JATTACK: Java JIT Testing using Template Programs

Zhiqiang Zang, Fu-Yao Yu, Nathan Wiatrek, Milos Gligoric, and August Shi
zhiqiang.zang@utexas.edu, fu.yao.yu@utexas.edu, nwiatrek@utexas.edu, gligoric@utexas.edu, august@utexas.edu
The University of Texas at Austin
Austin, TX, USA

***Abstract*—We present JATTACK, a framework that enables compiler testing using templates. JATTACK allows compiler developers to write a template program that describes a set of concrete programs to be used to test compilers. Such a template-based approach leverages developers' intuition on testing compilers, by allowing developers to write a template program in the host programming language (Java), which contains a basic program structure while provides an opportunity to express variants of specific language constructs in holes. Each hole, written in a domain-specific language embedded in the host language, is used to construct an extended abstract syntax tree (eAST), which defines the search space of a language construct, e.g., a set of numbers, expressions, statements, etc. JATTACK executes the template program to fill every hole by randomly choosing a number, expression, or statement within the search space defined by the hole, and it generates concrete programs with all holes filled. We used JATTACK to test Java just-in-time (JIT) compilers, and we have found seven critical bugs in Oracle JDK JIT compiler. Oracle developers confirmed and fixed all seven bugs, five of which were previously unknown, including two CVEs (Common Vulnerabilities and Exposures). JATTACK blends developers' intuition via templates with random testing to detect bugs in compilers. The demo video for JATTACK can be found at https://www.youtube.com/watch?v=meCFPxucqk4.**

***Index Terms*—Testing, test generation, program generation, compiler, templates**

I. INTRODUCTION

Compilers are the cornerstone of software development, and their correctness is vital. Compiler developers have written thousands of tests, i.e., programs in the compiler's target programming language, to check correctness of compilers [1]. Such hand-written tests nicely capture developers' intuition of what programs are more likely to trigger corner cases, but it is time-consuming to write a large number of tests. On the other hand, automated techniques [2]–[4] can generate a large number of programs used as test inputs to compilers, but they do not incorporate compiler developers' domain knowledge and insights into the testing process.

We present JATTACK [5], which blends developers' insights with automated testing. Using JATTACK, a developer writes a *template program* (template for short) that is similar to the tests they already write for compilers, but they express variants of the test using the template. (Figure 1 shows an example template, which is discussed in detail in Section II.) The goal of writing a template is similar to parameterized unit testing [6], where developers hand-write tests but use parameters to provide their insights for deeper exploration. JATTACK allows developers to specify exactly how to generate variants of a test.

Every template contains *holes* that JATTACK will fill. Each hole is written in a domain-specific language (DSL) embedded in the host programming language (Java). Using the DSL, developers can specify exactly how they want the hole to be filled. JATTACK constructs an *extended abstract syntax tree* (eAST) from the hole, which bounds the search space for the hole. The DSL is implemented with a set of APIs in JATTACK, e.g., `relation(intVal(), intVal(), GT, LT).eval()` defines a hole that represents a logical relation connecting two integer literals (each taking value between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`) using either $>$ (`GT`) or $<$ (`LT`) operators. This hole evaluates to a `boolean`.

JATTACK repeatedly executes the given template to fill each hole and as a result generates a concrete program. During execution, when it reaches a hole for the first time, JATTACK randomly chooses a number or expression available within the search space defined by the hole to fill the hole in the remaining execution. Next, since we focus on testing Java JIT compilers, every generated program from the template is executed a large number of times (so that JIT compilation is triggered [7]) using different JIT compilers, detecting bugs via differential testing [8].

We wrote 84 templates focusing on different Java language features and learning from existing tests for Java JIT compilers. As part of our evaluation on how well JATTACK can be used for automated compiler testing, we also automatically extracted 5,419 templates from 77 open-source Java projects in a wide variety of domains involving different Java language features. Using these templates, JATTACK detected seven bugs in the Oracle JDK JIT compiler. Oracle developers confirmed and fixed all seven bugs, five of which were previously unknown including two CVEs (Common Vulnerabilities and Exposures) that they acknowledged.

