The Report committee for Nathaniel Brett Wiatrek

Certifies that this is the approved version of the following report:

Automatically Extracting Templates for Testing Java JIT Compilers

SUPERVISING COMMITTEE:

Milos Gligoric, Supervisor

August Shi

Automatically Extracting Templates for Testing Java JIT Compilers

by

Nathaniel Brett Wiatrek

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2021

Acknowledgments

Thank you to my family for all the support given in pursuit of this degree. Thank you to all of the friends I've made along this journey, and a special thanks to my fiancée. Without her, I would never have felt confident enough to have applied for this masters program. Love you, Em.

Automatically Extracting Templates for Testing Java JIT Compilers

Nathaniel Brett Wiatrek, M.S.E.
The University of Texas at Austin, 2021

Supervisor: Milos Gligoric

The Java programming language is widely used in industry and academia. Since the language is object-oriented and platform independent, it is a perfect solution to deliver a variety of programs. To increase performance, Java optimizes running software with the use of just-in-time (JIT) compilation, in which the JIT compiler will generate new code that is more streamlined. If there is a bug in this newly generated code there could be significant consequences. To validate JIT correctness JITAttack was developed, which leverages program templates to test compilers (such as Java JIT). These templates are currently manually-written and time-intensive to create.

We present a tool, JITTEMPLATER, that is designed to take in a Java program and create templates to be used with JITAttack's JIT compiler testing. JITTEMPLATER parses the Abstract Syntax Tree of the given Java file to convert statements and expressions to their equivalent in JITAttack's API. JITTEMPLATER is part of a larger process that takes in a real-world Java project to convert into a project of templates for testing purposes. The automation of template extraction leads to novel and interesting inputs to be used with JITAttack. To date, we have found three bugs utilizing real-world Java programs that were extracted by JITTEMPLATER.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The just-in-time (JIT) compiler is a product of software engineering that is designed to increase the efficiency of long running software programs [1, 4, 11]. The JIT compilers require that the cost of the compilation be offset by the speedup gained from the optimization [21]. One of the results of these optimizations is that the JIT compiler will generate new code from the running program [21]. The generated code may be incorrect, which indicates that there is a bug in the JIT compiler. To check the correctness of the JIT compiler, JITAttack was developed. JITAttack is a template-based testing framework that takes in a template and generates concrete programs from that template that get executed many times to find inconsistencies at different JIT levels. With JITAttack a developer can write a template, which is a Java program file with holes, that JITAttack will then use to generate programs by filling in the holes with concrete values [26].

We present a tool, JITTEMPLATER, that streamlines the template creation process by taking in real-world Java programs and converting them into templates to be used alongside JITAttack.

Motivation. Manually creating complex and novel templates takes a significant amount of time. With JITAttack, a developer is given a large and robust application program interface (API) to be able to create a complex template. This large API allows for the

creation of very expressive templates, but also adds to the amount of underlying knowledge the developer must gain before using the framework to maximum effect.

Automating the creation of templates from real-world Java programs significantly reduces the time to create many complex and novel templates compared against manually writing them. By using automation to generate the templates, we also ensure the use of as many JITAttack APIs as are applicable for the new templates.

With the use of JITTEMPLATER, we can generate thousands of new templates for JITAttack to consume to find bugs in the JIT compilers. The JITTEMPLATER process was designed to take in a Java project, find all the opportunities for templates, and then templatize the Java code for use with JITAttack. The end-to-end process of JITTEMPLATER and JITAttack that we created can be seen in Figure 1.1.

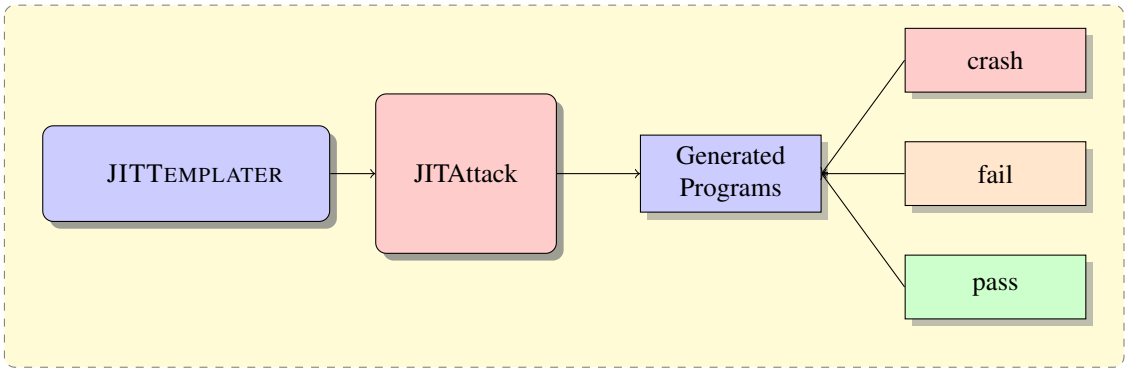

Figure 1.1: End-to-End Process.

Chapter 2

Background

This Section introduces important concepts related to JITAttack, and its corresponding API that are utilized for creating templates.

JITAttack requires a single entry method that will be used to trigger JIT compilation. There is only one requirement for determining if a method can be used as an entry method, which is that the method must be static. Having the method be static ensures that JITAttack does not have to worry about the initialization of a custom object.

