Java JIT Testing with Template Extraction

Zhiqiang Zang

Fu-Yao Yu

Aditya Thimmaiah

August Shi

Milos Gligoric

TEXAS
The University of Texas at Austin

Supported By

Google

ARMY FUTURES COMMAND
Transforming the Army

NSF

CCF-2107291
CCF-2217696
CCF-2313027

Motivation : Testing compilers is challenging

- Compilers are complex
  - Huge amount of code
    - GCC : 16M LOC
    - LLVM : 18M LOC
    - OpenJDK : 14M LOC
  - Composed of different components - lexer, parser, optimizer, code generation

Motivation : Testing compilers is challenging

- Compilers are complex
  - Huge amount of code
    - GCC : 16M LOC
    - LLVM : 18M LOC
    - OpenJDK : 14M LOC
  - Composed of different components - lexer, parser, optimizer, code generation

- Crafting test inputs (programs) is difficult
  - Non-trivial to construct input to reach deeper components
  - Un-intuitive to construct input to cover specific code blocks inside different components

Motivation : Testing compilers is challenging

- Compilers are complex
  - Huge amount of code
    - GCC : 16M LOC
    - LLVM : 18M LOC
    - OpenJDK : 14M LOC
  - Composed of different components - lexer, parser, optimizer, code generation

- Crafting test inputs (programs) is difficult
  - Non-trivial to construct input to reach deeper components
  - Un-intuitive to construct input to cover specific code blocks inside different components

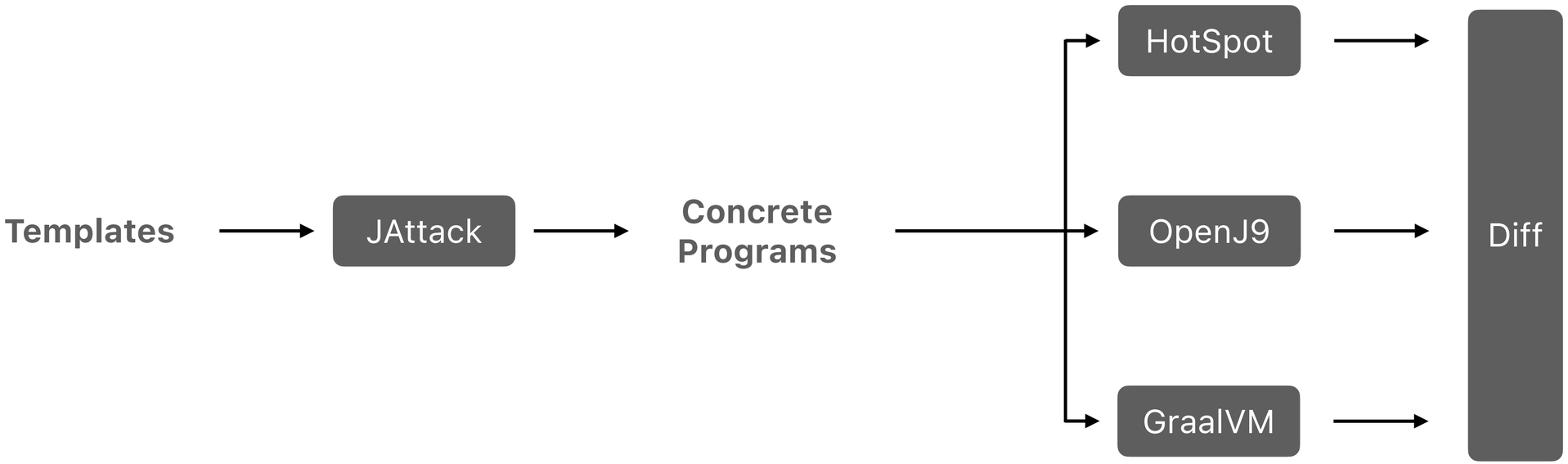
- Current approaches of compiler testing
  - Manually written programs - LLVM test suites, OpenJDK test suites etc - Manual
  - Automatically generated programs - Csmith, Java* Fuzzer - Inflexible
  - Hybrid - JAttack[1]
    - Developers intuition with manually written templates + auto gen concrete programs
    - Templates still have to be written manually!