JATTACK is open source and publicly available on GitHub at https://github.com/EngineeringSoftware/jattack.

II. EXAMPLE

Figure 1a shows a template program we wrote using JATTACK. Note that every template is a valid program, which means it can be type-checked, compiled and executed the same as any other Java program. This template involves different

```
1  import static jattack.Boom.*;
2  public class T {
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] &= arithmetic(intId(), intId(),
14                                ADD, MUL).eval() ❺;
15     return 0; } }
```

(a) An example of a template.

```
1  import static jattack.Boom.*;
2  public class TGen {
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.

Fig. 1: An example of a template and a program generated from the template.

Java language features, e.g., arrays, for loops, static variables, etc., to exercise Java JIT optimizations.

There are five holes in the template, three between lines 7 and 8, one between lines 10 and 12, and one between line 13 and 14. Each hole represents a place that JATTACK should fill in with a concrete number or expression. The first three holes (lines 7-8), defined by the `intVal` call, should be filled with integer literals between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. The next hole (lines 10-12) is defined by a logical relation expression (`logic` call) connecting two relational expressions using either `&&` (`AND` argument) or `||` (`OR` argument). Each relational expression connects two available integer variables (`intId` call) at this point, which can be `s1`, `s2`, `i` or any element of `arr1`. The last hole (lines 13-14) represents an arithmetic expression that adds (`ADD` argument) or multiplies (`MUL` argument) two available integer variables.

JATTACK *generates* programs from the given template in an *execution-based* model. Namely, JATTACK fills holes by executing the template. (The execution-based model provides unique advantages over static generation [5].) Every template has to define an entry method, annotated with `@Entry` (line 5), which is the starting point of execution. When JATTACK reaches an unfilled hole during execution for the first time, it randomly picks a valid expression within the search space defined by the hole, to fill the hole. When all reachable holes are filled, JATTACK outputs a concrete program with holes replaced by corresponding expressions as a generated program. Then, JATTACK repeats the entire generation procedure to generate the next program up to the specified maximum number of programs. Figure 1b shows one of the generated programs; each circled number corresponds to the same number in the template shown in Figure 1a.

Finally, for every generated program, the same entry method is executed a large number of times using different JIT compilers, and results from different compilers are collected for differential testing [8]. Note that Java JIT compilers optimize code sections that are frequently executed, so repeated execution of the entry method is necessary to trigger JIT compiler optimizations. The repeated execution of the generated program shown in Figure 1b exposed a bug which crashed the JVM due to an incorrect optimization for `if` nodes in the Oracle JDK JIT compiler.

III. Technique and Implementation

In this section, we describe design and implementation of JATTACK's DSL in Java, and the workflow of the entire JATTACK framework [5].

A. Design and Implementation of DSL