The entry method must have the annotation `@Entry` to signify that it is indeed the entry method. If there are parameters to the entry method, we must create additional methods that correspond to the return type of the parameter. These additional methods must all have the `@Argument` annotation which corresponds to the position of the parameter in the method. This position starts at 1 (0 is reserved for `this` which may be used in the future) and increasing from there.

The entry method specifications result in the method declaration in Figure 2.1a line 11, being converted into Figure 2.1b line 12. The two added methods `nonPrim1()` and `nonPrim2()` are the methods that define the parameters for the entry method, in this case the `double[][] real` and `double[][] imag`.

At the core of the JITAttack API is the concept of a *hole*. A hole is simply a

```
1  public class ComplexUtils {
2      public static Complex[] split2Complex(double[] real, double[] imag) {
3          final int length = real.length;
4          final Complex[] c = new Complex[length];
5          for (int n = 0; n < length; n++) {
6              c[n] = new Complex(real[n], imag[n]);
7          }
8          return c;
9      }
10
11     public static Complex[][] split2Complex(double[][] real, double[][] imag) {
12         final int length = real.length;
13         Complex[][] c = new Complex[length][];
14         for (int x = 0; x < length; x++) {
15             c[x] = split2Complex(real[x], imag[x]);
16         }
17         return c;
18     }
19 }
```

(a) A Class from the Apache Math Project [3] Before Templatization.

```
1  public class ComplexUtilsTemplate {
2      public static Complex[] split2ComplexTemplate(double[] real, double[] imag) {
3          final int length = intVal().eval();
4          final Complex[] c = new Complex[length];
5          for (int n = 0; n < intId().eval(); n++) {
6              c[intVal(0, c.length).eval()] = new Complex(real[n], imag[n]);
7          }
8          return c;
9      }
10
11     @jitattack.Entry()
12     public static Complex[][] split2ComplexTemplate(double[][] real, double[][] imag) {
13         final int length = intVal().eval();
14         Complex[][] c = new Complex[length][];
15         for (int x = 0; x < intId().eval(); x++) {
16             c[intVal(0, c.length).eval()] = split2ComplexTemplate(real[x], imag[x]);
17         }
18         return c;
19     }
20
21     @Argument(1)
22     public static double[][] nonPrim1() {
23         return new double[][] { { intVal().eval(), intVal().eval() }, { intVal().eval(), intVal()
               .eval() }, { intVal().eval(), intVal().eval() } };
24     }
25
26     @Argument(2)
27     public static double[][] nonPrim2() {
28         return new double[][] { { intVal().eval(), intVal().eval() }, { intVal().eval(), intVal()
               .eval() }, { intVal().eval(), intVal().eval() }, { intVal().eval(), intVal().eval() }};
29     }
30 }
```

(b) A Class from the Apache Math Project [3] After Templatization.

Figure 2.1: A Class from the Apache Math Project [3] Before and After Templatization.

bit of code that is going to be filled in by JITAttack. A program that has holes is called a *template*. There are several different types of method calls in the API that can create holes, and they can cover a plethora of situations. The most simple of calls are those that describe literal expressions; these method calls are `intVal()` and `boolVal()`. With these API calls, JITAttack will create a hole to be filled by a corresponding random value described by their search space [26].

More complicated holes are those that involve binary expressions. Binary expression holes can be created by three types of API calls, 1) `relation`, 2) `arithmetic`, and 3) `logic`. These calls are used whenever there is an example of the corresponding binary operator that has smaller API calls on either side. For example `if(relation(intId(), intVal).eval())` could result in a concrete expression such as `if(a > 7)`. The list of complex holes are `relation`, `arithmetic`, `logic`, `loop limits`, and `arrays`.

One thing to note is the use of `.eval()`. The `.eval()` corresponds to a hole, which constructs a valid Java expression and builds an AST. When the `.eval()` is executed, JITAttack converts the entire expression into a concrete Java expression. As such, `.eval()` is only used on the outermost hole.

Chapter 3

Technique

This Section describes the end-to-end automation process for use with JITAttack, as well as how the JITTEMPLATER tool is implemented.

The end-to-end process of automating the templates to be used in JITAttack boils down into the three steps that are shown in Figure 3.1.

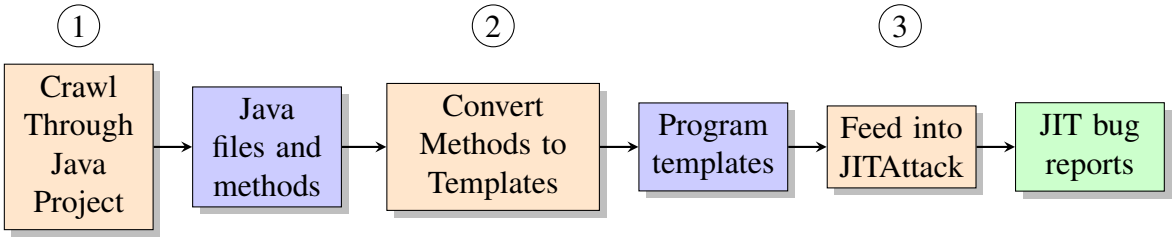

Figure 3.1: JITTEMPLATER Steps Overview.

The three stages are to first crawl and parse through a real-world Java project. This stage generates Java files and methods that will be used in the second stage. Second, JITTEMPLATER will convert an input file into an output template. In the third stage, the template will be utilized in JITAttack as a program to execute. The result of the third stage is a JIT bug report.

3.1 Crawl Through Java Project and Determine Inputs

The conversion of a real-world Java project to a project of templates for JITAttack begins with finding all of the methods in the project that can be used as an entry point to JITAttack.