OpenJDK

[1] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric and August Shi. Compiler Testing using Template Java Programs. In ASE 2022. 1 - 13

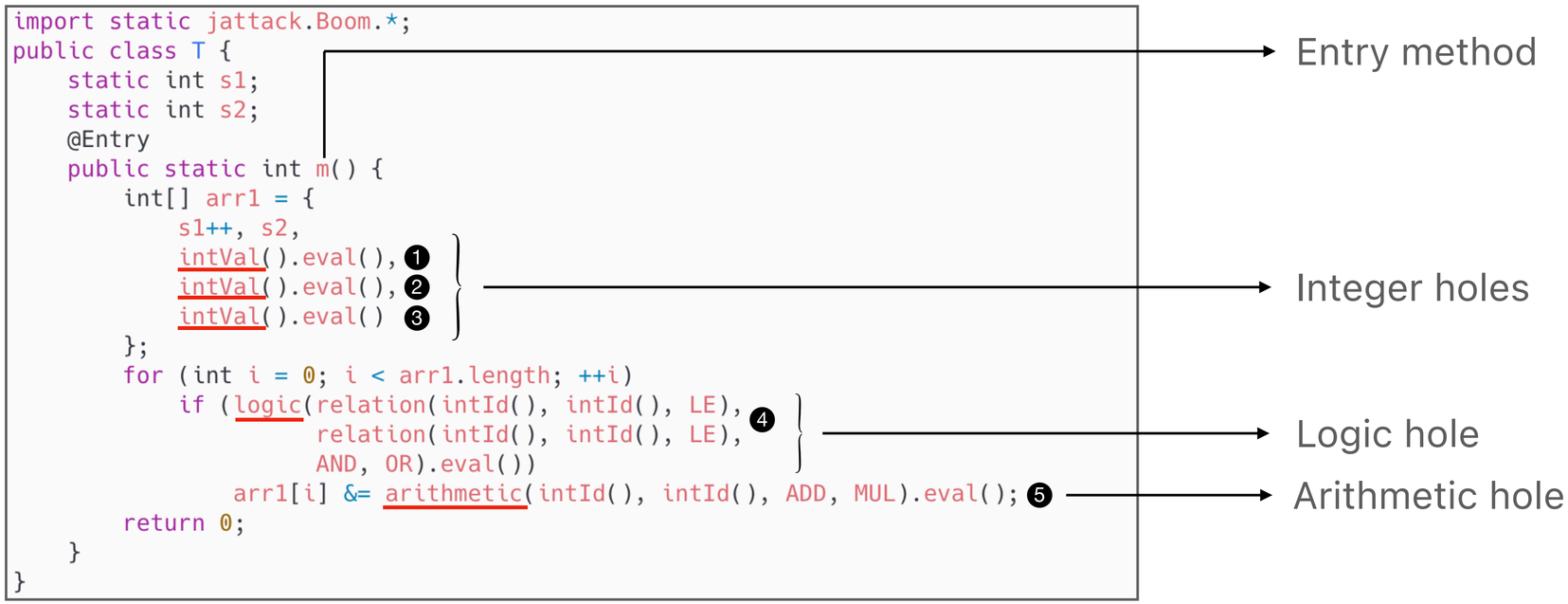
JAttack : Overview

- Manually write template programs in Java using JAttack's DSL
- JAttack generates concrete programs from templates
- Differentially test generated programs on Java JIT compilers



JAttack : Templates

- A template contains holes for expressions which are later filled to generate multiple concrete programs



JAttack : Program Generation

- JAttack generates several concrete programs from a template

```
import static jattack.Boom.*;
public class T {
    static int s1;
    static int s2;
    @Entry
    public static int m() {
        int[] arr1 = {
            s1++, s2,
            intVal().eval(), ❶
            intVal().eval(), ❷
            intVal().eval()  ❸
        };
        for (int i = 0; i < arr1.length; ++i)
            if (logic(relation(intId(), intId(), LE), ❹
                      relation(intId(), intId(), LE),
                      AND, OR).eval())
                arr1[i] &= arithmetic(intId(), intId(), ADD, MUL).eval(); ❺
        return 0;
    }
}
```

Template (manually written)

Filling holes

Concrete Program #1

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

Concrete Program #2

```
import static jattack.Boom.*;
public class TGen {
    static int s1;
    static int s2;
    @Entry
    public static int m() {
        int[] arr1 = {
            s1++,
            s2,
            26344342,❶
            3478638, ❷
            3269054  ❸
        };
        for (int i = 0; i < arr1.length; ++i)
            if (arr1[0] <= s1 || s1 <= arr1[1])❹
                arr1[i] &= arr1[1] + s2; ❺
        return 0;
    }
}
```

JAttack : Limitations

- Manually writing templates
- Only static methods supported
- Entry methods cannot have arguments

```
import static jattack.Boom.*;
public class T {
    static int s1;
    static int s2;
    @Entry
    public static int m() {
        int[] arr1 = {
            s1++, s2,
            intVal().eval(), ❶
            intVal().eval(), ❷
            intVal().eval()  ❸
        };
        for (int i = 0; i < arr1.length; ++i)
            if (logic(relation(intId(), intId(), LE), ❹
                      relation(intId(), intId(), LE),
                      AND, OR).eval())
                arr1[i] &= arithmetic(intId(), intId(), ADD, MUL).eval(); ❺
        return 0;
    }
}
```

Template (manually written)

