UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

VULPEX: Vulnerable path exploration through dynamic symbolic execution

VULPEX: Erkundung verwundbarer Pfade mittels dynamischer symbolischer Ausführung

Masterarbeit

verfasst am
Institut für IT Sicherheit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Nils Loose

ausgegeben und betreut von
Prof. Dr.-Ing. Thomas Eisenbarth

mit Unterstützung von
M. Sc. Florian Sieck

Lübeck, den 04.11.2022

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Nils Loose

Zusammenfassung

In der heutigen digitalisierten Welt, in der mehr Unternehmen denn je sensible Daten über Anwendungsschnittstellen (APIs) preisgeben, wird es immer schwieriger Sicherheit zu gewährleisten. Eine kürzlich durchgeführte Studie des Open Web Application Security Project (OWASP) hat ergeben, dass 94% der untersuchten Webanwendungen eine Form von Code-Injection-Schwachstelle enthalten. Injektionsangriffe stellen eines der drei größten Sicherheitsrisiken für Webanwendungen dar. Der Einsatz von Penetrationstests zur Bewertung der Sicherheit exponierter Endpunkte ist jedoch kostspielig und zeitaufwändig, weshalb der Bedarf an automatischen Sicherheitsbewertungen steigt. Die symbolische Ausführung ist eine Technik zur systematischen Erkundung des Zustandsraums eines Programms, welche auf der Aufzeichnung symbolischer Constraints von Verzweigungsbedingungen basiert. Jaint, eine symbolische Ausführungsengine verwendet eine spezielle Implementierung der Java Virtual Machine (JVM) und hat kürzlich das Potential von symbolischen Ausführungsengines für das Auffinden von Schwachstellen in Java-basierten Webanwendungen demonstriert. Java war im Jahr 2021 eine der fünf am häufigsten verwendeten Programmiersprachen und wird oft in Webanwendungen von Unternehmen eingesetzt. In dieser Arbeit wird SWAT vorgestellt, welche auf der symbolischen Ausführungsengine CATG aufbaut. SWAT ist die erste instrumentationsbasierte dynamische symbolische Ausführungsengine, die auf die Erkennung von Schwachstellen in Java-basierten Webdiensten zugeschnitten ist. SWAT nutzt eine lose gekoppelte Architektur, basierend auf der symbolischen Ausführungsengine COASTAL. Das System ist für die Anwendung auf Webservices zugeschnitten, indem der zugrundeliegende Webserver als Harness verwendet wird. SWAT bietet Adapter für schnelle und automatische Initialisierung der symbolischen Ausführung von Anwendungen, die mit gängigen Frameworks wie Spring erstellt wurden. In der Code-Injection-Kategorie des OWASP-Benchmarks erreicht SWAT eine Genauigkeit von 0,76 und ein F1-Maß von 0,86. SWAT schneidet somit nur wenig schlechter ab als Jaint, weist allerdings eine Verringerung der Laufzeit um den Faktor 60 auf.

Abstract

Securing today's digitalized world, where more entities than ever expose sensitive data through application programmable interfaces (APIs), is becoming increasingly difficult. A recent study by the open web application security project (OWASP) found 94% of web applications under evaluation to contain some form of injection vulnerability, placing injection attacks among the top three web application security risks. However, using penetration testing to evaluate the security of exposed endpoints is costly and time-consuming, raising the need for automatic security evaluations. Symbolic execution is a language-specific technique to systematically explore the state space of a program by recording symbolic constraints of branching conditions. Jaint, a symbolic execution engine utilizing a custom implementation of the Java Virtual Machine (JVM), has recently shown the potential of symbolic execution engines to find vulnerabilities in Java-based web applications. Java was one of the five most used programming languages in 2021 and is commonly utilized in enterprise-grade web applications. In this thesis, we present SWAT, our *symbolic web application testing* platform, building upon the CATG symbolic execution engine. SWAT is the first instrumentation-based dynamic symbolic execution engine tailored to detecting vulnerabilities in Java-based web services. We implement a loosely coupled architecture similar to the symbolic execution engine COASTAL, using the underlying web server as a harness. We offer adapters to quickly and automatically initialize symbolic execution on targets built with popular frameworks such as Spring. On the injection category of the OWASP benchmark, we achieve a precision of 0.76 and an F1 score of 0.86, scoring slightly lower than Jaint while reducing the runtime by a factor of 60.

Acknowledgements

I want to thank Prof. Thomas Eisenbarth and Florian Sieck for their excellent supervision of this thesis. In addition, I want to thank Felix Mächtle, Yara Schütt, and Lorena Rudolph for fruitful discussions and feedback.

Contents

1

Introduction

The last decade has seen an ever-increasing growth of web application programmable interface (API) deployment and usage among companies and organizations, providing services as web applications using APIs. APIs provide a set of protocols and definitions that allow services to communicate via the provided standards and eases the connection of different software without knowing the specific implementation. APIs are extensively used in web-facing applications to provide communication between different services and components, such as front and backends. As these publicly exposed services often deal with sensitive personal, financial, or medical information, their security is vital. Without proper security evaluation, applications may fall victim to a variety of attacks. Injection attacks (Figure 1.1) are ranked among the top three vulnerabilities present in web applications, with 94% of evaluated applications having some form of injection flaw [53]. However, manual code review and penetration testing are labor-intensive and often incomplete, raising the need for automatic evaluation frameworks like static (taint) analysis [41, 1], fuzzing [4, 3, 45, 18, 22], or symbolic execution [54, 29, 1, 45, 49]. Fuzzing is a fast approach that can quickly evaluate several branches from the target state space. However, without guidance, fuzzing performance tends to stagnate as branching conditions become more specific. Symbolic execution can systematically and formally explore a target's state space through mathematically modeling branching conditions. The model allows symbolic execution to reach deeper parts of the state space and enables more thorough testing. Recent advances in constraint solving build the foundation required to reason about strings as part of the constraints of a symbolic execution engine [26, 6]. String reasoning is a key tool required to explore the state space of web applications effectively, as most interactions between the user and the system are using strings. While many widely adopted programming languages for developing web services exist, this work focuses on Java-based applications. Java [2] was one of the five most used programming languages in 2021 [63] and is used especially in large-scale enterprise software.

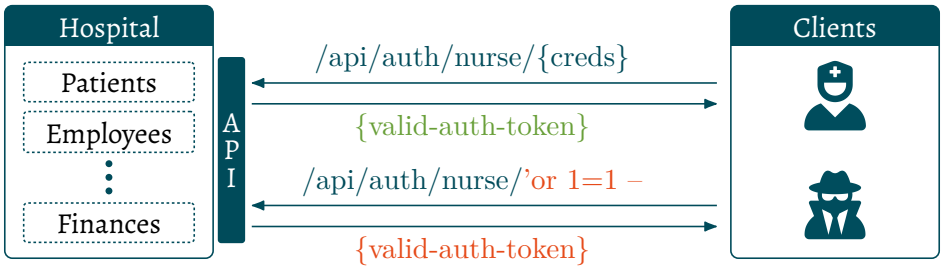


Figure 1.1: Example of the interaction between clients and an API exposed by a critical infrastructure organization. The malicious actor shows a possible SQL injection attack retrieving a valid authentication token without providing credentials.

Web service architectures typically rely on many external frameworks and libraries to handle, for example, the web server or database connections. Targets with external dependencies tailor well to dynamic symbolic execution where the target application is executed, in contrast to static symbolic execution. Values originating from areas the symbolic execution engine does not cover can be used since the concrete value is available. Java-based dynamic symbolic execution engines rely on one of two techniques to drive symbolic execution. Either a custom implementation of the JVM [31, 72] is used to observe the execution [55, 49, 31, 43, 48] or symbolic handling is added through instrumentation [64, 35, 59, 36]. Mues et al. have demonstrated with Jaint that symbolic execution is a capable technique to detect vulnerabilities in Java web applications [49]. However, the reliance on a custom JVM has several drawbacks, such as beeing bound to a specific version. COASTAL has recently demonstrated the potential of a loosely coupled instrumentation-based symbolic execution engine [36]. However, COASTAL relies on an active harness and spawns the target application inside the harness.

In this thesis, we introduce SWAT, an instrumentation-based dynamic symbolic execution engine based on CATG [64]. SWAT is developed to efficiently find vulnerabilities in Java-based web services. During this thesis, we focus on finding injection vulnerabilities. We combine a loosely coupled design with web-specific symbolic harnesses to effectively evaluate web services. We built on the web server's request handling to drive symbolic execution inside the natively running application. In detail, we introduce a new symbolic backend that utilizes the JavaSMT [6] framework to model constraints in the standardized SMT-Lib format [9]. With the new symbolic backend, we gain solver independence, support an additional 55 instructions symbolically, and introduce several new features. New features include symbolic overflow modeling, symbolic modeling of exceptions, and a correct model of the truncated division utilized by the JVM. Through our revised instrumentation core, we support modern language features such as lambda expressions while reducing the instrumentation footprint by 20%. The decoupled symbolic executor has been reworked to allow for parallel symbolic execution in different threads and provides functionality to easily initialize symbolic execution using either two new generic drivers or one of two web-specific drivers. We include web-specific instrumentation drivers for the Spring framework [68] and for javax servlets. Our new symbolic explorer is a standalone application that exposes endpoints to receive execution traces from the symbolic

executor. By using a custom type of execution trace, we are able to successively build an execution tree of multiple symbolic executions inside the symbolic explorer. We evaluate the effectiveness of our symbolic engine by providing results from state-of-the-art verification tools, and SWAT on a subset of the SV-Comp 21 [15] benchmark. By introducing symbolic-based vulnerability detection, we are able to achieve an F1 score of 0.86 at detecting injection vulnerabilities on the OWASP benchmark [51]. To summarize, our main contribution are:

- SWAT, an instrumentation-based symbolic execution engine, based on CATG [64].
- A decoupled symbolic executor and symbolic explorer.
- The first instrumentation-based sybmolic execution engine that is tailored to finding vulnerabilities in Java-based web services.
- An extensive evaluation on a benchmark for software verification and a benchmark for vulnerability detection demonstrating the capabilities of SWAT.

The structure of this thesis is split into six chapters. In Chapter 2, we introduce concepts used throughout the remainder of the thesis, including a brief introduction to symbolic execution, an overview of the Java virtual machine, and the bytecode instruction set. Lastly, Java-based web vulnerabilities are discussed. In Chapter 3, we provide an overview and a discussion of existing symbolic execution engines for Java and give a short digression into fuzzing and state selection heuristics for symbolic execution. We introduce SWAT, our symbolic web application testing platform in Chapter 4. At the end of the chapter, we evaluate the performance of SWAT on SV-Comp [15], a benchmark for evaluating verification tools. In Chapter 5, we discuss functionality specific to identifying vulnerabilities and evaluate the effectiveness of SWAT at finding injection vulnerabilities using the OWASP Benchmark [51]. Lastly, we summarize the evaluations and highlight limitations and future work in Chapter 6.

2

Background

This chapter introduces concepts and techniques that are used in the following chapters. To begin with, a short summary of symbolic execution and satisfiability modulo theory (SMT) is given, followed by an introduction of Java bytecode and the JVM specification. Lastly, the bytecode instrumentation is introduced, and Java-based web vulnerabilities are highlighted.

2.1 Symbolic Execution

Systematically exploring a program's state space is a valuable methodology in various applications ranging from software verification or test generation to bug finding or vulnerability detection. However, finding correct inputs to explore as many branches as possible in as little time as required is not trivial. A commonly used technique is *fuzzing* or fuzz testing. While fuzzing, the application is rapidly tested against various mutated values. Values are either randomly mutated when no context is available or mutated based on some heuristic built on additional information, such as the program's output or coverage information. However, while fuzzers are generally quite fast per test case, the depth they can reach is often relatively shallow because the provided values are not specifically tailored to the branches inside the program. Relying on random mutations often leads to a falloff in newfound branches as the values required to reach new branches increase in specificity. Symbolic execution aims to mitigate the falloff by closely monitoring the application and calculating new values that reach specific branches.

Symbolic execution is a technique to reason how specific values influence the control flow of an application by tracking metadata about the program state. Instead of concretely executing the target application, symbolic execution models program variables as logical expressions. Symbolic expressions represent the symbolic value of variables and their relations. Path constraints add bounds to symbolic expressions enforced by branching points in the target application. The execution state is tracked by maintaining symbolic expressions and path constraints that are explained below. The notations used in this section are adapted from Eisenbarth [66].

```
0  public static void main(String[] args){
1      int x = (int) args[0];
2      int y = (int) args[1];
3      int z = 2 * x + y;
4      if(z < 42){
5          System.out.println("Path 1");
       } else {
6          if(x > y){
7              System.out.println("Path 2");
           } else {
8              System.out.println("Path 3");
           }
       }
   }
```

Figure 2.1: Exemplary Java main function to illustrate symbolic execution. The function reads two integer values from the command line arguments, performs simple arithmetics, and contains two branching possibilities to illustrate symbolic expressions and path constraints. The symbolic state for each step is shown in Figure 2.2.

The metadata tracked using symbolic execution is stored in the *symbolic state*. The tracked data includes the path constraints π, the symbolic expressions store σ, and a variable mapping between symbolic variables and actual variables. Often symbolic execution engines also store the concrete values of variables inside the symbolic state.