To find all of the possibilities for templates we created a Java console program that searches over the entirety of a Java project. This program does a Depth First Search (DFS) of all the files and folders to find all of the static methods in the project. We ran this program on thirteen different Java projects, the list of these projects is in Table 3.1.

Project	**SHA**	**LOC**	**#Files**
Checkstyle	bdaac14	111242	408
Codec	4de60e8	22825	73
Compress	488425c	53899	221
Configuration	e63eeae	64936	261
Gson	f319c1b	14731	73
JFreeChart	b5affea	221155	637
JXPath	192f4c9	30643	171
Lang	261a579	88047	217
libGDX	a10a776	134042	577
Math	dff1a09	194050	851
Openfire	565da30	183971	731
Vectorz	d9b79ec	61631	295
ZXing	d2027d0	38429	231
TOTAL	N/A	1219601	4746

Table 3.1: Java Projects Used To Obtain Templates.

Table 3.1 contains the Java project that was used, the SHA of the project that we used in our experiments, the Lines of Code (LOC) in the project, and the total number of source files in the project. There is a wide range in number of files and LOC in the

thirteen projects, with LOC ranging from just under 15,000 all the way to over 200,000 and number of files going from 73 to 851. The method types for each of the Java projects, including a breakdown of static and instance methods, can be found in Table 3.2.

Project	**#Static M.**		**#Instance M.**		
	Total	**w/ Generics**	**Total**	**w/ Generics**	**w/ D. Ctor.**
Checkstyle	808	3	2531	1	1935
Codec	352	0	310	0	243
Compress	301	5	1678	1	577
Configuration	287	29	2029	101	1342
Gson	85	10	402	23	243
JFreeChart	463	0	7227	0	4861
JXPath	119	0	1364	0	231
Lang	1719	265	1270	15	918
libGDX	861	38	8054	89	5634
Math	878	49	4977	0	2123
Openfire	1014	22	6795	11	4679
Vectorz	746	7	5117	0	2047
ZXing	407	0	926	0	297
TOTAL	8040	428	42680	241	25130

Table 3.2: Java Project Method Types.

Table 3.2 breaks down the specifics of the types of methods that can be found in each of the Java projects. The `w/Generics` specifies that the method had generic type parameters. The specification of instance methods that only have default constructors (`w/D. Ctor.`) is an important data point for determining the next direction for JITAttack. Since we see that there are many methods that do not require a special instantiation, there may be an update in the future that allows these methods to be used by JITAttack as entry methods.

With the requirement of static methods that do not have generics, we are left with roughly 20% of the total methods in the Java projects to use as an input for extracting program templates. Creating this many inputs to JITAttack would be excruciatingly time-intensive if done manually. The output of the first phase of the process is a text file that is used by JITTEMPLATER, in the second phase.

3.2 Convert Methods to Templates

The second stage of the end-to-end process for testing with JITAttack is the conversion of a Java file into a template. The goal of the conversion process is to create a template that preserves the structure and expressiveness of the original, while inserting applicable holes for JITAttack. The product of this goal results in the template having the same semantic structure as the original input program, with several holes to be filled by JITAttack.

JITTEMPLATER, the program that does the conversion of the file, takes in a number of arguments that allow for the customization of a template. The two arguments that are required to perform the conversion are the input file (i.e., the source Java file) and the signature of the method that is going to be used as the entry point for JITAttack. Both of these inputs are given directly from the output of the console program from the first stage.

Other notable arguments for JITTEMPLATER are the maximum number of holes to insert into a given template and the boolean flag for making other holes. The `--max-num-holes` argument defines the maximum number of holes JITTEMPLATER should insert. The `--make-other-holes` argument determines whether or not JITTEMPLATER is allowed to insert holes into methods in the source file that are not the entry

method. The default values for `--max-num-of-holes` and `--make-other-holes` are `Integer.MAX_VALUE` and `true`, respectively.

JITTEMPLATER works by leveraging the Abstract Sytntax Tree (AST) that is formed by a Java file. With this structure at the core of our implementation we can traverse the tree to find leaves and branches that are useful for turning into holes with the JITAttack API.

First, JITTEMPLATER finds the entry method. Determining the entry method is trivial due to the two required inputs of the program. Once the program has been parsed into the AST data structure, a pre-order traversal is done on the tree to look for the method that matches the inputted signature. If the method is not found, or the method is found but is not static, JITTEMPLATER throws an exception to inform the user of the possible user error on input.

To generate the parameter methods for the entry method, the general idea is to convert whatever the input parameter is to be the default of that type. For example, if the input parameter is a primitive type (e.g., `char` or `boolean`) JITTEMPLATER will give it the default value (e.g., `u0000` or `false` respectively). When the input parameter is not a primitive type (e.g., a class), JITTEMPLATER will first search the Java project for the class definition. If JITTEMPLATER finds the class definition, then it determines whether there is a default constructor or a constructor that has only primitive type arguments. If the class's constructor matches either of those requirements we will make a new instance of that object. If all of the above failed we will use `null` as the input to the method.

With array parameters, a new array will be formed in one of two ways. If the array

is an integer array we will utilize the JITAttack API of `intArrVal()` to create it. In all other situations, a new array will be created with a randomly generated size, but will always be a square array in the event of multi-dimensional arrays. The creation of these arrays follows the same rules as the non array, whatever element type the array has will define the values at each index of the generated array.

