their intuition of guiding exploration. Similarly, QuickCheck [26, 43], as a property-based testing tool, is also capable of randomly generating test inputs but requires developers to provide generators for complex data types. In contrast to all of these, JAttack provides a different way for developers to write templates. Developers simply write programs with holes to make templates or even automatically extract templates from existing code. Templates are written in the host language the developers are already using.

Other research on JIT and JVM. There has been work on formal verification of JIT compilers [8, 14, 38, 41, 68, 82, 90, 93], JIT-induced side-channel detection [12, 13], and on identifying hard-to-optimize code [36] and unspecified JNI behaviors of a JVM [46].

9 CONCLUSION

We presented JAttack, a framework that enables template-based testing for compilers. Using JAttack, compiler developers can write templates in the same language as the compiler they are testing (Java), enabling them to leverage their domain knowledge to set up a code structure likely to lead to compiler optimizations while leaving holes representing expressions they want explored. JAttack executes templates, exploring possible expressions for holes and filling them in, generating programs to later be run on compilers. To speed up the generation process, we introduced three optimizations that reduced overall generation time by 99.50% in our experiments. Using 23 templates created on our own and 5,419 templates extracted from existing Java projects, JAttack found six critical (P3 or higher) bugs in Oracle JDK, all of which were confirmed and fixed by Oracle developers. Four of them were previously unknown, including two unknown CVEs. JAttack blends the power of developers insights, who are providing templates, and random testing to detect critical bugs.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Kush Jain, Sandeep Konchady, Owolabi Legunsen, Yu Liu, Pengyu Nie, Aditya Thimmaiah, Jiyang Zhang, and the anonymous reviewers for their comments and feedback. This work is partially supported by a Google Faculty Research Award, Army Futures Command, and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, and CCF-2217696.

REFERENCES

[1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. 2007. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[2] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating Focused Random Tests Using Directed Swarm Testing. In *International Symposium on Software Testing and Analysis.* ACM, 70–81.

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

[4] Oracle Corporation and/or its affiliates. 2021. *[JDK-8251535] Partial peeling at unsigned test adds incorrect loop exit check - Java Bug System.* https://bugs.openjdk.java.net/browse/JDK-8251535.

[5] Oracle Corporation and/or its affiliates. 2021. *Regression Test Harness for the JDK: jtreg.* https://openjdk.java.net/jtreg.

[6] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[7] John Aycock. 2003. A Brief History of Just-in-Time. *Comput. Surveys* 35, 2 (2003), 97–113.

[8] Auréle Barriére, Sandrine Blazy, and David Pichardie. 2020. Towards Formally Verified Just-in-Time compilation. In *International Workshop on Coq for Programming Languages.*

[9] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *International Conference on Software Engineering*. ACM, 550–561.

[10] Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In *Proceedings of the 2018 European Conference on Object-Oriented Programming*. Dagstuhl, 17:1–17:31.

[11] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *The Symposium on Network and Distributed System Security*. The Internet Society.

[12] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *Symposium on Security and Privacy*. IEEE, 1207–1222.

[13] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *International Conference on Software Engineering*. ACM, 1011–1023.

[14] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a Verified Range Analysis for JavaScript JITs. In *Programming Language Design and Implementation*. ACM, 135–150.

[15] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *International Conference on Very Large Data Bases*. VLDB Endowment, 1097–1107.

[16] Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. In *Symposium on Dynamic Languages*. ACM, 1–14.

[17] 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.

[18] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *International Conference on Software Engineering*. ACM, 180–190.

[19] 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.

[20] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *International Conference on Software Engineering*. IEEE, 305–316.

[21] TY Chen, SC Cheung, and SM Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report.

[22] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *International Conference on Software Engineering*. IEEE, 1257–1268.

[23] 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.

[24] Wai-Mee Ching and Alex Katz. 1993. The Testing of an APL Compiler. In *International Conference on APL*. ACM, 55–62.

[25] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink. In *International Conference on Software Engineering*. ACM, 335–346.

[26] 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.

[27] Intel Corporation. 2016. *android-art-intel/Fuzzer: Java* Fuzzer for Android**. https://github.com/android-art-intel/Fuzzer.

[28] 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.

[29] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[30] 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.

[31] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing. In *Programming Language Design and Implementation*. ACM, 1017–1032.

[32] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Programming Language Design and Implementation*. ACM, 229–239.

[33] 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* (Szeged, Hungary). ACM, 416–419.

[34] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *International Conference on Software Engineering*. ACM, 653–663.

[35] 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.

[36] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-Unfriendly JavaScript Code. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 357–368.

[37] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *International Symposium on Software Testing and Analysis*. ACM, 78–88.

[38] Shu-yu Guo and Jens Palsberg. 2011. The Essence of Compiling with Traces. In *Symposium on Principles of Programming Languages*. ACM, 563–574.

[39] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. In *Computer Aided Verification*. Springer, 418–423.