Symbolic expressions model the state of the variables inside the application. These expressions are stored in the symbolic expression store σ. When new variables are introduced without a fixed value, the free variable is assigned an α_i value. Typically only variables that, in some form, are under the user's control are marked as free variables whose initial value can be modified. In the example shown in Figure 2.1, lines 1 and 2 initialize two integers with values supplied through the command line; hence these values are under the user's control and are modeled accordingly as:

$$\sigma = \{\phi_1 = \alpha_1, \phi_2 = \alpha_2\}$$

To maintain the relation between symbolic expressions and actual variables, an additional variable mapping is stored where the actual variable maps to its symbolic counterpart: $x \rightarrow \phi_1$ and $y \rightarrow \phi_2$. Each operation that manipulates values has to be modeled symbolically as symbolic expressions. Line 3 in Figure 2.1 introduces a new variable that performs a mathematical operation on the previously defined variables. Since the operation does not introduce branching behavior, the effect is modeled as symbolic expressions ϕ in the symbolic expression store as:

$$\sigma = \{\phi_1 = \alpha_1, \phi_2 = \alpha_2, \phi_3 = 2 \cdot \phi_1 + \phi_2\}$$

with an additional variable mapping $z \rightarrow \phi_3$. The symbolic expressions stored alongside the variable mapping and path constraints for the example at hand are also visualized in

Figure 2.2. Each node represents the corresponding line number from Figure 2.1, and the updated symbolic state is listed along each branch. Each new entry is marked in blue for visibility.

When symbolic execution encounters a branching possibility such as an `if`-clause, the previous symbolic state is duplicated. New *path constraints* are added to both duplicated states modeling the corresponding branching condition. The duplication at each branching possibility results in a state space explosion because the resulting tree can have an exponential number of branches in the depth of the tree. Figure 2.1 has the first branching possibility in line 4, where $z < 42$ is checked. The equation is added to the path constraints of the two forked symbolic states so that for one path constraint, the inequality holds, and for the other, it does not. In the example, the path constraint

$$\pi = \phi_3 < 42$$

is added to the symbolic state modeling the path where the check evaluates to `true`. The negated constraint is added to the associated symbolic state to model the other branch:

$$\begin{aligned}\pi &= \neg(\phi_3 < 42) \\ &= \phi_3 \geq 42\end{aligned}$$

When a second branching condition is met, both need to hold to allow the execution to follow that path. Hence each new branching possibility is added to the existing path constraints using a logical and So for line 6, the path constraints are updated to

$$\pi = \phi_3 \geq 42 \wedge \phi_1 > \phi_2$$

for the path evaluating to `true` and

$$\pi = \phi_3 \geq 42 \wedge \phi_1 \leq \phi_2$$

for the other. The forking behavior is also visualized with the symbolic state in Figure 2.2.

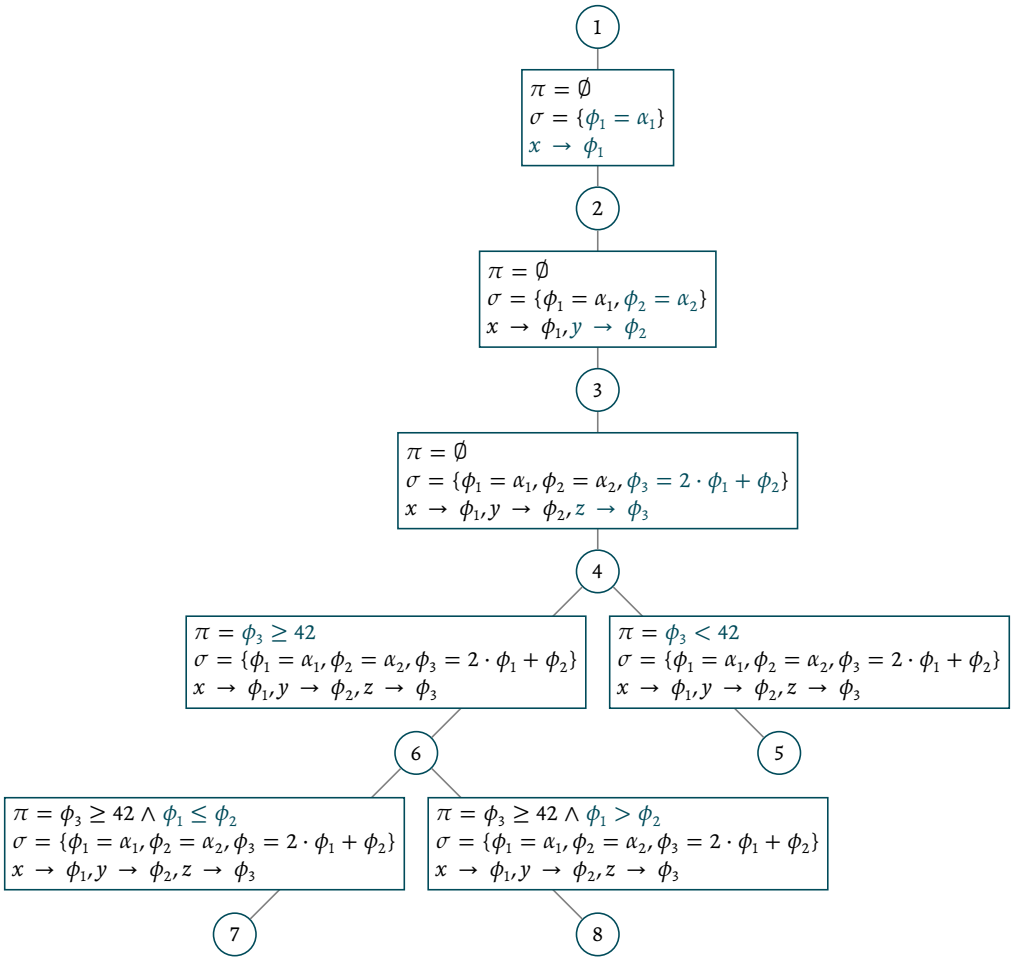


Figure 2.2: Symbolic state for all paths and steps in the exemplary function shown in Figure 2.1. The numbers in the nodes correlate to the line numbers, and along each edge, the current symbolic state is shown, where π stores the path constraints, σ the symbolic expressions and expressions of the form $x \rightarrow \phi_1$ are variable mappings between symbolic variables and the variable names in the source code. The path constraints are updated at each branching point, here in line 4 and line 6, with constraints that model the new bounds on the symbolic expressions required to reach that branch. The symbolic expressions are updated when symbolic variables are altered or introduced, as seen in lines 1, 2, and 3.

After symbolically tracking parts of the program or the entire program, generating inputs for a new path can be achieved by substituting all symbolic expressions in the path constraints. For example, generating inputs for the eights node from Figure 2.2 produces inputs that reach line 8 of the code shown in Figure 2.1 as shown below. The symbolic expression store is given as:

$$\sigma = \{\phi_1 = \alpha_1,\ \phi_2 = \alpha_2,\ \phi_3 = 2 \cdot \phi_1 + \phi_2\}$$

Using the expressions from the expression store, we can substitute all symbolic variables

inside the path constraints to obtain the actual constraint equation:

$$\pi = \phi_3 \geq 42 \wedge \phi_1 > \phi_2 \tag{2.1}$$
$$= 2 \cdot \alpha_1 + \alpha_2 \geq 42 \wedge \alpha_1 > \alpha_2 \tag{2.2}$$

Such equations can be checked for satisfiability, that is, whether a variable assignment exists that satisfies the equations and solved for inputs that satisfy the constraints using solvers such as Z3 [26] as further discussed below. One valid model would be

$$\alpha_1 = 20 \ \wedge \ \alpha_2 = 2$$

Using the variable mapping stored in the symbolic state, we can reach line 8 by executing the program with $x = 20$ and $y = 2$. Solving can be done for any symbolic state; in particular, the state does not have to be directly above a leaf node. When an inner node is selected, the behaviour after the selected node is undetermined. However, solving for states higher up the tree can have various benefits, such as easier solving due to fewer constraints, and is, as discussed below, required for offline symbolic execution where the tree is not fully built. However, not all branches need to be reachable; hence the constraints can also be unsatisfiable, in which case the solver cannot determine new values, and a different branch has to be selected.

There are several different flavors of symbolic execution. Static and dynamic symbolic execution in online and offline fashion are discussed below, with their respective benefits and drawbacks. The traditional variant of symbolic execution is *static symbolic execution* (SSE). When using SSE, the application is interpreted or emulated symbolically by propagating the symbolic state throughout the application instead of being executed. The symbolic execution engine interprets the program in a lifted representation like LLVM bitcode [39] or Java bytecode or on a specific instruction set like x86 to generate the constraints and propagate the symbolic state. By emulating, the target representation is not bound to the actual host architecture of the symbolic execution engine. Emulation allows for more versatility while compromising on performance because the emulation step introduces additional overhead. Using a lifted representation often makes the implementation easier due to a reduced instruction set and allows a broader range of architectures or languages to be tested. When using static symbolic execution, the program state is usually forked for each branching point as shown in Figure 2.2, allowing for parallel exploration of each branch. Parallel exploration is called *online* execution; on the other hand, *offline* execution is when only a single branch is followed and tracked at a time. Online execution is usually faster but requires significant memory, while offline execution is slower but much more memory efficient because only a single state needs to be tracked. One major drawback of static symbolic execution is that when the system under test relies on external libraries or frameworks, or the program is not entirely symbolically tracked, each call to an uninstrumented region has to be mocked. Mocking is not suitable to our scenario because web services typically rely on a large external software stack to provide the web server, database connections, and other functionalities that are very difficult and impractical to mock. Such scenarios cater better to dynamic symbolic execution, as discussed below.

Dynamic symbolic or *concolic execution* is a particular flavor of symbolic execution that has recently gained more attention. Dynamic symbolic execution runs the application with concrete values while maintaining the symbolic state and the path constraints as metadata, so the target program is concretely executed and symbolically tracked. While dynamic online execution is possible, it is typically performed offline, so only the concretely executed branch is symbolically tracked. Offline execution means that the tree shown in Figure 2.2 is not fully built, but a new branch is added with each run. The path constraints of a branching point that has been visited are flipped to explore the previously unexplored branch. Only exploring a single branch per run makes offline execution much more scalable. Additionally, since concrete values are available, only the user-controlled values need to be symbolically tracked, resulting in smaller constraints that are easier to solve. Dynamic symbolic execution also offers the advantage that the engine can work on targets that rely on information from untracked sources like databases or libraries because the concrete values are available in contrast to static symbolic execution. The main downside of dynamic symbolic execution is that the number of reached branches depends on the initial seed values because the engine can only explore one flip at a time from the initial seed. Yet, due to the aforementioned advantages, we focus on offline dynamic symbolic execution.

Offline symbolic execution requires a strategy to select the next branch to explore. Such a strategy is not required in online execution as each branch is explored in parallel. The performance of symbolic execution regarding its use cases, such as bug detection or coverage maximization, is highly dependent on the heuristic used for state selection when the provided time is not enough to explore all branches. Heuristics can select new branches depending on the structure of the tree and leverage additional information from the system under test or from the constraints to aid the selection. While crafting a heuristic that proves effective in maximizing the number of vulnerabilities found is outside the scope of this work, a selection of already existing heuristics is summarized in Section 3.1.

Satisfiability modulo theories (SMT)

Path constraints recorded during symbolic execution are (in)-equalities formulated as a logical formula that encodes whether a particular assignment of variables passes a branching point when executed. To be able to generate new assignments that reach previously unexplored branches, the path constraints need to be evaluated. Checking if a specific branch is reachable equates to whether the constraints have an assignment of variables that fulfill the inequalities, or in other terms, whether the equations are satisfiable or not. If a branch is reachable, new assignments can be generated by finding an instantiation of all free variables within the constraints to satisfy the path constraints under the instantiation. An assignment of variables can then be used to execute and explore previously unexplored branches. Evaluating the satisfiability of a boolean formula is known as the boolean satisfiability problem (SAT). Generally, the SAT problem is NP-complete [23]; however, fast solvers for practical instances exist that go beyond checking if the instance is satisfiable and also provide a model that fulfills the constraints [26, 11, 8]. Given

a propositional logic formula ϕ, SAT is the problem of determining whether an assignment exists that satisfies the given boolean formula. However, arbitrary constraints derived from instructions using symbolic execution are not directly applicable to boolean expressions due to, for example, mathematical operations on integers or floating point values, or the usage of arrays. While these constraints also evaluate to true or false the formula may contain non boolean variables. The satisfiability modulo theories (SMT) are an extension of the SAT problem and generalize the boolean satisfiability problem by introducing predicates over a set of non-boolean variables from predefined theories. Common theories include the integer theory, the floating point theory, or the theory over arrays. Each theory introduces a set of predicates and functions that model the operations allowed in the respective theory. Each predicate $p : X \to \{0,1\}$ is a function with an arbitrary domain X and a binary codomain. Within the domain X, functions with a non-boolean codomain, such as a predicate function $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ for addition, are also allowed. A formula ϕ in first-order logic is considered an SMT instance if all values are either binary values or predicate functions with a binary codomain. Given a formula ϕ and a theory T, the SMT problem is determining whether a model over T exists so that ϕ is satisfiable.

```
   ; Variable declarations
0  (declare-fun x () Int)
1  (declare-fun y () Int)
2  (declare-fun z () Int)

   ; Constraints
3  (assert (= z (+ (* 2 x) y)))
4  (assert (>= z 42))
5  (assert (> x y))

   ; Solve
6  (check-sat)
7  (get-model)
```

Figure 2.3: SMT instance in SMT-LIB format [9] for the path constraint shown in Equation 2.1. The first block defines the variables and assigns a theory, here the integer theory. The second paragraph contains assert statements that need to be satisfied. At last, the given instance should be checked for satisfiability and a model should be produced that satisfies the given constraints.