All non-entry methods are found in a similar fashion to the entry method, in which JITTEMPLATER runs a pre-ordered traversal to get all Method Declarations. Each Method Declaration will be used as the start of its own AST to allow for separate computations. It is important to note that only the entry method is required to be static, any other method in the Java file can be templatized without fear of being an instance method.

To traverse the AST for each method we do an in-order traversal. This allows us to modify the leafs of all the branches before modifying the overall branch. In the tree, we continue searching down the branches until one of the following is found: 1) Variable Declaration, 2) Binary Expression, 3) Return Statement, 4) Assign Statement, 5) Array Access Expression. Each of these five cases are handled differently to create a compilable template.

With the `VariableDeclarationExpression`, we get all of the variables that are being declared and determine if they are a simple declaration or something more complex. An example of a simple declaration would be `int a = 1;` while a more complex declaration would be `int a = 1 + variable;`. If the initializer is a simple declaration, we will convert the right hand side of the variable declaration into an `intVal().eval()` if it is an int, double, or long. Similarly, if the simple declaration is a boolean type we will create a `boolVal().eval()` as the hole. Any other type

backing a Variable Declaration is not supported to be a hole in JITAttack. In the event of a more complex declaration we will prefer a more complex hole to be created. We handle this case the same way as a `BinaryExpression` or a `MethodCallExpression`. If there was no initializer then the variable is left intact and no hole is created.

With `BinaryExpression` we utilize recursion to break down a binary expression into several smaller expressions. If either the left or the right side of the binary expression is itself an instance of a binary expression we will start this process over. Once the binary expression has been broken down into two non-binary expressions, we will convert the left hand side and the right hand side into holes if possible. Instances of places for valid holes are if the expression is an `IntegerLiteralExpression`, a `NameExpression` that backs an integer, an `ArrayAccessExpression`, an `AssignStatement`, or a `MethodCallExpression`. The breakdown of handling each type of hole is described below.

For an instance of an `IntegerLiteralExpression`, we insert an `intVal` hole. This hole is normally wrapped by a more complex hole (e.g., `relation` or `arithmetic`), but can be a standalone hole if necessary.

An `ArrayAccessExpression` (e.g., `arr[index]`) is handled by turning the index of the expression into a hole. To ensure a minimal amount of `ArrayIndexOutOfBoundsException` exceptions, we set the range of the hole that is being inserted to be between 0 and the length of the array we are accessing. The component type of the array is not considered when making these holes, as all we are concerned with is randomizing the element that was selected. An example result of the `ArrayAccessExpression` is `arr[intVal(0, arr.length).eval()]`.

The `AssignStatement` can be complex, since we have to consider all of the possibilities for the statement. Examples of these statements are `a = b != c;`, or `a += method() + method(int_arg);`, or `a = -int_number;`. As the `Assign Statement` has so many possible ways of being expressed, it requires going through the following checks to determine how we handle it. First, if the right hand side of the expression is an instance of a `BinaryExpression` we send it back to the function that handles the Binary Expression so we can accurately parse the AST and insert as many holes as possible. Second, if the right hand side of the assign expression is a `NameExpression` and is an integer variable, we will insert a `intId()` hole. Third, if the right hand side is an integer literal or a boolean literal we insert the corresponding simple hole of `intVal()` or `boolVal()`. Fourth, if the value of the expression is a `UnaryExpression` we will handle this by converting the target of the unary expression and then adding back on the unary operator to the newly created hole. Fifth and finally, if the value of the `AssignStatement` is a `MethodCallExpression`, we will go through each of the arguments of the method and determine if there is a hole that can be inserted. If none of the above checks are true then we will insert no hole. The choice of inserting no hole, instead of attempting to force one, is due to ensuring that the template we create accurately represents the original Java program in terms of structure and expressiveness.

Once the sides of the binary expression have been converted into valid holes we determine if we can add a more complex hole. The list of these complex holes are those listed in Section 2. The requirements to be able to create a more complex hole is that both sides of the operator must be holes themselves. For example, if there is an input of `a == 0`, a `relation` hole will ultimately be created because the input will first be converted

into `intId() == intVal()`. If either side of the binary operator had not been a hole (e.g., `non_int_var == intVal()`), then no higher level hole could be inserted.

After converting a method into a template there exists opportunities for infinite loops in any kind of loop statement. To handle this we have implemented loop limiters. These limiters are an addition to the boolean condition in each of the types of loops (`do while`, `while`, and `for`). Figures 3.2a and 3.2b show how we insert the loop limiter.

```
1 for (int i = intVal().eval(); relation(intId(), intVal()).eval(); ++i) {
2     ds.data[intVal(0, ds.data.length).eval()] *= a;
3 }
```

(a) Before Loop Limit.

```
1 int _2071434 = 0;
2 for (int i = intVal().eval(); relation(intId(), intVal()).eval() && _2071434 < 1000; ++i) {
3     _2071434++;
4     ds.data[intVal(0, ds.data.length).eval()] *= a;
5 }
```

(b) After Loop Limit.

Figure 3.2: For Loop Before and After Loop Limit.

In 3.2a there is a high possibility of an infinite loop as the hole could easily create something like `1 < 2`. To prevent this we create a variable that is based on the hash of the loop statement. Using the hash ensures that the variable created will not clash with another variable that has already been created in the method. In the body of the loop we immediately increment the created variable by one and only allow for 1000 runs of the loop.