[40] Andrew Haley. 2015. *How to change compilation policy to trigger C2 compilation ASAP?* https://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-May/018010.html.

[41] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-Version Compiler Validation. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 191–201.

[42] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. USENIX, 38.

[43] Paul Holser. 2020. *junit-quickcheck – junit-quickcheck: Property-based testing, JUnit-style*. https://pholser.github.io/junit-quickcheck/index.html.

[44] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-Driven Sketching for Java. In *International SPIN Symposium on Model Checking of Software*. ACM, 162–171.

[45] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *International Conference on Generative Programming: Concepts and Experiences*. ACM, 123–132.

[46] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *International Conference on Software Engineering*. IEEE, 1708–1718.

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

[48] Free Software Foundation, Inc. 2021. *Testsuites (GNU Compiler Collection (GCC) Internals)*. https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html.

[49] Red Hat, Inc. 2018. *shipilev/JavaFuzzer: Java* Fuzzer for JVM*. https://github.com/shipilev/JavaFuzzer.

[50] Radu Iosif. 2002. Symmetry Reduction Criteria for Software Model Checking. In *International SPIN Symposium on Model Checking of Software*. Springer, 22–41.

[51] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java. In *NASA Formal Methods Symposium*.

[52] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 934–937.

[53] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *International Conference on Software Engineering*. ACM, 215–224.

[54] 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.

[55] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[56] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

[57] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *International Symposium on the Foundations of Software Engineering*. 654–665.

[58] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Conference on Computer and Communications Security*. ACM, 2511–2513.

[59] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Programming Language Design and Implementation*. ACM, 216–226.

[60] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 386–399.

[61] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *International Symposium on Software Testing and Analysis*. ACM, 327–337.

[62] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *USENIX Security Symposium*. USENIX, 3559–3576.

[63] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Programming Language Design and Implementation*. ACM, 65–76.

[64] 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.

[65] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331.

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

[67] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Programming Language Design and Implementation*. ACM, 187–196.

[68] Magnus O. Myreen. 2010. Verified Just-in-Time Compiler on X86. In *Symposium on Principles of Programming Languages*. ACM, 107–118.

[69] 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.

[70] Jakob Botsch Nielsen. 2018. *Fuzzing the .NET JIT Compiler*. https://mattwarren.org/2018/08/28/Fuzzing-the-.NET-JIT-Compiler/.

[71] Jakob Botsch Nielsen. 2020. *jakobbotsch/Fuzzlyn: Fuzzer for the .NET toolchains, developed as a project for the 2018 Language-Based Security course at Aarhus University*. https://github.com/jakobbotsch/Fuzzlyn.

[72] 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.

[73] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *International Symposium on Software Testing and Analysis*. ACM, 398–401.

[74] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *International Symposium on Software Testing and Analysis*. ACM, 329–340.

[75] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 174:1–174:29.

[76] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Symposium on Security and Privacy*. IEEE, 1629–1642.

[77] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *Symposium on Security and Privacy*. IEEE, 697–710.

[78] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-guided Differential JIT Compiler Unit Testing. In *Programming Language Design and Implementation*. ACM, 981–992.

[79] LLVM Project. 2021. *LLVM Testing Infrastructure Guide*. https://llvm.org/docs/TestingGuide.html.

[80] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Programming Language Design and Implementation*. ACM, 335–346.

[81] Richard Schumi and Jun Sun. 2021. SpecTest: Specification-Based Compiler Testing. In *Fundamental Approaches to Software Engineering*. Springer, 269–291.

[82] Boris Shingarov. 2019. Formal Verification of JIT by Symbolic Execution. In *International Workshop on Virtual Machines and Intermediate Languages*.

[83] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 289–299.

[84] Emin Gün Sirer and Brian N. Bershad. 2000. Using Production Grammars in Software Testing. In *Conference on Domain-Specific Languages*. ACM, 1–13.

[85] Armando Solar-Lezama. 2013. Program Sketching. *International Journal on Software Tools for Technology Transfer* 15, 5–6 (2013), 475–495.

[86] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 404–415.

[87] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 849–863.

[88] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. 2020. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science* 14, 1 (2020), 1:20.

[89] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized Unit Tests. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 253–262.

[90] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *Computer Aided Verification*. Springer, 564–586.

[91] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing A Test Corpus with Bonsai Fuzzing. In *International Conference on Software Engineering*. ACM, 723–735.

[92] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Symposium on Security and Privacy*. IEEE, 497–512.

[93] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *Symposium on Operating Systems Design and Implementation*. USENIX, 33–47.

[94] 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.

[95] 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.

[96] 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.

[97] 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 *Programming Language Design and Implementation*. ACM, 435–450.

[98] 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.

[99] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *Symposium on Security and Privacy*. IEEE, 769–786.

[100] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Programming Language Design and Implementation*. ACM, 347–361.

[101] 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.