SMT-LIB [9] offers a formalization of SMT theories and syntax to facilitate a unified language recognized by a variety of popular SMT solvers such as Z3 [26] or CVC5 [8]. The standard enables interchangeability between solvers because they allow the same syntax as inputs. Previously, solvers such as CVC4 [11] had a native input language specific to the solver. Due to recent advancements, SMT-LIB [9] and Z3 [26] support string theories, enabling symbolic execution to utilize native string modeling in the SMT-LIB format. Following the example in Figure 2.1 and trying to solve for the constraint shown in Equation 2.1, the statement in SMT-LIB format is shown in Figure 2.3. Variables are de-

clared within their theory, which can be seen in lines 0 to 2, where three variables from the integer theory are instantiated. The actual path constraints as seen in Equation 2.1 are modelled using assert statements in lines 3 to 5. The solver checks whether these statements are satisfiable (line 6) and try to obtain a model of the previously declared variables that fulfill the statements (line 7). The output of Z3 [26] against the SMT instance is shown in Figure 2.4. Line 0 shows whether the instance is satisfiable or not, followed by a model that shows a satisfying assignment of all free variables. The model produced by the solver can be any instantiation that fulfills the constraints and does not need to be unique. Furthermore, the values provided are not optimized for either minimality or maximality by default; however, modern solvers such as Z3 also provide optimization techniques for model finding. For a more detailed explanation of the format, we refer the interested reader to the SMT-LIB Standard [9].

```
0 sat
1 (model
2   (define-fun y () Int
3     0)
4   (define-fun x () Int
5     21)
6   (define-fun z () Int
7     42)
8 )
```

Figure 2.4: The listing shows the result from the satisfiability check in line 0 and a model for the SMT instance shown in Figure 2.3 in lines 1 to 8 produced by Z3 [26]. A model is an instantiation of all variables with values so that all constraints or assert statements are fulfilled.

2.2 Java Virtual Machine

In this section, we briefly introduce the Java bytecode instruction format and give an overview of Java Virtual Machine (JVM) internals specific to a thread of execution. Lastly, we describe the Java agent that provides capabilities for load-time instrumentation of bytecode.

Java bytecode

Java source code compiles to Java bytecode, an architecture-independent instruction set that is interpreted or executed by the Java Virtual Machine (JVM). To run Java programs, the developer starts the JVM and specifies what class files in bytecode format should be executed by the virtual machine. In contrast, binaries are directly executed on the CPU in languages such as C. Thus the source code has to be compiled for a specific architecture like x86, requiring a compiler for each architecture. In contrast, Java bytecode can run on any architecture with a compatible JVM. The functionality and features required by the

JVM are detailed in the JVM specification (summarized in the next subsection). The JVM generally serves as an abstraction layer between the host architecture and the bytecode, allowing for one instruction set. Depending on the specific implementation of the JVM, the bytecode can either be interpreted inside the virtual machine or compiled to native instructions using a just-in-time (JiT) compiler. As the name suggests, instructions in the bytecode format use a bytecode structured instruction set developed for the JVM.

```
   /**
   * Instance method that
   * performs addition of
   * two integers.
   */
0  int add(int x, int y){
1      return x + y;
2  }
```

(a) Function

```
int addition(int, int);
    Code:
        0: iload_1
        1: iload_2
        2: iadd
        3: ireturn

    LocalVariableTable:
        .. Slot  Name  Signature
        ..  0    this   LClass;
        ..  1     x        I
        ..  2     y        I
```

(b) Bytecode for the function in (a)

Figure 2.5: (a) Source code of an instance method that performs the addition of two integers and returns the result. (b) The compiled method in byte code. The numbers before each instruction are byte offsets and can increase by more than one byte per instruction if additional bytes are used (for example for indices). The two integer values are loaded from the methods locals (seen in the LocalVariableTable) and put onto the stack using the iload instruction. The iadd instruction retrieves the two top values from the stack and puts the result of the addition back onto the stack. The ireturn sets the methods return value to the stack's top value.

The instructions are executed on a stack machine with additional storage possibilities using a register machine. When an instruction is executed, the required number of operands, if any, is popped from the stack, and the resulting value, if any, is pushed back onto the stack. Values on the stack can also be relocated into the register area to be made accessible using an index or reference.

The operation codes (opcodes) are one-byte wide, allowing for a total of 256 instructions, of which 202 are currently in use. In addition to the opcodes, each instruction may use zero or more bytes for indexing or other purposes, such as jump addresses. Most instructions fall into one of seven general categories:

- Load and store instructions can transfer entries between the stack and register. The value can be transferred between the stack and register for arithmetic values. For generic objects, only the reference or address is transferred.
- Arithmetic operation and comparison instructions perform the operations on the

values on top of the stack. For example, an integer addition is performed using the iadd instruction. To execute the iadd instruction, the JVM adds the top two stack values and returns the result to the stack. Some comparison operations, such as comparing if a value is zero, are also part of this group because they push the result as an integer back onto the stack instead of branching directly. Most arithmetic instructions are typed using a prefix such as i for integer operations (i.e. iadd or isub).
- Arithmetic type conversions are directly supported as instructions. The JVM does not differentiate between, for example, booleans, shorts, and integers; hence they are all stored as integers. However, value shortening can be performed using the corresponding instruction such as i2s that shortens the operand into the value range of a short. Nevertheless, the resulting value on the stack is again an integer.
- Stack management instruction can duplicate, remove or rearrange values on the stack. These instructions are type-indifferent but cannot split values that span across multiple entries.
- Object management instructions handle object creation and interactions, such as manipulating class variables. While arrays are also objects, they have dedicated instructions for creation and manipulation that are also typed.
- Method handling instructions are used for method invocation and termination. Calling these instructions invokes special handling to remove or add additional stack frames.
- (Conditional) control flow handling covers all instructions that can diverge the control flow. That group includes unconditional jumps such as goto alongside conditional jumps based on the top value on the operand stack like ifeq.

A few other instructions exist that handle more specialized tasks, such as thread synchronization and exception handling.

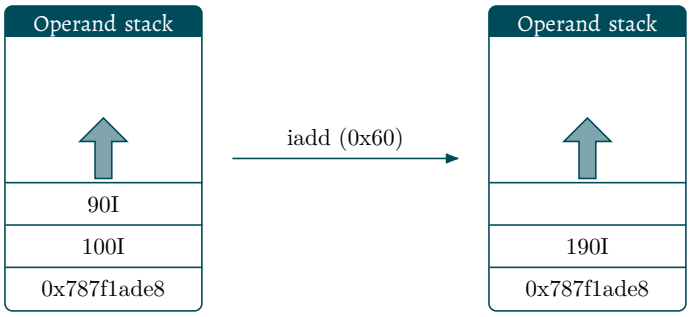


Figure 2.6: Java operand stack before and after an iadd instruction. This example visualizes the iadd instruction in line 2 from the code block in Figure 2.5b. The iadd instruction performs addition on the top two (integer) values on the stack and pushes the result back onto the stack. The first entry on the stack is a reference into the heap where the parent class object is stored. The second and third entries are the actual values. They can be stored directly in the operand stack because integers are stored in 32-bit two's complement, and the width of the operand stack is 32-bit. After addition, the stack size remains unchanged because the operand stack of a method has a fixed size determined at compile time.

An example of a method that performs the addition of two integers is shown in Figure 2.5. The method parameters are placed in the locals register on method invocation. For instance-methods, invoked on an object, the first local slot is reserved for the this-pointer, a reference to the object itself. In the bytecode shown in Figure 2.5b the two integers from the method's parameters are loaded from the locals using the iload_1 and iload_2 instructions respectively. As indicated by the prefix, they load an integer from the locals at positions 1 and 2, and put the value onto the stack. Further information on the values in the locals is also shown in the LocalVariableTable in Figure 2.5b. The next instruction (iadd) removes two values from the stack and puts the value resulting from adding the two values back onto the stack. Both values must be of type int. The type is not checked during runtime but validated during compilation. For the iadd instruction, the operand stack before and after the execution is visualized in Figure 2.6. The stack grows from the bottom up, as indicated by the arrow. The maximum size of the operand stack is fixed and determined during compilation. The bottom entry on the stack is a 32-bit address that references the parent class object in the heap area that is not visualized here. The two integers are pushed onto the stack using the aforementioned iload instructions. The specific values shown in Figure 2.6 are only for demonstrative purposes and would depend on the parameters passed to the method on invocation. While both integers are visualized in decimal format, they are stored as a 32-bit two's complement binary number on the stack. The iadd instruction removes the two topmost values from the stack and performs integer addition. After performing addition, the result is put back onto the stack. Lastly, the ireturn instruction shown in Figure 2.5 takes the top value from the stack and puts it onto the stack of the invoking method (not shown here).

Java virtual machine specification

Java uses a virtual machine, the Java virtual machine (JVM), to execute Java bytecode. Utilizing a virtual machine as an abstraction layer above the host architecture allows for architecture-independent source code compilation. Hence the same bytecode can be run on arbitrary architectures as long as a JVM exists for that architecture. The JVM specification [40] specifies the behavior required by an implementation of the JVM specification. The freedom provided by the specification allows for various JVM implementations for specific purposes, like performance-optimized implementations or implementations that focus on bytecode verification by monitoring the runtime and allowing modifications like the JVM implementation utilized by JavaPathfinder [31]. Each JVM generally has a shared heap that stores class instances and arrays. Because the heap is not directly exposed to programmers but managed by the JVM, garbage collection can be handled directly by the JVM. The JVM also has a shared memory area that stores JVM internals and the method area. The method area stores per-class information, including field and method references and constants.

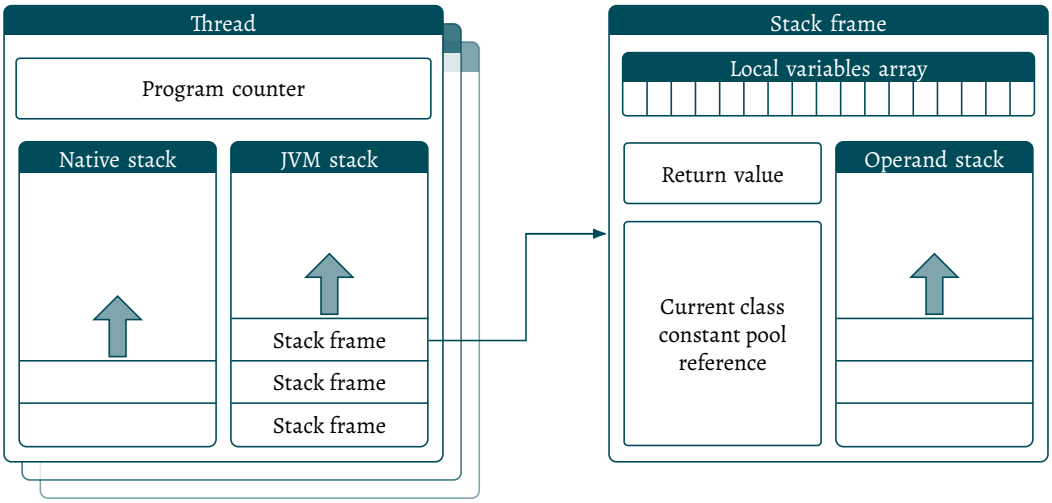


Figure 2.7: JVM internals that are specific to each thread of execution. Each thread has a program counter pointing to the currently executed instruction. The native stack contains method information needed to execute native functions. Depending on the JVM implementation, the native stack could be identical to the C stack. The JVM stack contains stack frames. A method-specific stack frame is created on method invocation and deleted on method termination. Each stack frame holds the return value of the corresponding method alongside an operand stack and a local variable array for storing primitive values and references. Each stack frame also maps into the heap to access the constant time pool.

The JVM specification allows for concurrency in threads of execution. Depending on the JVM implementation, these threads can translate to CPU threads but can also be executed in a single thread. Each thread has private memory to allow concurrency, as shown in Figure 2.7. The thread-specific area includes the program counter that points to the currently executed instruction when executing Java code or is undefined when native code is executed. Each thread of execution also has a native stack used to invoke native methods. The actual design of the native stack is implementation dependent but can, for example, be identical to the C stack. Because the JVM is a stack and register machine, instruction operands are not directly accessed in memory but by using a stack. A stack frame is created and put onto the JVM stack to model method-specific information upon method invocation. After method termination, the stack frame is cleared from the stack. Because the JVM uses a stack machine combined with a register machine, each stack frame contains an operand stack that stores instruction operands. This stack is augmented with a local variable array for storing operands or references. Because the stack and local's size are determined at compile-time, dynamic objects cannot be stored there and are put into the shared method area. Only a reference to the object is stored in the stack frame.