An example of a real-world program being turned into a templatized solution can

be seen in Figures 3.3 and 3.4. Figure 3.3 is a Java implementation of determining the greatest common denominator of two numbers (source code provided from Math [3]), and Figure 3.4 is the resulting template, a program that has multiple holes for JITAttack to fill.

```
1  private static int gcdPositive(int a, int b) {
2      if (a == 0) {
3          return b;
4      }
5      else if (b == 0) {
6          return a;
7      }
8
9      final int aTwos = Integer.numberOfTrailingZeros(a);
10     a >>= aTwos;
11     final int bTwos = Integer.numberOfTrailingZeros(b);
12     b >>= bTwos;
13     final int shift = FastMath.min(aTwos, bTwos);
14
15     while (a != b) {
16         final int delta = a  b;
17         b = Math.min(a, b);
18         a = Math.abs(delta);
19
20         a >>= Integer.numberOfTrailingZeros(a);
21     }
22
23     return a << shift;
24 }
```

Figure 3.3: Apcahe Math [3] - Greatest Common Denominator Positive Function Before Templatization.

```
1  @jitattack.Entry()
2  private static int gcdPositive(int a, int b) {
3      if (relation(intId(), intVal()).eval()) {
4          return intId().eval();
5      } else if (relation(intId(), intVal()).eval()) {
6          return intId().eval();
7      }
8      final int aTwos = Integer.numberOfTrailingZeros(intId().eval());
9      a >>= intId().eval();
10     final int bTwos = Integer.numberOfTrailingZeros(intId().eval());
11     b >>= intId().eval();
12     final int shift = FastMath.min(intId().eval(), intId().eval());
13     int _1363917262 = 0;
14     while (relation(intId(), intId()).eval() && _1363917262 < 1000) {
15         _1363917262++;
16         final int delta = arithmetic(intId(), intId()).eval();
17         b = Math.min(intId().eval(), intId().eval());
18         a = Math.abs(intId().eval());
19         a >>= Integer.numberOfTrailingZeros(intId().eval());
20     }
21     return arithmetic(intId(), intId()).eval();
22 }
```

Figure 3.4: Apache Math [3] - Greatest Common Denominator Positive Function After Templatization.

The templatized version of `gcdPositive` (Figure 3.4) example showcases several interesting holes that were inserted. Lines 3 and 5 show examples of a `relation` where we were first able to convert the binary expression to two holes and ultimately ended up with a more complex hole. Lines 8 and 11 showcase the ability to insert a hole into a method call. Line 17 shows the implementation of a Loop limiter in a while loop. Each of the return statements mirror their counterpart in Figure 3.3. In fact, every example of a hole here perfectly matches the structure and expressiveness of the original program. The

output of each templatized class is a new class that has *Template* appended to the name, and all possible instances of holes inserted.

3.3 Compile Safety

All templatized files are required to be able to compile. This ideology is a core staple of the end-to-end process that has been created. To ensure that the templatized files do compile there are a set of actions that must be done. The three primary actions are to ensure that 1) All references to the class we are templatizing get the *Template* suffix added, 2) there are no ill-placed `eval()`'s, and 3) that the JITAttack libraries are included as imports to the class.

We append *Template* to the end of the class that we are modifying to distinguish it from the original program (e.g., `ComplexUtils` to `ComplexUtilsTemplate` in Figure 2.1a and 2.1b). To update the class name we do a quick traversal of the AST for the entirety of the input Java file to find all the instances of this class type. In addition to this, we update all of the constructors to ensure that they generate the templatized class, using the same AST traversal.

To ensure the proper use of `eval()`s, we go through the updated AST one last time and validate that no complex holes have an `.eval()` attached to the hole inside of them. If there is an `.eval()` inside of a complex hole, we remove that from the method call. Finally, we import the static reference for the JITAttack API, `jitattack.*;`. All of these checks ensure that the template has a very high likelihood of compiling, thus giving a valid input to the JITAttack process.

Chapter 4

Evaluation

This Section describes the result of the JITTEMPLATER end-to-end process, which automatically extracts templates to be used with JITAttack.

Across the 13 projects all of the hole information was collected and stored. The types of holes are split out with Int Id, Int Val, Bool Val, Bool Id, Logical, Arithmetic, Relation, Array, and Loop Limits. Table 4.1 shows the results of the simple holes, Int Id through Bool Id, that JITTEMPLATER inserted.

Project	**Int Id**	**Int Val**	**Bool Val**	**Bool Id**	**TOTAL**
Checkstyle	1666	1211	1227	430	4543
Codec	5381	8607	20	13	14021
Compress	4623	5542	112	83	10359
Configuration	49	65	49	6	169
Gson	220	177	3	0	400
JFreeChart	6486	5421	337	670	12194
JXPath	1400	2464	82	0	3946
Lang	295619	268969	6563	6816	577967
libGDX	12693	14606	465	494	28258
Math	40362	95823	2082	2623	140890
Openfire	2507	6230	591	137	9465
Vectorz	30571	38318	4	0	68893
ZXing	11152	12312	724	350	14538
TOTAL	412729	459745	12259	11622	896355

Table 4.1: Statistics for Simple Holes in Each Project.