Filling holes

Concrete Program #1

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

Concrete Program #2

```
import static jattack.Boom.*;
public class TGen {
    static int s1;
    static int s2;
    @Entry
    public static int m() {
        int[] arr1 = {
            s1++,
            s2,
            26344342,❶
            3478638, ❷
            3269054  ❸
        };
        for (int i = 0; i < arr1.length; ++i)
            if (arr1[0] <= s1 || s1 <= arr1[1])❹
                arr1[i] &= arr1[1] + s2; ❺
        return 0;
    }
}
```

Our Contributions

- Framework - Design and Implementation
  - LeJit for automatically templatizing existing code in Java
  - Two algorithms (Pool based and Test based) to collect
    - Template entry methods from Java projects
    - Code sequences to construct arguments for entry methods

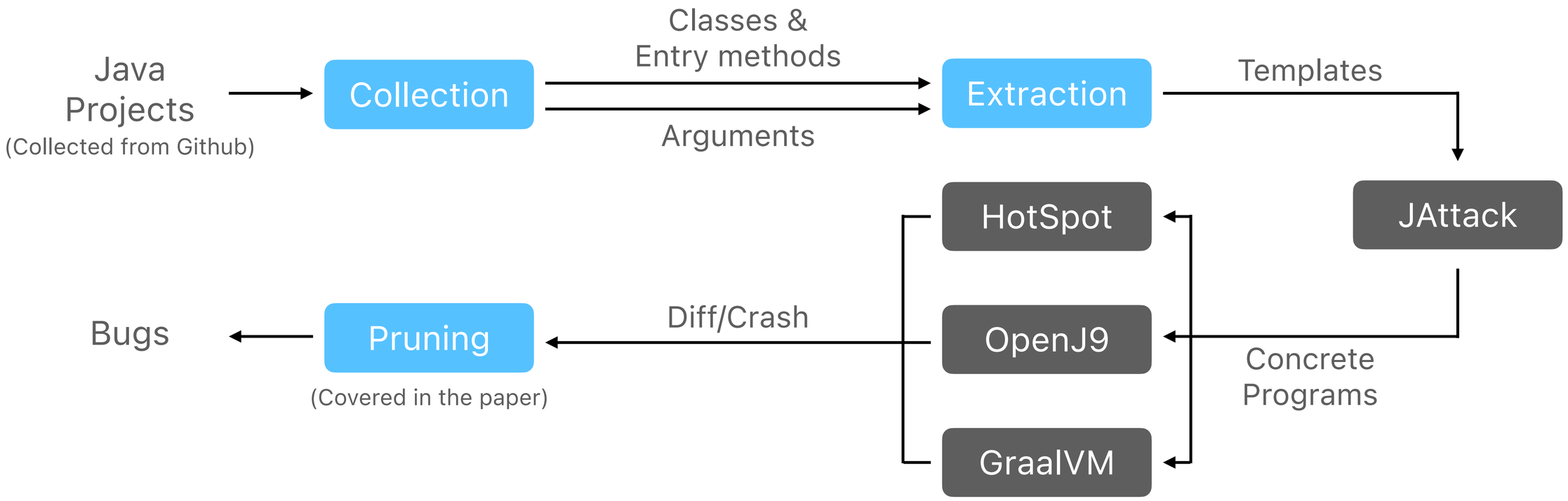
Our Contributions

- Framework - Design and Implementation
  - LeJit for automatically templatizing existing code in Java
  - Two algorithms (Pool based and Test based) to collect
    - Template entry methods from Java projects
    - Code sequences to construct arguments for entry methods
- Analysis
  - Impact of Java language features and type of holes on LeJit

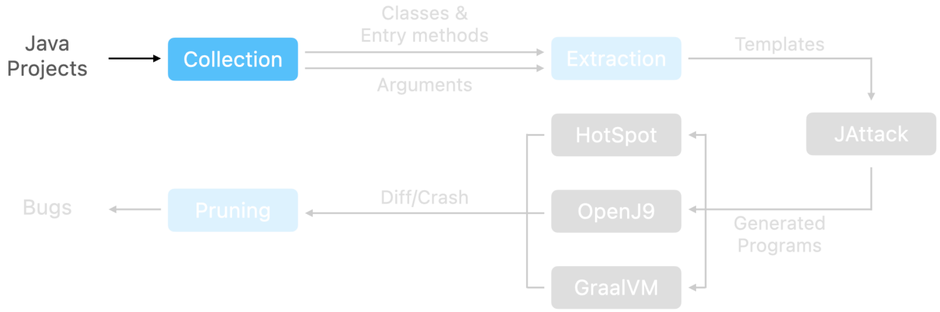
Our Contributions

- Framework - Design and Implementation
  - LeJit for automatically templatizing existing code in Java
  - Two algorithms (Pool based and Test based) to collect
    - Template entry methods from Java projects
    - Code sequences to construct arguments for entry methods
- Analysis
  - Impact of Java language features and type of holes on LeJit
- Results
  - Tested three compilers – Oracle HotSpot, IBM OpenJ9, and Oracle GraalVM
  - Comparison with JITfuzz and JavaTailor, SOTA for testing JVMs
  - Discovered 15 new bugs including two CVEs