Bytecode Instrumentation

Additional instructions must be added to the target application to perform instrumentation-based dynamic symbolic execution. Instrumentation of bytecode can be done offline before the application is started or online during the runtime of the JVM. To perform online instrumentation, the JVM offers functionality to attach a specially crafted jar-file, called a Java agent, to the JVM that utilizes the instrumentation API exposed by the JVM to manipulate bytecode during the loading process. When initializing a JVM, the Java agent can be attached using either a dynamic loading API or statically through a command line argument. When a class file is loaded into the JVM by the class loaders before execution, the attached Java agent can intercept the loaded bytecode before storing it in memory. The interception allows for altering existing instructions and adding or removing instructions, methods, or even classes. An example that invokes a native process is shown in Figure 2.8. After compilation, the method call is replaced by the invokevirtual instruction that fetches the method using an index from the class's constant pool. The parameter is loaded from the method locals through the aload instruction. Using a transformer inside the Java agent, the method call can be manipulated. For example, the parameters of the method could be duplicated and logged to another method.

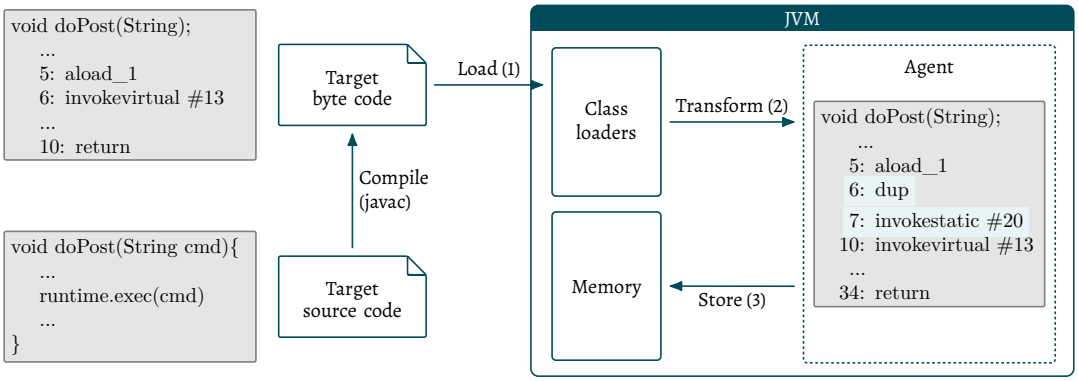


Figure 2.8: JVM and instrumentation agent interaction at load time of target files. The exemplary method shows the execution of a native process in source code and compiled byte code. The Java agent is initialized if attached on JVM startup and registers a class file transformer. When compiled, target applications are loaded (1), and the class-loader hands the class file definition to the Java agent for byte code transformation (2). During transformation, the agent can manipulate instructions and alter the class definition. In this example, the agent added a method call to a static method that receives the same parameter as the original method. After redefinition, the class is stored in memory (3) for usage by the target application.

The Java agent receives the instructions of a class as a byte stream which can be manipulated. To instrument the byte stream, we use the ASM [20] framework to perform bytecode transformations. While other libraries like Apache's bytecode engineering library (BCEL) [25] exist, ASM's architecture is performance oriented, and has a better memory footprint. Performing transformations on the bytecode level offers several advantages; firstly, it requires no access to source code, making it usable on already compiled

or closed source programs. Secondly, using ASM, we can programmatically inspect all code loaded into the runtime and perform custom transformations to the bytecode to add the required handling for symbolic execution.

2.3 Java Vulnerabilities

The OWASP Foundation publishes a yearly report outlining the top ten types of vulnerabilities present in web applications [53]. The top three types of vulnerabilities from 2021 include broken access control, cryptographic failures, and injection attacks. This thesis focuses on injection vulnerabilities caused by vulnerable Java code. The report outlines that 94% of evaluated applications contain some injection vulnerability. The injection category summarizes 33 Common Weakness Enumerations (CWE's) [47] including CWE-79: Cross-site Scripting and CWE-89: SQL Injections. In general, injection attacks describe vulnerabilities where user-controlled values are passed through an application to some interpreter without proper validation or sanitization. For injection attacks, the attack vector usually contains two major parts: the source, where a user-supplied value is read into the application context, and a sink, where the value is interpreted after it has been processed. Typical sources in the web context are, for example, values extracted from HTTP requests like cookies, headers, or parameters. Typical sinks include calls to database systems or command executions, where a command or statement is passed as an argument.

```
0 @Override
1 public void doPost(HttpServletRequest request, HttpServletResponse
      response){
      ...
2     String param = "";
3     if (request.getHeader("BenchmarkTest00008") != null) {
          // Source
4         param = request.getHeader("BenchmarkTest00008");
5     }
      ...
6     String sql = "INSERT INTO users (username, password) VALUES ('foo','"
          + param + "')";
      ...
7     Statement statement = DatabaseHelper.getSqlStatement();
      // Sink
8     int count = statement.executeUpdate(sql);
      ...
9 }
```

Figure 2.9: SQL Injection example from OWASP benchmark test suite [51]. The function is taken from the BenchmarkTest00008 servlet and is called when a user sends a POST request. The user-controlled header value is directly passed into an SQL update function without using prepared statements or sanitization.

While the evaluation by the OWASP foundation regards any web-facing applications, Java-based web services align well with the types of vulnerabilities presented in the report. Figure 2.9 shows an abbreviated example of an SQL-injection present in Java code. The example is adapted from a benchmarking dataset maintained by OWASP [51]. The dataset is explained in more detail and used for evaluation in Section 5.2. Line 4, shown in Figure 2.9, is the source of the vulnerability, where an HTTP header value is read into the application context. In line 6, the value is processed into an SQL query without any sanitization. The query is directly passed into the sink in line 8, where the query is handed to a database driver. The code is part of a method that handles POST requests of a web service. While the sink shown here communicates with an SQL database, it could also be a variety of other functionalities. The sink does not need to interpret the injected statement directly but can also pass it to external services. For example, when a user can store client-side scripts like Javascript into a database used to render a web page on a victim's machine, it is considered a cross-site scripting attack (XSS) falling in the injection attack category. The interplay between the target application and external services or client devices makes detecting whether an attack was successful difficult. Especially when only considering information from within the application context. However, one can detect whether malicious information can reach a vulnerable sink.

3

Related Work

This section summarizes work related to this thesis; at first, existing symbolic execution engines for Java are introduced, covering both interpretation and instrumentation-based approaches. Secondly, work that focuses on guiding symbolic execution, more precisely, the state selection towards a specific target metric, such as maximizing the number of found vulnerabilities, is explored. Lastly, a slight digression into fuzzing web services, mainly the REST-based fuzzer RESTler [4], is given.

3.1 Symbolic Execution Engines

The concept of dynamic symbolic execution, introduced at the beginning of Chapter 2, is a widely used technique for code analysis. It is applicable in a variety of fields from software verification [55] to test generation [59], bug finding [32] or vulnerability detection [49]. After a general overview of symbolic execution engines and techniques for various languages, this section summarizes existing implementations of symbolic execution engines for Java and highlights concepts and problems arising from different implementation techniques. These systems can be divided into interpretation-based approaches using a specifically designed JVM [31, 72], which are introduced first, and instrumentation-based approaches that utilize the ASM bytecode manipulation framework [20] to add symbolic handling into the target code.

Symbolic execution of binaries extensively researched. Interpretation-based approaches [69, 21] usually lift the binary to an intermediate representation such as LLVM [39]. Lifting reduces the number of instructions and removes architecture-specific handling. The benefit of lifting is more general applicability and ease of implementation at the cost of performance. Compilation or instrumentation-based approaches [73, 57] add symbolic handling to the binary and directly execute the target. These approaches are either combined with an entire system emulator like QEMU [12] to allow architecture independence or implemented to run on one specific architecture. Using instrumentation frameworks like Intel Pin [44], symbolic handling can be added directly into the binary. Alternatively, the binary can be lifted to an intermediate representation, where the handling is added and recompiled.

Actual execution allows for faster runtimes and combines well with dynamic symbolic execution.

The above approaches work on binaries, but as we target Java-based applications, these engines are not applicable in our context. This work focuses on bytecode and, due to the web context where runtime information from uninstrumented code regions and external libraries is vital, on dynamic symbolic execution. The JVM acts as an abstraction layer between the host architecture and the compiled Java bytecode when executing a Java program. Generally, two approaches for implementing dynamic symbolic execution for JVM-based languages exist. Firstly, interpretation-based systems [54, 43, 48] use a custom implementation of the JVM specification [40] to add additional functionality into the virtual machine that enables symbolic execution. By employing a custom JVM, the monitoring possibilities are extensive as one can observe, pause or alter the virtual machine's state at the cost of requiring architecture-specific implementations and being limited to a specific version of Java. Secondly, instrumentation-based approaches modify the bytecode of the target application to add handling for symbolic execution into the application. Thus, they can run on an arbitrary JVM without requiring modification to the virtual machine. However, relying on instrumentation is not as extensive because the JVM's internal components, such as the stack frames, cannot be directly observed and must be modeled. However, instrumentation-based approaches can run on any architecture that has a JVM. Instrumentation frameworks for Java bytecode [17], like ASM [20], can be utilized to add symbolic handling to the compiled target.

JavaPathfinder (JPF) [31] was initially developed at NASA[1] as a translator between Java programs and Promela models for model checking. Today JavaPathfinder is an entire suite of tools at the core of which an implementation of the JVM Specification specifically tailored to monitor the system under test lies. The JPF-JVM is used by several engines to implement symbolic execution [55, 43, 49, 45]. JPF can still perform model checking but also entails a suite of other functionalities, such as dynamic symbolic execution using the Symbolic Pathfinder (SPF) extension [55]. Symbolic Pathfinder builds on the JPF-JVM to observe the internal state of the runtime and drive the symbolic execution. It replaces the concrete execution semantics of the JPF-core with symbolic execution logic. A significant benefit of SPF is its possibility to analyze interleaved multi-threaded systems alongside the possibility of backtracking the execution. While the JPF-JVM offers powerful features, its implementation of the JVM specification binds the system to one version of Java. Additionally, any symbolic execution engine built on top of the JPF-JVM is limited by the VMs performance.

JDart [43], a dynamic symbolic execution framework for Java, is built on top of Java PathFinder [31] and uses its custom JVM. It is designed as a robust engine to handle industrial-scale applications and complex mathematical NASA software. The system is split into two major components; the executor executes the system under test by utilizing the information and interfaces provided by Java Pathfinder [31]. The system tracks method parame-

[1] https://software.nasa.gov/software/ARC-17487-1

ters symbolically and allows the user to supply additional bounds on the input space of the variables to restrict the state space. Symbolic handling is implemented for various complex data types, including bit operations, floating-point arithmetic, and non-linear arithmetic. The explorer builds a constraint tree from the supplied symbolic traces and utilizes the JConstraints library [33], developed as part of JDart, to interface a variety of solvers, including CVC4 [11], and Z3 [26]. Some mathematical operations are approximated to allow for faster SMT constraint solving. However, approximating constraints can lead to invalid models that the explorer validates using the new concrete values against the constraint tree before re-executing the system under test. While JDart is a very mature system, its usage of the custom JVM results in performance losses. However, the system won the SV-COMP 2022 [13]. SV-Comp is a benchmark for comparing software verification tools on which a yearly competition is hosted at TACAS. It is the first symbolic executing engine to beat classical model-checking techniques, highlighting the potential of symbolic execution. While the system's performance was fast on the benchmarked systems, these do not include larger systems such as web services requiring additional components and systems to be operational. Besides no support for symbolic string tracking in the original version, its dependency on the JPF limits the system's applicability.

Jaint [49], a combination of dynamic symbolic execution and dynamic multi-colored taint analysis for Java, is a framework to automatically detect vulnerabilities in Java programs given a user-specified definition of vulnerabilities. Using a domain-specific language, the user can specify different sources, targets, and sanitization methods. The framework utilizes JDart [43] as the dynamic symbolic execution engine; the dependency on the JPF-JVM limits the framework's applicability to Java programs that are compiled to run in the specific JVM used by SymbolicPathfinder [54], and entails the same drawbacks previously discussed. The authors evaluated their framework on the OWASP benchmark [51] and achieved a 100% true-positive rate with no false negatives. Mues et al. achieved vulnerability detection by allowing user-defined sink, source, and sanitization specifications using an expressive interface description language utilized by the taint engine to track whether values propagate from a source to a sink. We use a similar approach to detect source-to-sink behavior (Chapter 5). While the results seem promising, no artifacts are publicly available to verify the results. Furthermore, the system entails a thirty-fold increase in runtime compared to FindSecBugs [56], a static vulnerability scanner, on the OWASP Benchmark evaluation suite [51]. Due to the underlying symbolic engine using JPF [31], the performance issues will likely increase for larger industrial-scale applications. Additionally, by the design of the underlying system, the methods that are symbolically evaluated are executed inside a symbolic harness. This implies that the web server and architecture underneath the endpoints are not utilized, but a custom driver is employed. Hence, external systems such as databases are not running and require custom symbolic peers that mock the behavior. We aim to mitigate these issues by using the underlying web server as our harness and providing symbolic execution capabilities inside the native execution of the target application. Using the web server to drive symbolic execution removes the need for a symbolic harness and peers as all components fully function.