Table 4.2 shows the result of the complex holes, Logical through Loop Limits,

Project	Logic	Arithmetic	Relational	Array	Loop Limits	TOTAL
Checkstyle	80	181	243	50	1084	1638
Codec	0	1261	396	2673	633	4963
Compress	74	984	1100	1205	1289	3662
Configuration	2	4	14	3	25	48
Gson	4	49	46	9	26	134
JFreeChart	32	884	851	1072	2750	5589
JXPath	3	153	540	115	266	1077
Lang	9461	54819	132049	46571	62202	305102
libGDX	396	5094	2529	2255	1674	11948
Math	2124	9713	9272	24619	13091	58819
Openfire	36	343	792	3074	956	5201
Vectorz	12	6446	5665	16835	12175	41133
ZXing	317	2919	2851	2818	2379	11284
TOTAL	12541	82850	156348	101299	98550	421588

Table 4.2: Statistics for Complex Holes in Each Project.

JITTEMPLATER inserted.

These two tables show that some of the projects that were chosen do not have very many opportunities for holes. A common reason for projects that may have had many static methods from Table 3.2, but very few holes in Table 4.1 and 4.2, is that the methods themselves are rather small and primarily deal with custom classes. Some of the projects, such as Math and Lang, have a large number of multiple types of holes to be able to create a new template. Figure 4.1 shows the box plots of the length of the entry method used for every project.

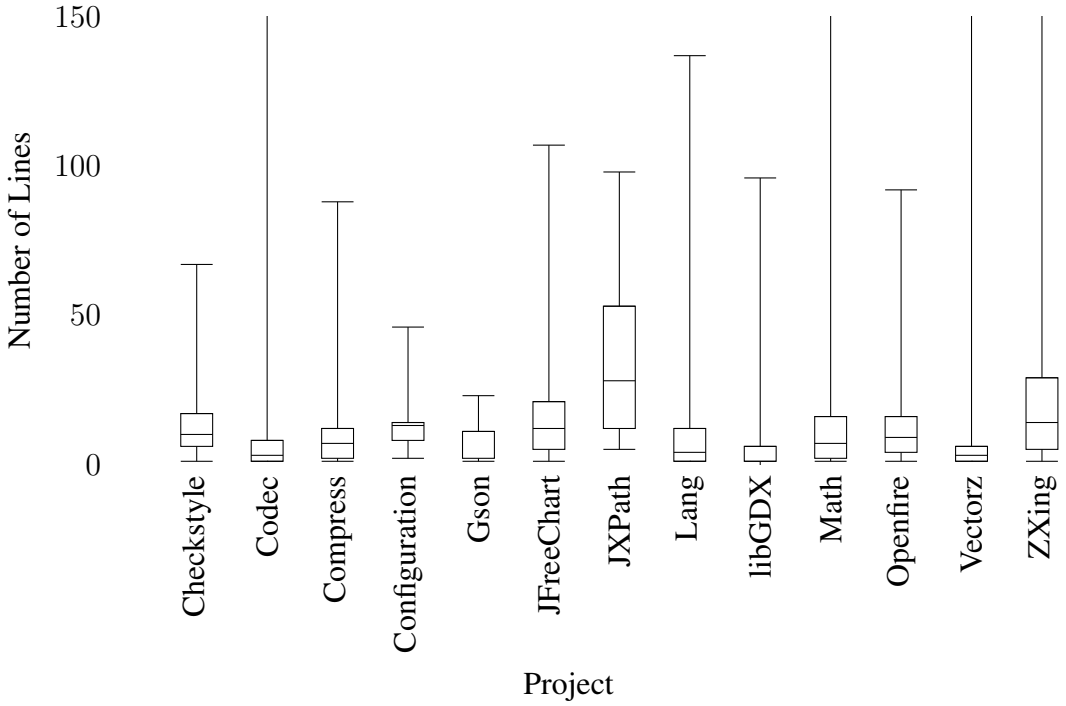

Figure 4.1: Boxplot Showing Lines of Code for Entry Methods by Project.

It is important to disclaim that, while many holes were generated, there is no guarantee that JITAttack will fill and utilize all of these holes. This is due to how JITAttack fills only holes that are reachable from the previously filled holes. The entirety of holes made are shown to describe the effectiveness of JITTEMPLATER.

It is unsurprising that the number of simple hole types are more evident than arithmetic holes as the simple holes are the building blocks for these holes. One surprising result is the very low number of boolean holes that were inserted. These holes are the fewest by far. The drawn conclusion is that there are very few instances of just a literal boolean value, but rather they are used in conjunction with relational and logical expres-

sions.

In Table 4.3 we see the breakdown in the number of templates that were able to be generated for each of the projects and the number of generated programs that were ultimately executed by JITAttack.

Project	#Methods	Mining Time	#Templates	Convert Time
Checkstyle	805	00:02	193	01:54
Codec	352	00:01	142	01:08
Compress	296	00:01	143	01:02
Configuration	258	00:01	8	00:06
Gson	75	00:01	11	00:06
JFreeChart	463	00:03	159	01:45
JXPath	119	00:01	37	00:21
Lang	1454	00:02	732	11:42
Math	829	00:03	567	05:08
Openfire	992	00:03	138	01:55
Vectorz	739	00:02	398	03:22
ZXing	407	00:02	224	01:54
TOTAL	7612	00:26	3112	32:58

(a) Methods and Template Time.

Project	#Generated	Gen Time	Exec Time At	
			Lvl 4	Lvl 1
Checkstyle	193	03:13	01:57	04:21
Codec	142	02:35	10:52	11:32
Compress	143	02:29	02:30	03:10
Configuration	8	00:08	00:04	00:06
Gson	11	00:10	00:03	00:08
JFreeChart	159	02:48	03:18	05:19
JXPath	37	00:40	00:10	00:21
Lang	732	22:23	03:57	13:35
Math	567	12:09	41:49	44:58
Openfire	138	03:12	07:01	08:21
Vectorz	398	07:40	21:25	23:48
ZXing	224	04:02	03:25	04:44
TOTAL	3112	01:07:37	01:58:57	02:40:37

(b) Programs Generated and Execution Time.