We design the DSL of JATTACK so that (1) developers decide where to place a hole and define the search space for the hole, and (2) developers do not need to learn a new programming language or change the compiler or runtime of the host programming language (Java). We first briefly describe the syntax and semantics of the DSL. The new concept we introduced on top of Java is the *hole*. A hole can represent any number, expression, or statement, and the hole obeys the same grammar rules as the language construct it represents. When a hole is evaluated, it generates a concrete number, expression or statement, which is *randomly* chosen from all possible candidates in the search space defined by the hole, and the hole computes the result of the chosen candidate. Any subsequent calls to the same hole during the same run will always use the same candidate, because a hole could be evaluated more than once, i.e., in a loop. However, the actual result computed from the same hole could change at subsequent calls, since even the same candidate could be evaluated to different values at different times, i.e., an integer variable as an iteration counter in a loop.

To support the concept of holes and to integrate it into Java without changing the Java language itself, we introduce a set of API methods that construct holes. Every API method returns an eAST with a specific type that corresponds to a hole to be filled, e.g., the `intVal(int min, int max)` method, which represents a hole to be filled with an `int` number, creates an `IntVal` node that evaluates to any integer between `min` and `max`. The eAST contains a range of candidates to fill the hole. As an example, consider the logical expression hole in Figure 1a (line 10-12), which returns a root node of an eAST, illustrated in Figure 2. Candidates for the hole are obtained by recursively obtaining candidates for nodes in subtrees and combining them together. `RelExp` nodes combine candidates for integer variables and connect them with the specified operators (LE), and the `LogExp` node again combines the candidates returned from `RelExp` nodes with the specified operators AND or OR, as the final candidates. For our Java implementation, creating an eAST does not necessarily mean filling the hole; only after calling the `eval` method does a candidate get randomly generated to fill the hole. Once generated, the candidate is memoized and subsequent calls to the same hole will always compute the result using the candidate rather than re-fill the hole.

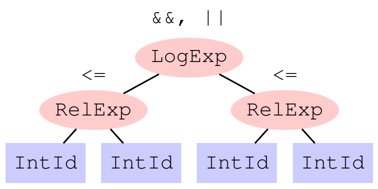


Fig. 2: eAST corresponding to `logic` hole from Figure 1a.

B. Generation and JIT-Testing

The workflow of JATTACK framework can be divided into two phases: generation and JIT-testing. In the generation phase, JATTACK executes the given template to generate the desired number of concrete programs. Next, in the JIT-testing phase, JATTACK runs those generated programs over different JIT compilers and detect bugs through differential testing [8]. Figure 3 shows a high level overview of the two phases.

Generation. Given a template T, JATTACK first captures the initial global state of the template and finds the entry method in the template. Next, JATTACK repeatedly executes the entry method until it fills all the holes in the template or it reaches the given maximum iterations N, whichever comes first. Then, JATTACK replaces every hole with the corresponding concrete generated code and outputs a concrete generated program. This concludes the generation of one program. Next, JATTACK resets the state of the template to be the same as the captured initial global state and repeats all the previous steps to generate more programs until it generates the specified number of programs (M).

JIT-Testing. For each generated program from the given template, JATTACK runs the program through executing the entry method a large number of times (N) (so as to trigger JIT compilation). JATTACK hashes the return value from the entry method every time, and the final global state of the generated program after all the iterations, into a final checksum value. JATTACK repeats the execution on different JIT compilers and compares the checksum values. Any difference between the checksum values indicates a bug within some JIT compiler. Also, JATTACK reports a bug if the execution crashes on any JIT compiler.

IV. Tool Installation and Usage

A. Installation

The first step is to clone the JATTACK repository and to check out the tag for the demo.