LeJit : Framework Architecture



Collection : Setup



❶ Get a Java project

(Eg: Apache commons-text)

GitHub

❷ Pick any class

(Eg: StrBuilder class)

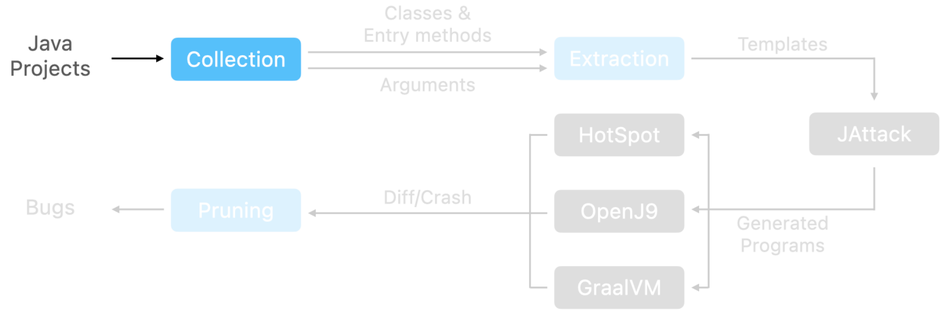
```
public class StrBuilder implements ... {
    ...
    public StrBuilder trim() {
        ...
        int len = size;
        final char[] buf = buffer;
        int pos = 0;
        while (pos < len && buf[pos] <= ' ') {
            pos++;
        }
        ...
        return this;
    }
}
```

❸ Generate unit tests for picked class

```
@Test
public void test1() {
    StrBuilder sb1 = new StrBuilder("date");
    StrBuilder sb2 = sb1.append((Object) 10.0);
    StrBuilder sb3 = sb1.appendSeparator("d");
    Object[] arr = new Object[] {1.0};
    StrBuilder sb4 = sb1.append("resourceBundle", arr);
    sb4.trim();
}
```

Collection : Entry Methods

- How can we collect entry methods for templates?



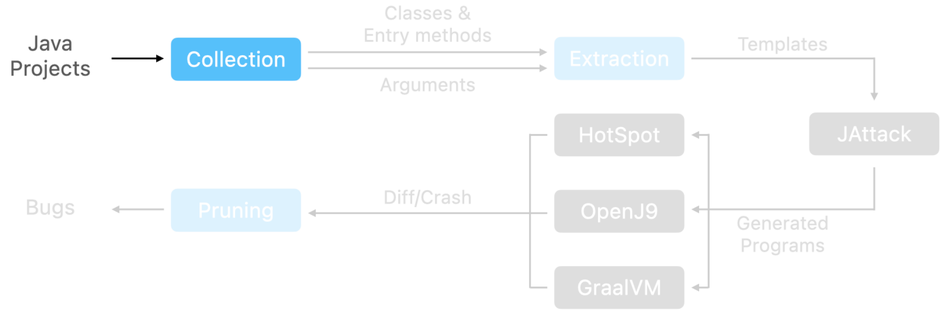
❸ Generate **unit tests** for picked class

❹ Last method call in unit tests is selected as entry method

```
@Test
public void test1() {
    StrBuilder sb1 = new StrBuilder("date");
    StrBuilder sb2 = sb1.append((Object) 10.0);
    StrBuilder sb3 = sb1.appendSeparator("d");
    Object[] arr = new Object[] {1.0};
    StrBuilder sb4 = sb1.append("resourceBundle", arr);
    sb4.trim();
}
```

Collection : Arguments

- How can we collect arguments for entry methods?

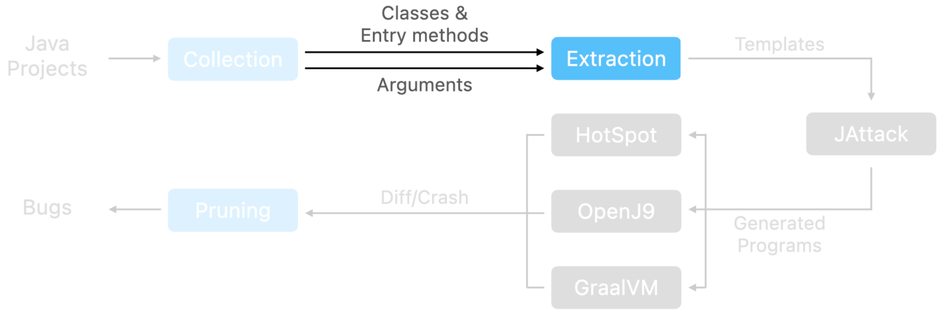


❸ Generate **unit tests** for picked class

❺ Collect code sequence preceding last method call as argument generating code block