GDart [48], a recently proposed dynamic symbolic execution framework, is built on top of the GraalVM [72]. The GraalVM is a robust and high-performance Java virtual machine maintained by Oracle and has a similar feature set to the JPF-VM [31]. GDart already features a modular design comparable to Coastal [36] that is split into three major components. SPouT, their concolic driver, is implemented using the Java on Truffle or Espresso framework [50]. The SPouT framework is a Java bytecode interpreter built as a secondary VM layer on top of the GraalVM. The concolic driver can be seeded with concrete values to guide the execution towards certain states. During execution, a symbolic execution trace is recorded in the SMT-Lib format. The symbolic exploration module handles symbolic traces, builds a constraint tree, and holds the strategies used for symbolic exploration. This framework is a continuation of the JDart [43] engine and uses its constraint framework to construct an SMT problem that is solved to obtain new concrete values. This architecture is a promising design and achieved fourth place at the SV-Comp 2022 [13]. We use the SV-Comp benchmark for evaluation in Section 4.6. However, its dependency on the GraalVM requires a startup of a new JVM instance per explored path. While this is a constant overhead for smaller projects, restarting the web application per request could introduce significant performance penalties in the web context. Furthermore, because the framework is a relatively immature system, its symbolic capabilities are currently limited. String tracing, for example, is not supported in the latest release of the framework. However, using the GraalVM to drive symbolic execution seems promising, considering the number of languages the GraalVM supports and the performance it offers.

JSEfuzz [45] is a vulnerability detection framework that focuses on finding vulnerabilities in Java web services. The framework combines fuzzing and dynamic symbolic execution. JSEfuzz is developed using a modular design that uses multiple worker nodes for fuzzing and one controller node for driving the dynamic symbolic execution. The application and its dependencies are split into its core modules. Fuzzing drivers for each module are derived from unit tests for the respective functions. However, the drivers must be manually written if no tests are present. Each module is independently fuzzed by a worker node using Kelinci [38], a wrapper for the AFL fuzzer [46] that enables AFL to fuzz Java applications. If an exception occurs during fuzzing, the responsible method is identified using the stack trace and marked as vulnerable. A controller node identifies all call-sites of the vulnerable function using source code level search. Given the call sites, the controller node utilizes Symbolic Pathfinder [55] to obtain path constraints that lead to calls of vulnerable functions. Using either the CVC4 [11] or Z3 [26] solver frameworks, concrete inputs are generated to validate the exploitability of the vulnerabilities identified by the fuzzing procedure. While this framework is already tailored towards automated vulnerability detection in web services, it depends on the presence of unit tests for the entire code base, including dependencies. Furthermore, the vulnerabilities considered by the authors are limited to exceptions thrown during fuzzing. They argue that these exceptions could be used to trigger Denial of Service (DoS) attacks against a web service. However, other more sophisticated attacks that do not lead to exceptions, such as injection attacks or de-serialization attacks, cannot be detected using this framework. Furthermore, the dynamic symbolic execution framework is based on Java Pathfinder [31], and

uses its custom Java virtual machine that introduces a timing overhead and limits the applicability to smaller web services because of the aforementioned issues. The authors have not provided artifacts or repositories to verify or improve their design. While their system features a modular design, it divides the target application into smaller modules. However, the modularity makes a system-wide test using external fuzzers like RESTler [4] infeasible.

Considroid [27], a vulnerability detection framework for mobile applications written in Java, utilizes taint analysis and dynamic symbolic execution to detect SQL injections. They utilize the Symbolic Pathfinder engine [54] for concolic execution and combine it with a multi-colored taint analysis. However, they only use the concolic execution to execute specific paths found by static analysis. Using the static analysis, they construct a stack containing what branch has to be taken to reach a possibly vulnerable sink. Most of the previously described systems rely on the JVM supplied by the JavaPathfinder suite [31] and entail the benefits and limitations of that system. While the recent work by Mues et al. [48] utilizing the performance-optimized GraalVM [72] is promising, its current design requires a restart of the JVM for each test run and symbolic strings are not yet supported. Hence we explore a second avenue enabling symbolic execution through instrumentation-based systems, which are summarized below. Instrumentation-based systems can either work directly on the instruction set or on an intermediate representation (IR) like Jimple [67]. Performing transformations on intermediate representations by lifting the bytecode simplifies development due to a more miniature representation. Simplification comes at the cost of either interpreting the intermediate representation or recompiling the transformed code into its target instruction set. The current state of the art does not utilize an IR but works directly on the instruction set described below.

Tanno et al. [64] constructed a tool suite for automated test generation of Java web applications. Given an input design model constituting information regarding screen elements such as input fields and their constraints, business logic definitions, and database table definitions, TesMa generates a set of test cases for the application under test. TesMa relies on CATG, a dynamic symbolic execution engine for Java applications developed by Tanno et al. [64] to generate new inputs and an initial database state. CATG is an open-source dynamic symbolic execution engine that relies on instrumentation to drive symbolic execution. The system has two possible variants, the online mode, and the offline mode. Both modes will be introduced briefly; however, as the system developed in this thesis is based on the CATG engine, more in-depth discussions of the modules and functionalities of the system can be found in Chapter 4. In the online scenario, an overview of the architecture is given in Figure 3.1. The system relies on the Java agent introduced in Section 2.2 to perform symbolic execution and can be split into three core components. Firstly, instrumentation of the target application is performed using ASM [20], introducing calls to a custom logger to log each visited instruction and its values. After transformation, the system under tests runtime can be observed through the function calls added during instrumentation. In the online scenario, a direct concolic execution is performed in the second core module that maintains a shadow stack and heap to build path constraints as the execution progresses. Lastly, after the symbolic execution fin-

ishes, the third module selects a new branch to explore, builds the path constraints, and writes them into a file. The constraint building is implemented using a custom logic and builds constraints in the CVC4 [11] native input format, binding the system to the CVC4 solver.

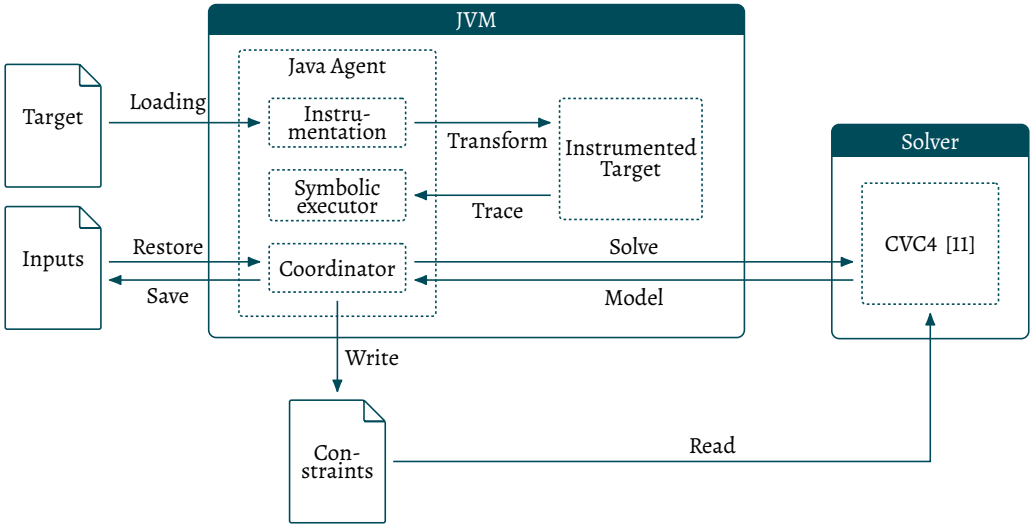


Figure 3.1: Architecture overview for the CATG symbolic engine [64] in online mode. CATG is an instrumentation-based dynamic symbolic execution engine. All functionality except for the SMT solver is part of the Java agent. The agent includes modules for instrumentation, the symbolic driver and a coordinator. The instrumentation agent intercepts loaded classes and adds functionality to observe the executed byte codes. The symbolic executor models the shadow state and builds constraints. Constraints are stored in a file, and the CVC4 [11] solver is called using a command invocation. The constraints are written in CVC4's native input language. New inputs returned from the solver are stored on file to be read when the JVM is restarted.

CATG also offers an offline mode that decouples the symbolic executor and the coordinator modules from the execution of the target application by saving the trace generated during execution to a file. The trace file can then be read by another Java program that holds the symbolic driver and state selector. Effectively the symbolic driver reinterprets the trace and performs symbolic execution decoupled from the concrete execution. Decoupling between instrumentation and symbolic execution allows for separation between concrete and symbolic execution. However, the size of the trace that needs to be written to file can proliferate with more extensive programs because each instruction needs to be logged, limiting the performance. Furthermore, by simply logging each instruction sequentially when it occurs, programs that rely on multiple threads will appear interleaved in the symbolic execution, making symbolic tracking impossible. Furthermore, despite the separation, CATG still needs to execute the target symbolically or fully each time a new input should be generated. As further outlined in Chapter 4, we aim to mitigate some of the above issues by separating the symbolic engine between the symbolic executor and the coordinator. The system is open source, but has not been maintained in recent years.

Islam and Csallner [35] introduced another instrumentation-based dynamic symbolic execution engine that utilizes ASM [20] to instrument the target application. Their system enables symbolic execution for code that relies on interface abstraction. More specifically, Islam and Csallner show a technique to build mock classes of interface instances to enable symbolic execution when the superclass is not available. The system is geared towards test case generation during development and does not align with the focus of this thesis. Even though their engine is published, it is outdated and has several documented issues that we mitigate in our system. Issues include only partial support for symbolic arrays and floating-point arithmetic. The system also does not allow for multi-threading, and string reasoning is not supported.

An alternative dynamic symbolic execution engine for Java is EvoSuiteDSE [59], which is based on EvoSuite [28], an automatic test case generation tool for Java classes. While EvoSuiteDSE works similarly by supplying an instrumentation agent to the runtime that adds symbolic handling to the target bytecode, it is more geared towards test generation in combination with EvoSuite and not a standalone symbolic execution engine.

COASTAL [36] is a more recent instrumentation-based dynamic symbolic execution engine that also relies on ASM [20] to instrument the system under test. Coastal is the first instrumentation-based system with a loosely coupled design between the symbolic driver and symbolic explorer. By allowing the symbolic explorer, a Java program, to load the target application's class files and instrument them with a loader that utilizes ASM it achieves the loosely coupled design. After a target is loaded, a trace is generated by spawning a new thread that performs the target execution independently of the main component. Hence Coastal does not use the Java agent to attach symbolic handling to the target application, but it spawns the target application inside a symbolic harness. While their design shows potential for independent targets, for the use case of this thesis it is not applicable. We focus on targets that are larger web service architectures that rely on libraries to handle the server communication and other components, and starting them multiple times in parallel for each trace is not possible and startup times would significantly hinder the performance. In Chapter 4 we aim to combine the modular design of COASTAL with the symbolic architecture of CATG [64] to achieve independence between the symbolic executor and explorer in a way that the symbolic executor is driven by incoming web requests and not directly through the symbolic explorer.

Guided symbolic execution using machine learning

When symbolic execution is used to evaluate large-scale applications, it quickly becomes unfeasible to explore every possible branch because the number of possible branches is exponential in the depth of the execution tree, also known as the path explosion problem [7]. Each time a branching possibility is encountered, two new branches emerge, leading to exponential growth. Hence, when symbolic execution is used to detect vulnerabilities or bugs in a program where not all branches can be explored, the system's performance directly correlates to the choices of the state selection heuristic. If a heuristic

effectively steers the execution towards vulnerable states, the performance of the system increases. Crafting a heuristic that effectively steers symbolic execution on Java bytecode towards injection vulnerabilities is beyond the scope of this work, but is planned as future research, and some existing work is summarized below. While well-crafted manual heuristics have shown success in improving the performance of symbolic execution engines, they fail to encapsulate the global goal (i.e. maximizing coverage) and only represent certain flavors. These flavors can lead to an exploration engine getting stuck in parts of the code favored by the current heuristic. Using a heuristic approximated by a regression network can lead to a more nuanced decision that encapsulates a larger picture.

Learch [32], an extension to the KLEE engine [21], considers using the heuristics crafted for coverage-optimized search as features for a neural network. These heuristics enable a regression model to approximate the reward of a given state. Because Learch utilizes a feed-forward neural network, the training requires a labeled dataset containing the features representing a state and its actual reward (coverage per time). The reward cannot be calculated online when selecting a state because its successors and coverage are unknown. Given a list of actual tests and their reached states, the authors utilize test trees, a novel approach for efficiently calculating the rewards of a given test. However, test trees are order-dependent because a test case inserted into the tree is only assigned coverage for states that previous test cases have not covered. For training the model, they utilize an iterative approach, where for the first iteration KLEE is running with manual heuristics for test case generation. After the first model is trained, it is used as the heuristic for executing the same target programs symbolically again. The training is done iteratively for several rounds. During inference, instead of using the model obtained in the last training iteration, all learned models are used as a combined heuristic for selecting the next state. While the authors evaluated the model's performance regarding both achieved coverage and reached UBSan security violations [42], the model is only trained for maximizing coverage. Still, one can observe an increase in both achieved coverage and found vulnerabilities, but the increase in discovered vulnerabilities likely correlates with higher coverage.