Table 4.3: Project Timings.

As can be seen from Table 4.3, JITTEMPLATER was able to convert over 3,000 methods into valid templates in 33 minutes. It is important to note that there is a discrepancy between the number of methods and the number of templates. This discrepancy is

generally due to either 1) the class references and extends another class (this is not currently supported by JITTEMPLATER), or 2) there were no holes that could be leveraged by JITAttack. In the event of no holes that can be leveraged, we throw away that template as it is not useful for our testing purposes. In the thirteen projects we had 3,555 templates that had no holes to be filled.

The ultimate test of the process of utilizing JITTEMPLATER as an input to JITAttack is: were any bugs in the JIT compilers found. To date, we have found three bugs in the Oracle Java JIT compiler that were created from converting a real-world Java programs into templates. The first JIT bug was produced from the Math project, the second was produced from the Checkstyle project, the third was produced from the Codec project. While JITTEMPLATER would still be successful without the uncovering of bugs, due to it fulfilling its desired purpose, it is excellent to have become a valuable input to the JIT bug finding process.

The JIT bug resulting from the template made in the Math project was shown in Figure 2.1b, the bug found was that initializing a very large array (above one billion in length), then setting a random index to a value of 1 caused a crash in high-level JIT optimizations. The bug has been confirmed by Oracle and is listed as bug number `JDK-8271130`.

The JIT bug resulting from the template made in the Checkstyle project was found with the template shown in Figure 4.2. The bug found was that compiling a regex pattern with an empty string, and then trying to match it to a `null` value results in a crash in some JIT optimization levels. The bug has been confirmed by Oracle and is listed as bug number `JDK-8271276`.

```
1  public final class SuppressionsStringPrinterm106Template {
2      private static final Pattern VALID_SUPPRESSION_LINE_COLUMN_NUMBER_REGEX =
           Pattern.compile("^([09]+):([09]+)$");
3      private static final String LINE_SEPARATOR = System.getProperty("line.separator");
4
5      @jitattack.Entry()
6      public static String printSuppressions(File file, String suppressionLineColumnNumber, int
           tabWidth) throws IOException, CheckstyleException {
7          final Matcher matcher = VALID_SUPPRESSION_LINE_COLUMN_NUMBER_REGEX.
               matcher(suppressionLineColumnNumber);
8          if (!matcher.matches()) {
9              final String exceptionMsg = String.format(Locale.ROOT, "%s does not match valid
                   format 'line:column'.", suppressionLineColumnNumber);
10             throw new IllegalStateException(exceptionMsg);
11         }
12         // removed for brevity...
13         return generate(fileText, detailAST, lineNumber, columnNumber, tabWidth);
14     }
15
16     private static String generate(FileText fileText, DetailAST detailAST, int lineNumber, int
           columnNumber, int tabWidth) {
17         final XpathQueryGenerator queryGenerator = new XpathQueryGenerator(detailAST,
               lineNumber, columnNumber, fileText, tabWidth);
18         final List<String> suppressions = queryGenerator.generate();
19         return suppressions.stream().collect(Collectors.joining(LINE_SEPARATOR, "",
               LINE_SEPARATOR));
20     }
21
22     @Argument(1)
23     public static File nonPrim1() {
24         return null;
25     }
26
27     @Argument(2)
28     public static String nonPrim2() {
29         return null;
30     }
31
32     @Argument(3)
33     public static int intArg3() {
34         return intVal().eval();
35     }
36 }
```

Figure 4.2: Checkstyle [8] Bug Template.

The JIT bug resulting from the template made in the Codec project was found with the template shown in Figure 4.3. The bug in Codec was that creating a `StringBuilder` with a capacity of `-1` should always result in an exception being thrown. In some levels of JIT optimization this was not always the case. The bug has been confirmed by Oracle and is listed as bug number `JDK-8271459`.

```
1  public class Sha2Cryptm292Template {
2
3      @jitattack.Entry()
4      public static String sha256Crypt(final byte[] keyBytes) {
5          return sha256Crypt(keyBytes, null);
6      }
7
8      public static String sha256Crypt(final byte[] keyBytes, String salt) {
9          if (salt == null) {
10             salt = SHA256_PREFIX + B64.getRandomSalt(intVal().eval());
11         }
12         return sha2Crypt(keyBytes, salt, SHA256_PREFIX, SHA256_BLOCKSIZE,
               MessageDigestAlgorithms.SHA_256);
13     }
14
15     private static String sha2Crypt(final byte[] keyBytes, final String salt, final String saltPrefix,
           final int blocksize, final String algorithm) {
16         final int keyLen = intVal().eval();
17         int rounds = intVal().eval();
18         boolean roundsCustom = boolVal().eval();
19         // removed for brevity...
20         final String saltString = m.group(intVal().eval());
21         final byte[] saltBytes = saltString.getBytes(StandardCharsets.UTF_8);
22         cnt = keyBytes.length;
23         // removed for brevity...
24         byte[] tempResult = altCtx.digest();
25         final byte[] pBytes = new byte[keyLen];
26         int cp = intVal().eval();
27         final StringBuilder buffer = new StringBuilder(saltPrefix);
28         // removed for brevity...
29         buffer.append(saltString);
30         buffer.append("$");
31         return buffer.toString();
32     }
33
34
35     @Argument(1)
36     public static byte[] nonPrim1() {
37         return new byte[] { 0, 0, 0, 0, 0, 0, 0, 0 };
38     }
39 }
```

Figure 4.3: Codec [2] Bug Template.