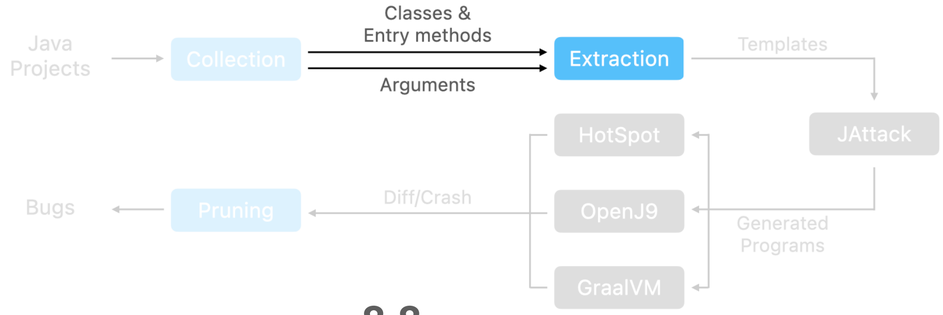
```
@Test
public void test1() {
    StrBuilder sb1 = new StrBuilder("date");
    StrBuilder sb2 = sb1.append((Object) 10.0);
    StrBuilder sb3 = sb1.appendSeparator("d");
    Object[] arr = new Object[] {1.0};
    StrBuilder sb4 = sb1.append("resourceBundle", arr);
    sb4.trim();
}
```

Extraction



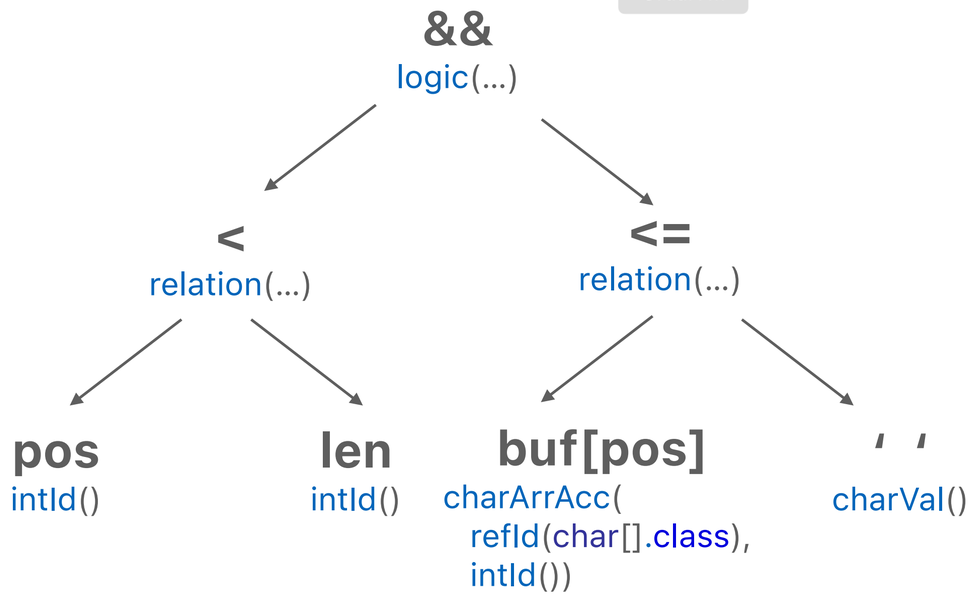

```
public class StrBuilder implements...{
    ...
    public StrBuilder trim() {
        int len = size; ❶
        final char[] buf = buffer; ❷
        int pos = 0; ❸
        while (pos < len && buf[pos] <= ' ') { ❹
            pos++;
        }
        ...
        return this;
    }
}
```

Extraction : Templatizing

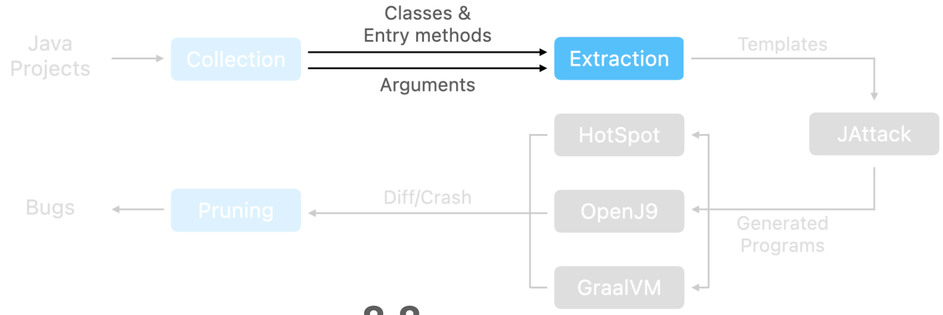



```java
public class StrBuilder implements...{
    ...
    public StrBuilder trim() {
        int len = size; ❶
        final char[] buf = buffer; ❷
        int pos = 0; ❸
        while (pos < len && buf[pos] <= ' ') { ❹
            pos++;
        }
        ...
        return this;
    }
}
```