```
$ git clone https://github.com/EngineeringSoftware/jattack
$ cd jattack
$ git checkout icse23-demo
```

JATTACK requires at least JDK 11 and a Python 3.8 environment with pip package installer [9]. We assume the dependencies are available on the system. To install JATTACK:

```
$ ./tool/install.sh
```

This command calls a bash script to build Java jars, install required Python packages, and create an executable. If the command completes normally, an executable file `jattack` will appear in `tool` directory, i.e, `./tool/jattack`.

B. Usage

After installation, users can interact with JATTACK through the executable file `./tool/jattack`, e.g.,

```
$ ./tool/jattack -h
```

We provide a sample template `./T.java` in the repository. To run JATTACK with the template, users need to provide JATTACK with at least two required arguments: (1) `--clz`, the fully qualified class name of the given template, and (2) `--n_gen`, the total number of generated programs, and several optional arguments (with default values if not provided), e.g., `--seed` (the random seed used during generation), `--java_envs` (the Java environments to be differentially tested, each including the path to Java home and a list of Java options), etc. (See more arguments in a help message via `-h`.)

```
$ ./tool/jattack --clz T --n_gen 3 --seed 42 \
    --java_envs "[\
      [.downloads/jdk-11.0.8+10,[-XX:TieredStopAtLevel=4]],\
      [.downloads/jdk-11.0.8+10,[-XX:TieredStopAtLevel=1]]]"
```

This command generates 3 programs from the provided template `T.java` and uses these generated programs to test specified Java JIT compilers, i.e., level 4 and level 1 [10] of JIT compilers from the JDK installed at `.downloads/jdk-11.0.8+10`. Figure 4 shows a screenshot of running the command. As a result, the first generated program crashes level 4 of the JIT compiler and reports a JIT bug. JATTACK creates a hidden directory `.jattack` to save the generated programs and outputs (i.e., checksum values) of the programs' execution.

V. Evaluation

We wrote 84 templates focusing on different Java language features, e.g., arrays, loops, conditions, etc., and learning from existing tests for Java JIT compilers. We let JATTACK generate 1,000 programs for each template. Using these templates we found two bugs in the Oracle JDK JIT compiler. We also evaluated the efficiency of JATTACK at generating programs and executing those generated programs, using 23 of these hand-written templates. It took around 20 minutes to generate all 23,000 programs, and the total execution time across all generated programs is around two hours on level 4 and around two and a half hours on level 1.

We also used JATTACK for automated compiler testing via extracted templates from a large number of existing Java programs and compared its effectiveness with an existing automated compiler testing tool, Java* Fuzzer [2], which is a fuzzer tool that has been successful at detecting bugs in the

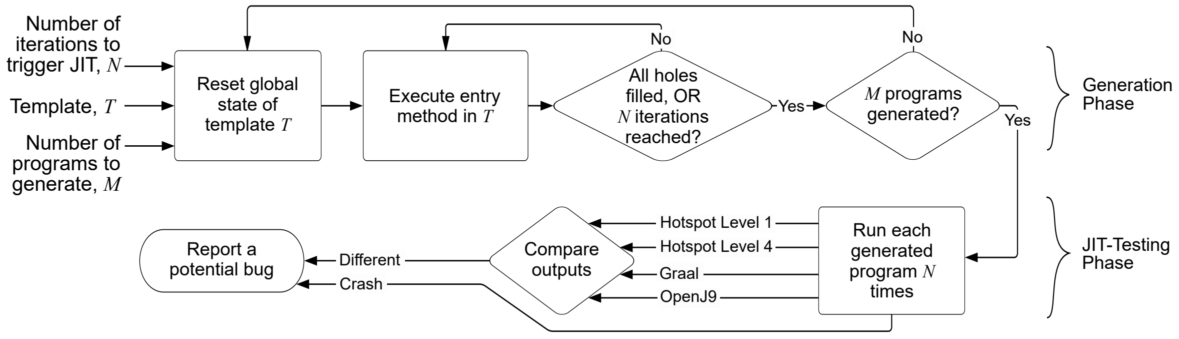
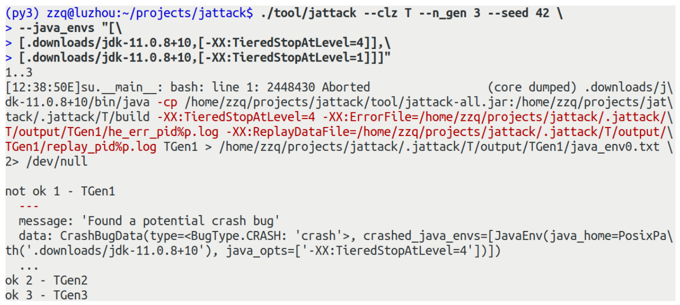

Number of iterations to trigger JIT, N
Template, T
Number of programs to generate, M
Reset global state of template T
Execute entry method in T
All holes filled, OR N iterations reached?
No
Yes
M programs generated?
No
Yes
Generation Phase
Run each generated program N times
Hotspot Level 1
Hotspot Level 4
Graal
OpenJ9
Compare outputs
Different
Crash
Report a potential bug
JIT-Testing Phase

Fig. 3: Overview of JATTACK framework.