SyML [60] also tries to mitigate the path explosion problem by prioritizing paths based on an approximated reward using a machine learning algorithm. In contrast to Learch [32], SyML's target function directly considers vulnerabilities instead of coverage. They utilize both the execution history and a forward-moving window for feature extraction. The model is trained on crashing inputs to learn underlying patterns in vulnerable paths. While their approach is promising, the found vulnerabilities are exclusively low-level, like overflows, out-of-bounds accesses, or dereferences.

Another approach uses guided symbolic execution in combination with the Q-learning algorithm [70] to guide the symbolic execution towards a predefined state in the target program [71]. The approach is instantiated in KLEE and shows promising results for combining reinforcement learning and symbolic execution.

3.2 Web Service Fuzzing

When exploring the state space of a program, an orthogonal approach to symbolic execution is fuzzing. In its basic form, fuzzing controls the inputs to the system under test to explore the application with little to no knowledge of the internal architecture of the target. Fuzzing utilizes mutators to vary the program inputs in various ways to reach unintended and new code regions. Because fuzzing does not alter the binary, it allows for a much higher execution speed than symbolic engines. Without additional knowledge, fuzzing mutates the inputs randomly. Often context-unspecific mutations lead to a quick drop off in newfound branches as more difficult path constraints cannot be overcome.

Fuzzers primarily benefit from fast execution times, being able to test many inputs in a short period. These characteristics also allow for a good combination of instrumentation, taint tracking, or dynamic symbolic execution. By combining symbolic execution and fuzzing in a hybrid approach, one can benefit from the fast exploration speed of the fuzzer and augment the mutators with inputs obtained through symbolic execution when the fuzzer's performance stagnates. Integrating the fuzzer described below into the symbolic execution engine developed as part of this thesis is ongoing work at the Institute for IT-Security by Florian Sieck.

RESTler [4] is a black box REST API fuzzer that is grammar-based and uses the OpenAPI specification [65] for grammar generation. The OpenAPI Specification defines a formal interface description for HTTP APIs. It allows programmatic discovery of the endpoints exposed by the service without access to other documentation or source code. Atlidakis et al. developed a parser that automatically generates a grammar from an OpenAPI specification [4]. The grammar effectively provides the structure required for the test harness to fuzz a web service in a fully programmatic manner. As mentioned, fuzzers often only reach a shallow depth because no information about the target program is used. Pythia [3] is an extension of RESTler that instruments the target program to obtain coverage information for guiding the fuzzer, achieving higher coverage rates than native RESTler. Pythia highlights the potential of a hybrid system that steers the fuzzer.

4

Symbolic Web Application Testing

Automatically and systematically identifying vulnerabilities in web services plays a crucial role in securing today's online world. Static application security testing (SAST) and dynamic application security testing (DAST) methodologies are commonly used to detect security vulnerabilities that make an application susceptible to attacks. SAST techniques statically analyze an application's source code to find issues and are often less expensive to run compared to DAST scanners. However, runtime issues and problems arising from calls to external libraries are typically not detectable due to their dynamic nature. DAST tools, on the other hand, rely on testing the application dynamically without knowledge of the internal structure of the application. DAST techniques rely on drivers that test the application from the outside and can only react to observable changes, such as responses or timing behavior. Dynamic testing allows tools to evaluate the dependencies between different components at runtime. However, being a black box scenario, the depth at which DAST tools can evaluate a target is limited, comparable to pure fuzzing. Interactive Application Security Testing (IAST) techniques aim to mitigate the shortcomings of both SAST and DAST approaches. By dynamically integrating an agent into the application, that can guide a dynamic external testing component with the knowledge gained by analyzing the internal structure of the system under test.

Using symbolic execution as an IAST technique to find web vulnerabilities in Java-based web services has recently been shown to be effective by Mues et al. [49]. However, their system comes with several significant limitations. Firstly, it builds upon a custom driver for the methods considered entry points to enable symbolic execution through the JPF-JVM [31]. Such a harness is typical for symbolic execution and often needed when symbolic execution does not start with the `main` method. However, in the case of web services, typically, the exposed endpoint could already be seen as a driver that can control the entry points of an application. For the case of Jaint, by building a harness that has control over a specific method inside the applications, the agent and testing component become interleaved because the application can no longer be tested through its exposed endpoints. By decoupling symbolic execution between the symbolic executor and symbolic explorer, we aim to develop a system that enables symbolic exploration steering through HTTP endpoints in natively running web services that require no mocking of system components or external libraries.

Recent works have demonstrated the potential of loosely coupled dynamic symbolic execution engines [36, 48].

To mitigate the above-mentioned issues and enable an effective combination of API-based fuzzing and symbolic execution as an IAST tool for Java-based web services, we developed a *symbolic web application testing platform* (SWAT) during this thesis. SWAT is based on the instrumentation-based dynamic symbolic execution engine CATG [64]. SWAT, and its differences from CATG, are introduced in this chapter, beginning with an architectural overview of the system's current state in Section 4.1. The instrumentation that enables symbolic execution is described in Section 4.2, followed by a detailed analysis of the symbolic backend that symbolically tracks instructions and generates constraints in Section 4.3. The harness to automatically initialize symbolic recording is described in Section 4.4. We describe our new symbolic explorer in Section 4.5. In Section 4.6, the effectiveness and efficiency of the symbolic engine is evaluated using SV-Comp, a standardized benchmark suite developed for automatic, comparable, and reproducible evaluation of software verification tools [15]. We evaluate SWAT's effectiveness in detecting vulnerabilities in Chapter 5 and provide comparisons to other vulnerability scanners.

4.1 Architecture

This section provides a general overview of the architecture and functionalities of the system. More detailed discussions of the different modules are given in the corresponding sections below. Common frameworks that enable Java applications to act as web services listening to multiple endpoints require a relatively long start-up sequence during which all components are initialized. External services such as databases are also attached. After the service is successfully started, new incoming requests do not start a new JVM instance but are usually just threads inside the application that execute the code serving the requested endpoint. Hence, a symbolic engine that does not require a restart of the application for each test that is executed would be favorable. A restart between iterations would entail a large overhead and, in turn, significantly worsens the performance. Loosely coupled engines such as GDart [48] already prevent repeated restarts. However, their design has an active harness that executes the method under test from inside the application. Such a harness does not use the underlying code that actually executed the method but instead uses a custom driver. SWAT is designed to allow the service to run in its normal configuration, and the symbolic executor is activated when a new request is incoming. The symbolic excecutor symbolically tracks the application code under test, and when the response is sent, the symbolic constraints that were recorded are sent to the symbolic explorer.

To achieve symbolic execution for continuously running web services in their native configuration, the architecture of CATG, as shown in Figure 3.1, was reworked into a loosely coupled design, where the symbolic explorer is separated from the symbolic executor. For CATG, because both of these components are part of the Java agent attached to the system under test, new inputs are obtained directly after symbolic execution. In a loosely

coupled scenario, depending on the strategy, the symbolic explorer can observe multiple symbolic execution runs that could, for example, be guided by a fuzzer before analyzing the execution tree and trying to determine new inputs.

An architectural overview of the different components of SWAT and their interaction is shown in Figure 4.1. Modules visualized with a grey background are not part of this thesis but are still shown to give a complete overview of the system in usage. Modules represented as a stack can be replicated to increase the system's performance. The symbolic executor is attached to the target using a Java agent in the same way CATG was added to the target. However, the symbolic executor (Section 4.3) is only responsible for instrumenting the target application (Section 4.2) and modeling the symbolic execution to build an execution trace augmented with the corresponding path constraints. The complete decoupling of the symbolic executor from any symbolic exploration strategy enables multiple instances of the symbolic executor running in parallel. Furthermore, we enabled the executor to run in multiple threads on the same instance as long as the threads do not interact with each other. Multi-threading is especially useful in the web scenario, as many frameworks allow concurrent execution of multiple different requests in different threads. The entry and exit point of the symbolic tracking can either be user specified through configuration files or several adapters for popular frameworks are available that allow for automatic detection of all endpoints (Section 4.4). The coverage engine works similarly to the symbolic executor without the symbolic tracking capabilities. The same execution trace is built, just omitting the symbolic annotations. Using the same trace format and identical identifier enables the combination of both symbolic and coverage traces in the same tree while making the coverage engine significantly faster, as it does not need to maintain symbolic capabilities. After a (symbolic) execution trace has been completely tracked, the information is sent to the symbolic explorer using HTTP.

The symbolic explorer exposes endpoints for both symbolic and coverage traces. After a trace is received, it is added to the execution tree. Depending on the selected scenario, the symbolic explorer can either drive the symbolic exploration itself or expose another endpoint to allow other components to drive the symbolic exploration. To obtain new inputs, the tree is analyzed using a search strategy to find nodes with unexplored branches. The constraints leading to that node are retrieved and sent to a solver to generate new inputs that satisfy the SMT instance. Solver communication is done through the APIs provided by different solvers such as Z3 [26], because the constraints are already received in the SMT-Lib format [9] by the symbolic explorer. The solution can either be relayed back to an external service, such as a fuzzer, responsible for testing the target or can be directly used as input on the target.

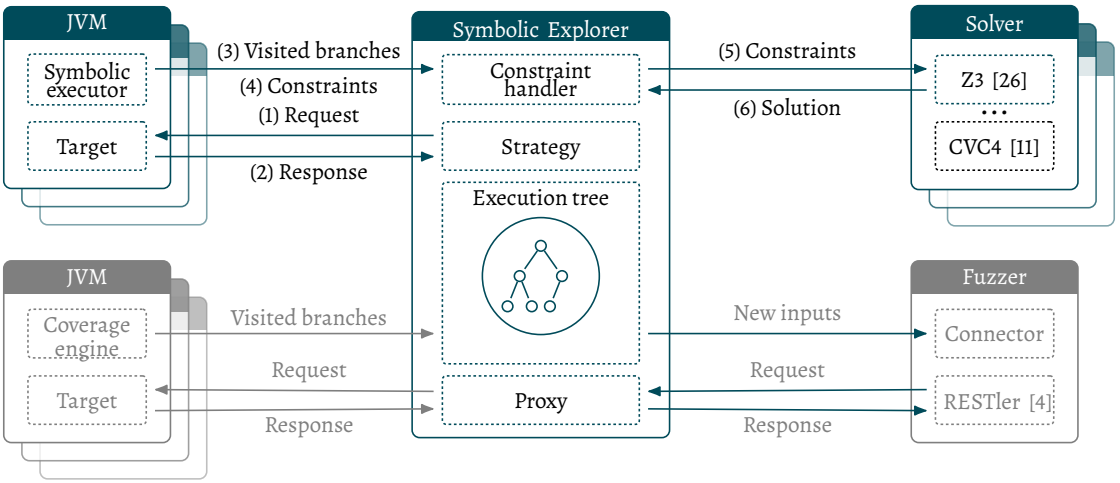


Figure 4.1: Architectural overview of SWAT modules and target interaction. Modules visualized in grey are being developed outside the scope of this thesis. The symbolic executor and coverage engine are attached to a JVM instance using a Java agent when the system under test (target) is initialized. The symbolic explorer is a stand-alone web service that can interact with other components using APIs. The symbolic explorer can send requests to the target application using the target's native endpoints. The symbolic executor records constraints and sends them back to the explorer, where an execution tree is built, and a strategy selects constraints that are sent to a solver to find new inputs. Modules that are visualized as stacks can be replicated to increase performance. Gray modules are developed outside the scope of this thesis.

All SMT solvers that allow instances in the SMT-Lib format can be used, but currently, Z3 is of primary interest because it recently integrated the string theory into the solver. This theory enables native solving of instances that contain constraints over strings without modeling strings as, for example, sequences of ASCII-encoded integers. Integration with RESTler [4] as an external testing component to test the target is outside this thesis's scope but is being developed at the Institute for IT-Security by Florian Sieck.