Templatize

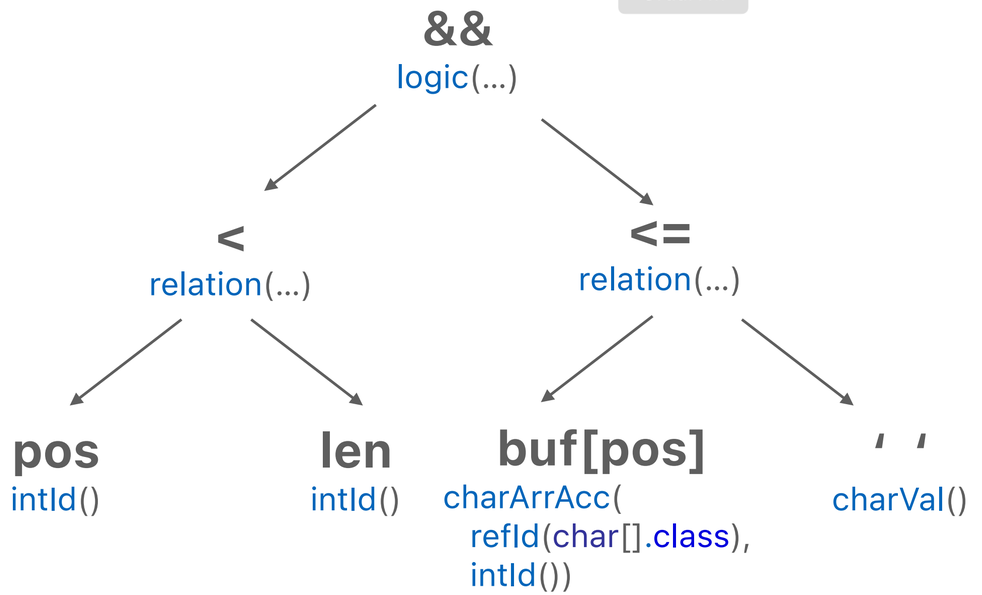


Extraction : Templatizing



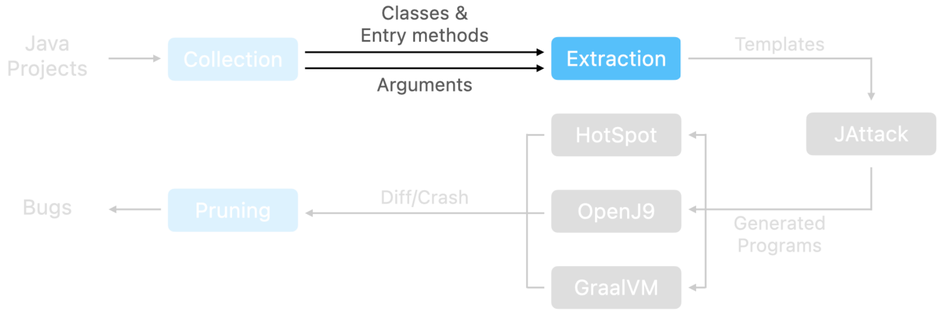

```java
public class StrBuilder implements...{
    ...
    public StrBuilder trim() {
        int len = size; ❶
        final char[] buf = buffer; ❷
        int pos = 0; ❸
        while (pos < len && buf[pos] <= ' ') { ❹
            pos++;
        }
        ...
        return this;
    }
}
```

Templatize




```
logic(
   relation(intId(), intId()),
   relation(charArrAcc(refId(char[].class), intId()), charVal())
)
```

Extraction : Templatizing

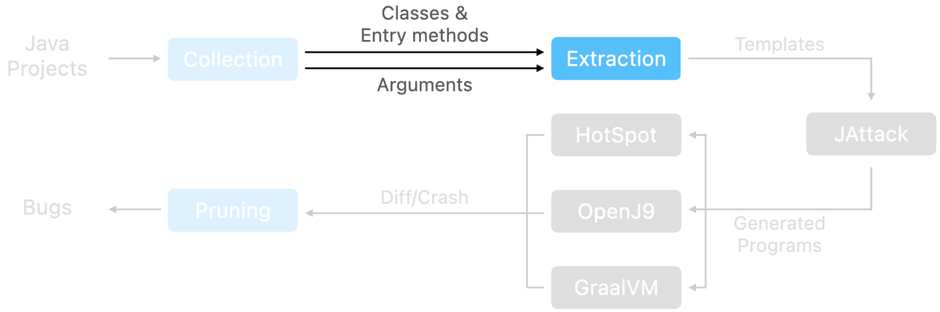



```java
public class StrBuilder implements...{
    ...
    public StrBuilder trim() {
        int len = size; ❶
        final char[] buf = buffer; ❷
        int pos = 0; ❸
        while (pos < len && buf[pos] <= ' ') { ❹
            pos++;
        }
        ...
        return this;
    }
}
```