```
(py3) zzq@luzhou:~/projects/jattack$ ./tool/jattack --clz T --n_gen 3 --seed 42 \
> --java_envs "[\
> [.downloads/jdk-11.0.8+10,[-XX:TieredStopAtLevel=4]],\
> [.downloads/jdk-11.0.8+10,[-XX:TieredStopAtLevel=1]]]"
1..3
[12:38:50E]su.__main__: bash: line 1: 2448430 Aborted                 (core dumped) .downloads/j\
dk-11.0.8+10/bin/java -cp /home/zzq/projects/jattack/tool/jattack-all.jar:/home/zzq/projects/jat\
tack/.jattack/T/build -XX:TieredStopAtLevel=4 -XX:ErrorFile=/home/zzq/projects/jattack/.jattack/\
T/output/TGen1/he_err_pid%p.log -XX:ReplayDataFile=/home/zzq/projects/jattack/.jattack/T/output/\
TGen1/replay_pid%p.log TGen1 > /home/zzq/projects/jattack/.jattack/T/output/TGen1/java_env0.txt \
2> /dev/null

not ok 1 - TGen1
  ---
  message: 'Found a potential crash bug'
  data: CrashBugData(type=<BugType.CRASH: 'crash'>, crashed_java_envs=[JavaEnv(java_home=PosixPa\
th('.downloads/jdk-11.0.8+10'), java_opts=['-XX:TieredStopAtLevel=4'])])
  ...
ok 2 - TGen2
ok 3 - TGen3
```

Fig. 4: Screenshot of using JATTACK from command-line.

TABLE I: Detected JIT Bugs.

Bug ID	**Type**	**Priority**	**Affected JDK Versions**	**Status**
JDK-8239244	Diff	CVE	8, 11, 13	Fixed
JDK-8258981	Crash	P3	9, 10, 11, 15, 16	Fixed
JDK-8271130	Crash	CVE	8, 11, 16, 17	Fixed
JDK-8271276	Crash	P2	16, 17, 18	Fixed
JDK-8271459	Diff	P2	8, 11, 16, 17, 18	Fixed
JDK-8271926	Crash	P3	11, 16	Fixed
JDK-8297730	Diff	P3	9, 11, 17, 18, 19, 20, 21	Fixed

Oracle JDK. We automatically extracted 5,419 templates from 77 open-source Java projects via replacing concrete numbers and expressions in existing programs with holes, e.g., replacing `a + 1` with `arithmetic(intId(), intVal()).eval()`. Using these templates, JATTACK detected 137 failures, while Java* Fuzzer did not detect any bug during the same time frame of one week. JATTACK also achieved a higher coverage of C1 and C2 compilers [11] compared with Java* Fuzzer.

During our experiments, JATTACK in total detected seven bugs in the Oracle JDK JIT compiler, as shown in Table I. The Oracle developers confirmed and fixed all seven bugs. They labeled all the bugs we reported with priority P3 (major loss of function) or higher, including two CVEs: JDK-8239244 showed mismatching outputs on different tiers because C2's range-check elimination led to incorrect loop executions, and JDK-8239244 crashed C1 because an array store in C1-compiled code wrote to an arbitrary location due to index overflow. Both CVEs [12], [13] were fixed in recent Oracle Critical Patch Updates [14], [15].

VI. Related Work

There is a large body of work on compiler testing, systematically reviewed in recent surveys [16], [17]. Grammar-based generation [2], [18]–[20] and mutation-based fuzzing [3], [4], [21]–[26] are two common approaches to obtaining programs as test inputs to compilers. Unlike them, JATTACK was primarily developed to complement hand-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. There is also work on synthesizing programs given initial templates using SAT/SMT solvers [27], combinatorial techniques focusing on variables [28], or code executions [29], [30]. In contrast, JATTACK generates concrete programs for testing a JIT compiler by executing templates and allows richer expressions to be generated for holes.

VII. Conclusion

We presented JATTACK, a framework that enables compiler testing using templates. Using JATTACK, compiler developers use their domain knowledge to write templates in the same language (Java). A template contains a basic program structure and allows specific language constructs represented by holes, e.g., numbers, expressions, etc., to be explored randomly. JATTACK executes templates to randomly fill holes with possible numbers or expressions and generates programs to be used as test inputs to compilers. Using 84 templates created on our own and 5,419 templates extracted from existing Java programs, JATTACK found seven critical (P3 or higher) bugs in the Oracle JDK JIT compiler. Oracle developers confirmed and fixed all seven bugs, five of which were previously unknown, including two unknown CVEs. JATTACK blends developers' intuition via templates with automated testing to detect bugs in compilers.