Chapter 5

Related Work

Source code manipulation. Source code manipulation has been studied and utilized in research for quite some time [6, 7, 13]. The main purpose of these source code manipulators is to run analysis and make adjustments for other software engineering tools to utilize. JITTEMPLATER uses these source code manipulations. We utilize JavaParser [15] to show us the Abstract Syntax Tree of a Java program so we can modify the source code to be used with JITAttack's testing framework.

Automated test generation. There are many tools in the realm of software engineering that are designed to automatically create test suites. However many of these tools are focused on code coverage as opposed to being primarily about bug detection [5, 12]. Fuzzing has been used as a popular and effective method to find bugs [18], and this seems to be where the core of JITAttack testing ideology stems from. JITTEMPLATER is not concerned with a specific coverage criteria, but rather the amount of holes that can be fed into the system running the tests to determine correctness.

JVM testing. Testing the Java Virtual Machine (JVM) is similar to testing compilers, in that each JVM must adhere to specifications for JVMs [19]. There has been work to utilize differential testing along with mutation of code to reveal bugs inside of JVM implementations [10]. JITTEMPLATER creates inputs for JITAttack which leverages dif-

ferential testing to determine correctness of a JIT compiler, similar to how Chen et al. [10] utilize differential testing for JVM implementation.

Compiler testing. There have been several research projects devoted to specifically testing compilers [9]. Mutation-based fuzzing has been widely used for testing compilers [9]. In this approach, existing programs are mutated and run through the compiler to determine if there are any bugs in the compiler, e.g., the compiler crashes or outputs a wrong compiled program. There has been work to mutate not only the live code of a program but also dead code [22]. Sun et al. found that allowing for mutation of dead code increased the variant space and allowed for a more thorough stress test of a compiler [22].

Equivalence Modulo Inputs (EMI) is another testing technique that has been used to test compilers [9]. EMI takes in code and transforms it into different, yet equivalent, versions of the original code to be tested on a single compiler [16]. EMI was utilized in CLsmith [24] to great success, finding over 50 compiler bugs [9].

Verified compilers. Verified compilers ensure preserving the semantics of the original source code to the compiled code [22]. These compilers have a guarantee of preserving the correctness of the original program. Leroy previously used a proof assistant, Coq, to implement a compiler that is intended to be both certified with a proof as well as be useful and practical for a plethora of programs [17]. The correctness guarantee of a verified compiler is critical for ensuring a program does not have bugs due to incorrect translation of source code to byte code.

Chapter 6

Conclusion

Automatically extracting program templates from real-world Java projects allows for very expressive and widely varying input templates for JITAttack. The automation of such templates is incredibly beneficial, reducing the manual labor of making templates from hours to minutes, as well as allowing for a standard way to implement JITAttack APIs. To date, we have found three bugs using templatized versions of real-world Java projects. Utilizing the data from the console program to gather inputs, the JITTEMPLATER solution to convert methods, and finally JITAttack to run the tests, we believe that many more bugs will be found with our approach, with minimal manual effort.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 2002.

[2] Apache. apache/commons-codec at 4de60e8b68fb749e5380ecef018511bed946bee8.

[3] Apache. apache/commons-math: Miscellaneous math-related utilities.

[4] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, 2009.

[5] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *International Conference on Software Engineering Advances*, pages 252–257. IEEE, 2008.

[6] Marat Boshernitsan and Susan L Graham. ixj: interactive source-to-source transformations for Java. In *Conference on Object-oriented programming systems, languages, and applications*, pages 212–213, 2004.

[7] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[8] Checkstyle. checkstyle/checkstyle at bdaac140eaf161c3055c7d1fe208f21d5f1c629a.

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

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[11] Keith D. Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2011.

[12] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[13] Mark Harman. Why source code analysis and manipulation will always be important. In *Conference on Source Code Analysis and Manipulation*, pages 7–19. IEEE, 2010.

[14] Itti Hooda and Rajender Singh Chhillar. Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13), 2015.

[15] Javaparser. javaparser/javaparser: Java 1-15 parser and abstract syntax tree for java, including preview features to java 13.

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[17] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference on Principles of Programming Languages*, pages 42–54, 2006.

[18] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[19] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. Java Virtual Machine Specification: Java SE 8 Edition. 2015.

[20] William M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[21] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm Java just-in-time compiler. *IBM systems Journal*, 39(1):175–193, 2000.

[22] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.

[23] Maneela Tuteja and Gaurav Dubey. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3):251–257, 2012.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Conference on Programming language design and implementation*, pages 283–294, 2011.

[25] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *International Conference on Quality Software, 2003. Proceedings.*, pages 20–23, 2003.

[26] Zhiqiang Zang, August Shi, and Milos Gligoric. Test templates for Java JIT compilers. 2021.

Vita

Nathaniel ”Nathan” Wiatrek is currently a Software Engineer at United Services Automobile Association (USAA). In this position he designs and implements automation to reduce Computer Engineer work loads. Nathan received his undergraduate degree from Texas A&M University in Electronic Systems Engineering, focusing on Embedded systems development. He currently lives in Houston, Texas.

Permanent address: 2727 Revere St. Apt 5010
Houston, Texas 77098

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.