Templatize

```java
public class StrBuilder {
    @Entry
    public StrBuilder trim() {
        int len = intId().eval();❶
        final char[] buf = refId(char[].class).eval();❷
        int pos = intVal().eval(); ❸
        int _limiter1 = 0;
        while (logic(                                    ❹
                relation(intId(), intId()),
                relation(charArrAcc(refId(char[].class),
                    intId()), charVal())
                ).eval() && _limiter1++ < 1000) {
            pos++;
        }
        return this;
    }
}
```

Extraction : Add Arguments




```java
public class StrBuilder {
    @Entry
    public StrBuilder trim() {
        int len = intId().eval();❶
        final char[] buf = refId(char[].class).eval();❷
        int pos = intVal().eval();❸
        int _limiter1 = 0;
        while (logic(                                    ❹
                relation(intId(), intId()),
                relation(charArrAcc(refId(char[].class),
                    intId()), charVal())
                ).eval() && _limiter1++ < 1000) {
            pos++;
        }
        return this;
    }
}
```

Add arguments to make templates executable

```java
@Argument(0)
public static StrBuilder _arg0() {
    StrBuilder sb1 = new StrBuilder("date");
    StrBuilder sb2 = sb1.append((Object) 10.0);
    StrBuilder sb3 = sb1.appendSeparator("d");
    Object[] arr = new Object[] {
        1.0
    };
    StrBuilder sb4 = sb1.append("resourceBundle", arr);
    return sb4;
}
```

From Collection Phase

Evaluation

RQ1: What critical bugs does LeJit detect in Java JIT compilers?

RQ2: How efficient is JAttack at generating programs from templates?

RQ3: What is the impact of holes in various Java language features on LeJit's bug detection?

RQ4: What is the impact of different kinds of templates on LeJit's bug detection?

RQ5: What are contributions of major components of LeJit?

RQ6: How effective is LeJit compared with the state-of-the-art techniques?

RQ1 : What critical bugs does LeJit detect in Java JIT compilers?

- 15 previously unknown bugs including 2 CVEs
- Popular JVMs: Hotspot, OpenJ9, GraalVM

JVM	Bug ID
GraalVM	GR-45498
HotSpot	JDK-8239244/CVE-2020-14792 JDK-8258981 JDK-8271130/CVE-2022-21305 JDK-8271276 JDK-8271459 JDK-8271926 JDK-8297730 JDK-8301663 JDK-8303946 JDK-8304336/CVE-2023-22044 JDK-8305946/CVE-2023-22045
OpenJ9	17066 17129 17139 17171 17212 17249 17250

→ 2 New CVEs

- CVE-2023-22044: ArrayIndexOutOfBoundsException not thrown from inside loop for C1 tiered compilation
- CVE-2023-22045: JRE crash due to JIT compiling ArrayIndexOutOfBoundsException from a loop

RQ4 : What is the impact of different kinds of templates on LeJit's bug detection?

We generate templates with only one kind of hole allowed to study the impact of different language features

	xxxId	xxxVal	arithmetic & shift	logic & relation	All
#Templates	13,090	12,139	13,606	13,682	12,061
#Programs	105,759	87,230	59,917	64,906	102,670
#Failures	106	29	39	70	131
#Bugs*	1	1	3	1	4

*This is part of our experiments, which does not contain all the bugs we discovered through the entire evaluation

Conclusion

Power of combining developers' insights via templates with automated testing

- LeJit: End-to-end testing framework around JAttack for Java JIT testing
  - Collects classes, entry methods and arguments from Java projects with tests
  - Templatizes existing Java programs by converting expressions into holes and adding arguments
- Large-scale evaluation for LeJit
  - Extracted ~140K templates from 77 Java projects
  - Generated ~900K concrete programs
- Results
  - Language features and template kinds have impact on bug detection
  - 15 new bugs (2 CVEs) in popular JVMs (HotSpot, OpenJ9, and GraalVM)

JAttack : https://github.com/EngineeringSoftware/jattack (Our recent work)

LeJit : https://github.com/EngineeringSoftware/lejit (This work)

Contact : Aditya Thimmaiah (auditt@utexas.edu)

Appendix

RQ3 Language Features

What is the impact of holes in various Java language features on LeJit's bug detection?