Acknowledgement

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 and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, CCF-2145774, and CCF-2217696.

References

[1] Oracle Corporation and/or its affiliates. (2022) Regression test harness for the JDK: jtreg. https://openjdk.java.net/jtreg.

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

[3] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Programming Language Design and Implementation*. ACM, 2016, pp. 85–99.

[4] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *International Conference on Software Engineering*. ACM, 2022, pp. 1133–1144.

[5] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template Java programs," in *International Conference on Automated Software Engineering*, 2022, pp. 23:1–23:13.

[6] N. Tillmann and W. Schulte, "Parameterized unit tests," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 2005, pp. 253–262.

[7] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[8] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[9] Python Software Foundation. (2022) pip . pypi. https://pypi.org/project/pip/.

[10] Oracle Corporation and/or its affiliates. (2022) jdk-updates/jdk11u: 405102e26a62 src/hotspot/share/runtime/tieredthresholdpolicy.hpp. https://hg.openjdk.java.net/jdk-updates/jdk11u/file/405102e26a62/src/hotspot/share/runtime/tieredThresholdPolicy.hpp.

[11] ——. (2021) The Java HotSpot performance engine architecture. https://www.oracle.com/java/technologies/whitepaper.html.

[12] The MITRE Corporation. (2022) CVE - CVE-2020-14792. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14792.

[13] ——. (2022) CVE - CVE-2022-21305. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-21305.

[14] Oracle. (2022) Oracle critical patch update advisory - October 2020. https://www.oracle.com/security-alerts/cpuoct2020.html.

[15] ——. (2022) Oracle critical patch update advisory - January 2022. https://www.oracle.com/security-alerts/cpujan2022.html.

[16] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, 2020.

[17] Y. Tang, Z. Ren, W. Kong, and H. Jiang, "Compiler testing: a systematic literature analysis," *Frontiers of Computer Science*, vol. 14, no. 1, p. 1:20, 2020.

[18] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Programming Language Design and Implementation*. ACM, 2011, pp. 283–294.

[19] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for Java JIT compiler test system," in *International Conference on Quality Software*. IEEE, 2003, pp. 20–23.

[20] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2020, pp. 196:1–196:25.

[21] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *International Conference on Software Engineering*. ACM, 2016, pp. 180–190.

[22] Y. Chen, T. Su, and Z. Su, "Deep differential testing of JVM implementations," in *International Conference on Software Engineering*. IEEE, 2019, pp. 1257–1268.

[23] Code Intelligence GmbH. (2021) CodeIntelligenceTesting/-jazzer: Coverage-guided, in-process fuzzing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer.

[24] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos, "Finding typing compiler bugs," in *Programming Language Design and Implementation*. ACM, 2022, pp. 183–198.

[25] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-picking: Differential fuzzing of JavaScript engines," in *Conference on Computer and Communications Security*. ACM, 2022, pp. 351–364.

[26] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler," in *USENIX Security Symposium*. USENIX, 2023, p. to appear.

[27] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5–6, pp. 475–495, 2013.

[28] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Programming Language Design and Implementation*. ACM, 2017, pp. 347–361.

[29] J. Hua and S. Khurshid, "EdSketch: Execution-driven sketching for Java," in *International SPIN Symposium on Model Checking of Software*. ACM, 2017, pp. 162–171.

[30] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "CodeHint: Dynamic and interactive synthesis of code snippets," in *International Conference on Software Engineering*. ACM, 2014, pp. 653–663.