4.2 Instrumentation

Symbolic execution capabilities for Java-based applications exist in two primary flavors. Interpretation-based approaches, as introduced in Chapter 3, rely on an implementation of the JVM specification that provides capabilities to observe the JVM internals during the execution. While using a custom JVM is a powerful approach, it has a few fundamental drawbacks. Firstly, implementing a JVM is an expensive task; thus, all existing approaches rely on either the Java Pathfinder JVM [31] or the GraalVM [72] to provide the required capabilities. Secondly, whenever a new version of Java is released, the JVM must be adapted to support the new version. On the other hand, instrumentation-based approaches only need to be adapted when the instruction set changes. CATG [64], introduced in Chapter 3, is an instrumentation-based dynamic symbolic execution engine based on dynamic bytecode instrumentation that operates directly on the instruction set

specified by the JVM specification [40]. CATG [64] relies on the Java agent functionality and the ASM framework [20] (Section 2.2) to transform the bytecode of the system under test. The symbolic executor needs to observe each instruction to track what happens during code execution. For each instruction, CATG adds a call to a static support library during instrumentation that symbolically tracks the instruction. The parameters of the call also contain a method id (MID), instruction id (IID) and additional information that the instruction requires, such as indices. Figure 4.2 shows the instrumented version of the method shown in Figure 2.5. The highlighted instructions are the original instructions of the method. To symbolically track the iload_1 instruction at offset 8, offsets 0 and 2 push the IID and MID onto the stack. The instruction at offset 4 puts the index of the value onto the stack, and the instruction at offset 5 calls the corresponding static method, from the support library, with the required values. Additionally, the concrete value is duplicated at offset 9 and sent to the symbolic state at offset 10. The concrete value is used in the symbolic backend to validate if the symbolic value aligns with the concrete value.

```
int addition(int, int);
    Code:
       0: ldc           10247
       2: ldc           10
       4: iconst_1
       5: invokestatic de/.../DJVM.ILOAD        (III)V
       8: iload_1
       9: dup
      10: invokestatic de/.../DJVM.GETVALUE_int (I)V
      13: ldc           10248
      15: ldc           10
      17: iconst_2
      18: invokestatic de/.../DJVM.ILOAD        (III)V
      21: iload_2
      22: dup
      23: invokestatic de/.../DJVM.GETVALUE_int (I)V
      26: ldc           10249
      28: ldc           10
      30: invokestatic de/.../DJVM.IADD         (II)V
      33: iadd
      34: ldc           10250
      36: ldc           10
      38: invokestatic de/.../DJVM.IRETURN      (II)V
      41: ireturn
```

Figure 4.2: Java byte code of the method shown in Figure 2.5b after instrumentation by CATG. The lines highlighted in blue are the original instructions. For each instruction, a single static method call into the support library is added.

We can mostly reuse the instrumentation provided by CATG except for a few adaptions described below.

Static initializer (<clinit>) are methods that the JVM automatically invokes to initialize arbitrary class objects. Static initializers are invoked when a class is referenced for the first time and can contain arbitrary code. For example, when a static value from a different class is used somewhere, and the class the value belongs to has not been used previously, its static initializer is invoked to initialize the class. However, if the same method or value is used again, the class has already been loaded, and the initializer is not called again. For CATG, static initializers do not present a problem because between each symbolic execution, the JVM is terminated and restarted. Hence the same fields or methods may invoke the initializer again. However, as previously highlighted, SWAT allows the JVM to remain online between different runs. When SWAT executes the same piece of code a second time, the static initializers is not called again. If that initializer is also instrumented, it appears in the symbolic trace only once, leading to two different traces. One trace contains the possible branching behavior inside the static initializer, while the second trace does not. Without proper handling, it would lead to inconsistencies inside the execution tree. To avoid this, handling is added to static initializers that add a *special* node to the execution trace (Section 4.5). Using these nodes, static initializers can be modeled as branching points in the program that are not under the users' control.

Invokedynamic is a new instruction added to the JVM instruction set as an effort to support dynamically typed languages[2]. Despite Java being statically typed, invokedynamic finds several applications and enables lambda expressions. CATG [64] is not actively maintained, and hence instrumentation and symbolic handling for invokedynamic is missing. Before the addition of invokedynamic, method invocation was handled by one of four invocation types. For static methods, invokestatic is used, invokeinterface for interface methods, invokevirtual to call instance methods, and invocespecial for private methods or constructors. While these instructions differ in how the call site is retrieved, this difference is transparent to the symbolic execution engine and does not need to be explicitly modeled. During instrumentation, when a call to one of these four invocations is observed, the methods owner, name, description, and whether it is an interface are passed to the symbolic execution by CATG. However, when invokedynamic is used to call a method, in contrast to all other invocation instructions, its call site is unknown, also called unlaced, before executing the instruction. To allow for unknown call sites during the runtime, without relying on the reflections, each invokedynamic instruction references a bootstrap method (BSM) in the classes constant pool. Bootstrap methods are responsible for determining and lacing the call site for the actual invocation. The handling of bootstrapped methods is not always transparent to the symbolic execution and needs to be modeled accordingly as described in Section 4.3. To support the symbolic tracking of invokedynamic instructions, the handling for the existing invocations can be reused. However, the handling must be slightly adapted during instrumentation.

[2] https://jcp.org/en/jsr/detail?id=292

Instead of the method's owner, the bootstrap owner, alongside the method's name and description, is passed to the symbolic backend. Additionally, the bootstrap arguments are also observed and transferred.

```
int addition(int, int);
    Code:
     0: iconst_1
     1: invokestatic de/.../DJVM.ILOAD        (I)V
     4: iload_1
     5: dup
     6: iconst_0
     7: invokestatic de/.../DJVM.GETVALUE_int (II)V
    10: iconst_2
    11: invokestatic de/.../DJVM.ILOAD        (I)V
    14: iload_2
    15: dup
    16: iconst_0
    17: invokestatic de/.../DJVM.GETVALUE_int (II)V
    20: invokestatic de/.../DJVM.IADD         ()V
    23: iadd
    24: ldc           10243
    26: invokestatic de/.../DJVM.IRETURN      (I)V
    29: ireturn
```

Figure 4.3: Java byte code of the method shown in Figure 2.5b after instrumentation by SWAT. The instrumentation is based on the instrumentation by CATG as shown in Figure 4.2 but minimized to reduce the footprint. The lines highlighted in blue are the original instructions. For each instruction, a single static method call into the support library is added.

Optimizations to the instrumentation can lead to performance increases. We can decrease the number of instructions added for each target instruction by removing the MID from all instructions. The IID is already a unique id that can be used to identify the corresponding instruction. Furthermore, instructions that do not appear in the symbolic trace (Section 4.5) do not need an IID. The IID is only used as the correlating factor to combine multiple traces into a single execution tree. So only instructions that can alter the control flow, either through branching behavior, exceptions, or class loading, need an IID. While most of the runtime overhead is caused by the support library that performs the symbolic handling, the performance is also increased by reducing the number of added instructions. Especially if the number of instrumented classes is not limited to the methods that are symbolically tracked. The symbolic handling does not introduce overhead if classes are instrumented but not symbolically tracked. Hence only the execution of the instrumented instructions contributes to the overhead. By omitting the MIDs and only adding IIDs for instructions that can in some way alter the control flow of an application,

the size of the instrumented class can be reduced (Figure 4.3). Figure 4.4 shows an evaluation of the size reduction SWAT achieves compared to CATG. The bars visualize the relative increase in the number of instructions compared to the original class. A total of 4108 classes over three different datasets were analyzed for a representative result. The datasets used for the evaluation include the OWASP benchmark [51], OWASP WebGoat [52], and SV-Comp [15]. The OWASP benchmark [51] contains small samples of HTTP controller classes that may contain a vulnerability. The OWASP WebGoat [52] application is a fully-fledged web service that also contains several vulnerabilities SV-Comp [15] contains many classes used to evaluate verification tools. On average, SWAT decreases the overhead by 20% from 9.19× to 7.65×.

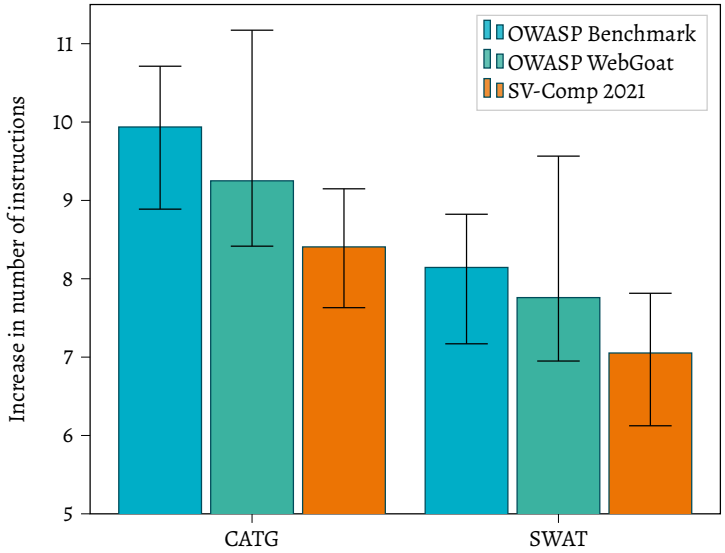


Figure 4.4: Relative increase of the number of instructions through instrumentation using CATG and SWAT against the original byte code. The datasets used for evaluation contain a total of 4108 classes. Each factor represents the median over the classes from the respective dataset. The shown error represents the distance between the median and the tenth and ninetieth quantile.

4.3 Symbolic Executor

To achieve a loosely coupled architecture, the symbolic executor, which is attached to the target, was significantly reworked. This section discusses changes made to the symbolic backend of CATG and features added to SWAT's symbolic executor to both decouple symbolic exploration and execution and increase the symbolic capabilities. To begin with, we introduce the new symbolic backend utilizing JavaSMT [6] for constraint handling. Symbolic peers for Java's built-in methods, including string handling, are introduced next. Following, the symbolic overflow modeling added in SWAT for all numerical values is specified. Next, we discuss differences in the integral division between the JVM and solvers. We introduce a model for correctly tracking the division utilized by the JVM. SWAT can also build symbolic constraints that validate if the target application

handles exceptions correctly. Lastly, changes in the shadow state are highlighted, which are required to correctly model language features of the JVM that allows SWAT to operate on modern Java versions and enable symbolic evaluation of web services that use multi-threading.

Symbolic Instruction Tracking

Instrumentation-based symbolic execution engines typically have symbolic handling for each instruction to maintain a shadow version of the runtime information. To correctly model the runtime information, maintaining a shadow memory is often required for languages like C. Because the JVM instructions operate on a stack and register machine, they do not directly modify the memory. Hence, the symbolic executor maintains shadow copies of the stack frames (see Figure 2.7). CATG [64] offers implementations for the shadow stack frames and has handling implemented for each instruction, including correctly updating the shadow state and maintaining symbolic constraints on supported data types. The initial design implemented in the CATG concolic engine communicates with the CVC4 [11] solver via a custom implementation that correctly formats constraints and utilizes a file for transferring the constraints. The symbolic backend is limited to CVC4, because constraints are formatted in CVC4's native input language[3] and not in the SMT-LIB standard format [9]. However, CVC4 is outdated and exceeded by CVC5 [8]. The solver dependency also limits the number of symbolically supported instructions provided by CATG. The concolic engine does not utilize newer features of the CVC4 solver, i.e., floating-point and string theories [10] and thus has no support for symbolically tracking either double or floating point values. The shadow stack is correctly modeled even with these values, but no symbolic information is propagated. For integers, only basic operations are supported symbolically (multiplication, addition, subtraction). All other integral data types, except booleans, are symbolically supported but modeled as integers without any constraints regarding the size of the actual datatype. Furthermore, overflows are not symbolically modeled for any numerical data types. The system has limited support for both string and array modeling. However, the solver does not natively support a string theory. Strings are symbolically tracked in the concolic engine by modeling a string of length n as n individual integers, each representing one ASCII-encoded character. The modeling requires solving for the length of a string first without binding the characters and limits the number of supported features. Arrays also have a custom model to support some operations on arrays that do not utilize a formal array theory.

To mitigate these issues and gain a solver-independent implementation, we utilize JavaSMT [6]. JavaSMT provides a Java-based standard API layer to build constraints using the SMT-lib format [9] and access several solvers interchangeably, including CVC5 [8], and Z3 [26]. The JavaSMT library introduces very little overhead compared to the native solver APIs while offering easy utilization of different solvers. Constraints are not stored in the memory of the JVM under test; only a reference to a constraint inside the context of the specific solver is stored. Utilizing JavaSMT requires new handling for all instructions that support symbolic handling.

The custom constraint logic initially implemented is substituted by the JavaSMT library that handles the concrete instantiation of constraints, as well as new value logic for each type on the shadow stack.

Table 4.5: Overview of symbolic tracking capabilities of Java's primitive data types by CATG [64] and SWAT.

	Double	Float	Long	Integer	Short	Character	Byte	Boolean
CATG	No	No	Yes[1,2]	Yes[1,3]	Yes[1,2]	Yes[1,2]	Yes[1,2]	Yes
SWAT	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

[1] Integers are modeled without size constraints leading to erroneous behavior if an integral value over- or under-flows.
[2] All other integral data types besides integers are modeled as integers leading to erroneous behavior if a value is casted into a range that the value exceeds.
[3] Only basic operations are supported symbolically (addition, subtraction, and multiplication).

The new symbolic backend allows SWAT to symbolically track and reason about all primitive data types, even supporting dynamic casting between data types. A comparison between the supported data types in CATG and SWAT is given in Table 4.5. Casting between data types, which CATG does not support, is essential because the JVM does not have primitive types for byte, short, character, and boolean. These data types are stored and operated on as integers and only bound explicitly to their datatype when required using the respective instruction (for example `s2i` for the conversion between short and integer). SWAT can symbolically model instructions that cast values between different ranges.

Table 4.6: Overview of the number of symbolically supported JVM instructions for both CATG [64] and SWAT.

	Symbolic tracking	Partial sym. tracking	Erroneous sym. tracking	Concrete tracking	No tracking
CATG	119	2	10	60	11
SWAT	172	4	4	11	11

Overall, as shown in Table 4.6, SWAT can fully support symbolic tracking on 172 instructions, compared to 119 for CATG, while reducing the number of only concretely tracked instructions from 60 to 11. SWAT also reduces the number of instructions that have (partially) incorrect symbolic handling to 4 instead of 10. To ensure the correctness of the symbolic model, all instructions that can either modify a numerical value or branch based on a numerical value are automatically tested for the correctness of the symbolic behavior. The 69 instructions that have the above behavior are tested against a total of 8382 test cases that validate behavior in normal ranges, corner cases, and overflow behavior. Instructions that manipulate the stack or other areas of the shadow state are currently

not explicitly tested. However, errors leading to misaligned stacks are more verbose and often trigger exceptions that are placed to assert the shadow state's correctness.