	Total	Arrays	Conditional	Loops	Ref Arguments
# Failures due to Bugs	**174**	70	144	127	142
# Bugs (unique)	**10**	7	8	8	9

RQ2 Efficiency

- 23 hand-written templates
- 1K generated programs per template
- 21m 20s for generating all 23K programs
- Over 2 days without our generation optimizations, 99.50% reduction

RQ5 Components

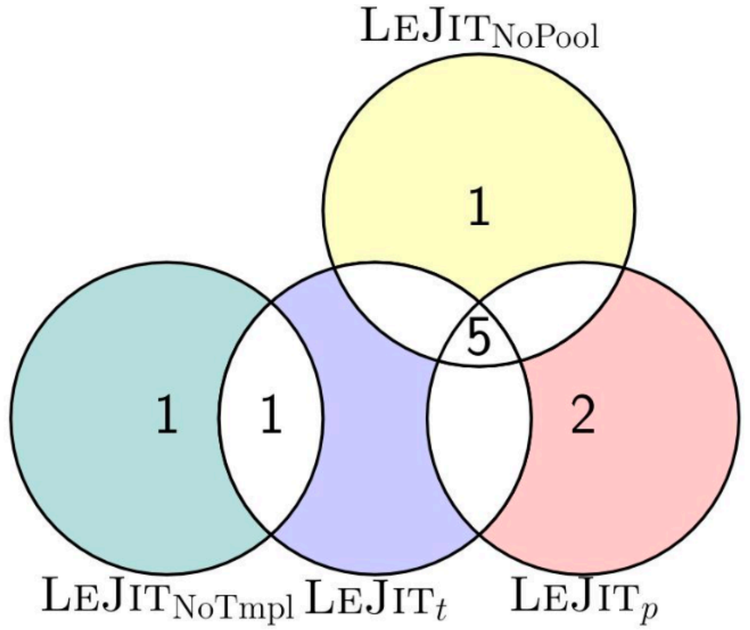
What are contributions of major components of LeJit?

- Holes? LeJit_t (test-based) vs. LeJit_NoTmpl (~= no extraction phase)
- Arguments? LeJit_p (pool-based) vs. LeJit_NoPool (~= no collection phase)

	#Tmpl.	#Gen.	#Fail.*	#Bugs	Coverage(%) C1**	Coverage(%) C2**
LeJit_t	10,714	90,916	66	3.3	83.8	79.0
LeJit_NoTmpl	14,854	14,854	24	0.7	83.6	78.4
LeJit_p	11,921	99,644	129	5.3	84.4	79.6
LeJit_NoPool	10,185	89,977	56	4.0	84.0	78.7

Tmpl. = Templates, Gen. = Generated programs, Fail. = Failures, all numbers are average of 3 runs

*: Failures include false positives, and even multiple true positives could point to a same bug
**: C1 and C2 are the two JIT compilers in HotSpot



RQ6 SOTA

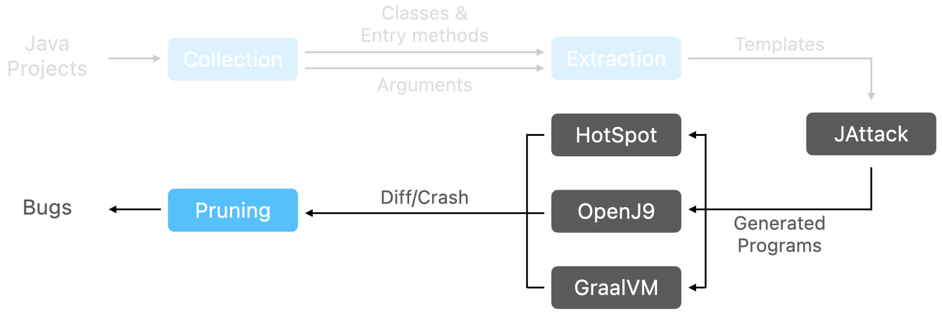
How effective is LeJit compared with the state-of-the-art techniques?

- Java* Fuzzer[1], grammar-based
- JavaTailor[2], mutation-based from historical bug reports
- JITfuzz[3], mutation-based with designed mutators for JIT

	#Generated Programs	#Failures	#Bugs	Coverage(%)	
				C1	C2
Java* Fuzzer	15,931	0	0	80.6	67.5
JavaTailor	/	102	1*	74.4	68.5
JITfuzz	45,352	2,115	1*	69.7	64.3

[1] Azul Systems, Inc. Azulsystems/JavaFuzzer: Java* Fuzzer for Android*, 2018. https://github.com/AzulSystems/JavaFuzzer.
[2] Yingquan Zhao, et al. 2022. History-Driven Test Program Synthesis for JVM Testing. In International Conference on Software Engineering.
[3] Mingyuan Wu, et al. 2023. JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers. In International Conference on Software Engineering. *: Both bugs were previously reported. JavaTailor: OpenJ9 13242 (not related to JIT); JITfuzz: JDK-8280126

Pruning



- Non-deterministic features
  - Timestamp, randomness, hashcode, etc
- Discrepancies of JVM implementations
  - HotSpot and OpenJ9 disagree on maximum array size