Java built-in library support

To rigorously test web services, symbolically tracking strings is vital. However, the JVM does not represent strings as a primitive datatype with native instructions. Instead, Java ships with a package (java/lang/String) that adds string handling to Java. We have two possibilities for symbolically tracking strings. The package and all dependencies could be instrumented to rely on the existing symbolic handling to track string values. However, instrumenting the packages would entail a significant runtime overhead and is currently not possible because Java's native methods, including String methods, are also used by the symbolic execution engine. To allow for the instrumentation of Java's built-in classes, the packages would need to be loaded twice under different packages, once for symbolic tracking and once for usage by the symbolic executor. Instead, in alignment with other engines, we add symbolic peers for the methods provided by Java's string class. While this approach requires manual handling for all exposed methods, it does not require the instrumentation of Java's base packages, significantly increasing performance. Another important set of classes that we provide symbolic peers for is the reference classes of all primitive types. Depending on the signature of a method, the compiler automatically performs boxing or unboxing. Symbolic information would be lost without a symbolic model for the boxed versions. In addition, the reference classes also expose methods that require symbolic models.

CATG also offers symbolic peers for the string, integer, and long classes with 21 supported methods. In contrast, SWAT offers support for all reference classes that wrap a primitive data type alongside support for string classes with a total of 44 supported methods. All string methods utilize the formal string theory described in the SMT-Lib standard [9] provided by JavaSMT [6]. The symbolic peers were developed in cooperation with Florian Sieck, Institute for IT-Security. In addition, SWAT also offers essential support for lists and enumerations. For all of these classes, the most common methods are implemented first, and while many are still missing, the implementation of the remaining methods primarily presents an engineering overhead. To ease the usage of the symbolic engine, we provide functionality that detects when a method is not instrumented and has no symbolic peer during execution. If that happens, the execution continues with the concrete return value and log the missing method. The log allows users to see what methods are currently missing and aid the decision of whether the method should be implemented and the evaluation rerun.

Overflow modelling

When operating on data types with fixed sizes, operations or instructions that increase or decrease the size of the value can cause the value to grow beyond the size of the data type. The over- or underflow behavior is defined by the JVM specification and bound by the binary representation of the corresponding datatype. Java's integral data types are

represented as signed binary numbers in two's complement format. The format enables representing negative numbers in binary format by utilizing the most significant bit of the binary number as a sign indicator. When operating on integral data types, the JVM performs bitwise operations, ignoring the sign and truncating values larger than the data type. These operations lead to overflow behavior where after the largest positive number the smallest negative number continues ($127_{10} + 1_{10} = -128_{10}$ / $01111111_2 + 00000001_2 = 10000000_2$). Because we utilize SMT-LIB's integer theory, modeling overflows by employing bitwise operations would require a transformation of the formulas at each step. We model the behavior inside the integer theory to prevent transformations after each instruction. Given a potentially out-of-bounds value $x \in [a, b]$, we can model the overflow as a function of x:

$$o(x) = ((x - a) \mod (b - a)) + a$$

By default, we initialize the bounds a and b to the respective data type's smallest and largest possible value. The function $o(x)$ is used for instructions that can cause an overflow within an integer, such as addition or multiplication, and is used when casting values to a set with smaller bounds. Correctly handling castings is especially important because all additions of integral values are handled as integer additions within the JVM and are casted back to their original datatype after the operation. Certain bitwise operations on integers, such as shifts or bitwise (X)OR, require a transformation to bit-vectors before applying the transformation. Hence these operations do not require overflow modeling.

Truncated division

The definitions used for the division and remainder operations in computer science can differ. Primarily the handling of negative divisors or dividends differs between definitions [19]. Given a dividend $D \in \mathbb{Z}$ and a divisor $d \in \mathbb{Z}$ with $d \neq 0$, the JVM specifies division behaviour for integers following a division dominant definition based on truncation (T-Definition) [19]:

$$\begin{aligned} D \ \text{div} \ d &= \text{trunc}(D/d) \\ D \ \text{mod} \ d &= D - d \cdot (D \ \text{div} \ d) \end{aligned}$$

The function $\text{trunc}(x)$ rounds all values of x towards zero so i.e. $\text{trunc}(3.9) = 3$ and $\text{trunc}(-1.3) = -1$. However, the integer division utilized in Z3 [26] uses a division dominant definition based on flooring (F-definition) [19]:

$$\begin{aligned} D \ \text{div} \ d &= \lfloor (D/d) \rfloor \\ D \ \text{mod} \ d &= D - d \cdot (D \ \text{div} \ d) \end{aligned}$$

The difference in definitions leads to incorrect behavior when using Z3's division function for modeling the xDIV and xREM instruction. To ensure correctness in all cases, the

following function is used to obtain truncated division by using floor-based division:

$$\text{truncdiv}\ (D, d) = \begin{cases} \frac{D}{d}, & \text{if}\ \ (D \geq 0) \vee ((D\ \bmod\ d) = 0) \\ \frac{D}{d} + 1, & \text{else if}\ \ d \geq 0 \\ \frac{D}{d} - 1, & \text{else} \end{cases}$$

Given the function for truncated division, the truncated modulus is obtained by utilizing the following function:

$$\text{truncmod}(D, d) = D - (\text{truncdiv}(D, d) \cdot d)$$

Exception modelling

The JVM allows instructions to throw pre-specified exceptions when certain conditions are not met that the compiler does not check. For symbolic execution, these exceptions are of interest for two reasons; firstly, modeling inputs, that cause the system under test to throw an unhandled exception and cause a crash, enables more thorough testing and eases bug finding. Secondly, exception catching allows the execution of special code to handle the caught exception. Essentially, exception catching can be interpreted as an additional branching condition that can be handled if the cause of the exception has a symbolic model. By providing symbolic models for exceptions, we effectively increase the range of code that can be systematically explored. We model a subset of runtime exceptions that check if an arithmetic value satisfies some bounds. In total, 47 instructions throw an exception, of which we can model 23 symbolically. These include 16 ArrayIndexOutOfBoundsExceptions, three NegativeArraySizeExceptions and four ArithmeticExceptions. Other exceptions, such as NullPointerExceptions, cannot be symbolically tracked because we have no model to validate whether a reference is valid and would not be able to find new values such that a class instance replaces the null value.

For example, when an instruction that loads a value from an array, such as IALOAD or AALOAD is executed, an additional branching node is inserted into the execution tree that models whether the index is inside the bounds of the array. Given an array with size y and an index z, and assuming the exception was not thrown during execution, the path constraint added is given by the SMT-Lib expression in Figure 4.7. If the actual execution already triggered the exception, the assert statement would be negated to model values that lie outside the array's bounds. When solving for the constraint, it is negated again to find values that either trigger or do not trigger the exception. Other instructions are modeled accordingly.

```
   ; Variable declarations
0  (declare-fun y () Int)
1  (declare-fun z () Int)

   ; Constraints
3  (assert (and (< z y) (>= z 0)))
```

Figure 4.7: SMT instance in SMT-Lib format [9] modelling the condition on which an ArrayIndexOutOfBoundsException is thrown while execution xALOAD or xASTORE. The prefix x can be substituted with any type modifier. The constraint asserts that an array index z is in bounds for an array of size y.

Handling dynamic invocations

While the primary usage for invokedynamic is specific to dynamically typed languages, the instruction finds several applications as part of the javac compile chain discussed below. With the addition of invokedynamic, the implementation of string concatenation changed. Instead of utilizing the StringBuilder for concatenation, the new implementation utilizes bootstrapping with the StringConcatFactory[4] to concatenate strings. Bootstrapping utilizes dynamic invocation. However, bootstrapped methods require novel symbolic handling, as the way the parameters are passed differs from previous invocations. The arguments for concatenation are passed inside the bytecode arguments, more specifically as part of the bootstrap arguments. These arguments are visible to the symbolic backend due to the addition to the instrumentation described previously. Since Java's String methods are not instrumented, after a call to the corresponding bootstrap method is detected, we can appropriately advance the symbolic state using symbolic peers and prepare the return value that is pushed back onto the parent stack.

```
0  public int ex(int input){
1      int localVariable = 2;
2      Function<Integer, Integer> lambda = x -> localVariable * x;
3      int res = lambda.apply(input);
4      return res;
5  }
```

Figure 4.8: Example application of a lambda expression visualizing different aspects of lambda expressions. The lambda expression is created in line 2. The method variable (line 1) is accessed inside the lambda expression. The constructed lambda expression is applied in line 3.

Lambda expressions are a feature enabled by invokedynamic and are, in contrast to the

[4] https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/StringConcatFactory.html

reworking of the string concatenation, not oblivious to the developer. They allow for a new language construct. Figure 4.8 shows an exemplary application of a lambda expression. Lambda expression are a reference type, and the expressions parent object has to inherit a functional interface.

```
Constant Pool:
 #7  InvokeDynamic   #0:apply:(I)LFunction;
#11 Method           Integer.valueOf:(I)LInteger;
#12 class            Integer
#17 InterfaceMethod Function.apply:(LObject;)LObject;
#22 Method           Integer.intValue:()I


public int ex(int);
    Code:
       0: iconst_2
       1: istore_2
       2: iload_2
       3: invokedynamic     #7,  0
       8: astore_3
       9: aload_3
      10: iload_1
      11: invokestatic      #11
      14: invokeinterface   #17,  2
      19: checkcast         #12
      22: invokevirtual     #22
      25: istore            4
      27: iload             4
      29: ireturn

private static Integer lambda$ex$0(int, Integer);
    Code:
       0: iload_0
       1: aload_1
       2: invokevirtual     #22
       5: imul
       6: invokestatic      #11
       9: areturn
```

Figure 4.9: Byte code for the source code shown in Figure 4.8. For readability, the Constant Pool is significantly simplified. Entries that are not explicitly used in the code at hand are omitted. Entries in the pool are dereferenced, and the fully qualified class names are shortened. The method public int ex(int); is the compiled version of the method from Figure 4.8. The second method (private static Integer lambdaex0(int, Integer);) is generated from the body of the lambda expression shown in Figure 4.8 (line 2). It is compiled into a static method in the same class as the initial expression, shown here. The lambda method performs the transformations specified inside the lambda expression. It is constructed during compilation and is visible to all class loaders.

A lambda expression is created in line 2, and the expression is assigned an object of type Function, with one parameter and one return value, both of type Integer. The lambda expression is applied in the following line (3). To gain a deeper understanding of how far the dynamic invocation differs from other invocations and what needs to be adapted to allow symbolic tracking of the parameters and return values of lambda expressions, the bytecode for the source code from Figure 4.8 is listed in Figure 4.9. At first, a simplified version of the Constant Pool, alongside the compiled method (public int ex(int);), is given. The lambda expression is rendered as a static method (private static Integer lambdaex0(int, Integer);) inside the class and not as an inner class. Compiling the expressions as methods has optimization reasons that are omitted here. The static method is generated at compile time and is thus visible to the instrumentation engine. So, symbolically tracking what the lambda expression is performing is trivial. However, a difference is apparent in comparing the signature of the initial Function<Integer,Integer> object that stores the lambda expression and the signature of the generated lambda method. In contrast, the original lambda expression requires one parameter, and the generated method requires two parameters. The difference in parameters would not be a problem if the parameters were prepared on the original stack. However, the invocation of the lambda's apply function (Line #17) has a single parameter and does not call the actual lambda method but an interface method of the object.

When the invokedynamic instruction is executed, the bootstrap method, which is part of the LambdaMetaFactory, generates an implementation of the target functional interface. The generated class is a synthetic class that is dynamically generated during runtime and is not accessible through the instrumentation API. Hence, we cannot instrument the synthetic class and are oblivious to the class's behavior; thus, a generic symbolic model of this class is required to enable symbolic tracking of lambda expressions. The simplified generated class for the example in Figure 4.8 is listed in Figure 4.10. It is responsible for invoking the method containing the lambda expressions logic. Analyzing the class, it is apparent why the signatures do not align in the example. The lambda expression uses a local variable from the original method that is unavailable in the generated static method. Hence, the generated method includes any formal arguments of the lambda expression along any captured variables. The captured values, a primitive integer in the example, are passed to the synthetic class through the bootstrapping method (as part of the invokedynamic call) and are stored as class variables in the synthetic class (arg$1) during initialization. Since the values are duplicated when the synthetic class is created, any captured values in lambda expressions are required to be final or at least effectively final. The method (public Object apply(Object);) generated as part of the synthetic class is the method that is called by the original method and bridges the gap to the lambda method.

To summarize, lambda expressions are compiled as a static method inside the original class that is instrumented, allowing symbolic tracking of its logic. The static method, visible to instrumentation, expects the formal arguments of the lambda expression and any captured values as parameters. When the compiler hits the creation of a lambda expression, an invoke dynamic call is added. On the first invocation of invokedynamic, a bootstrap method is called which generates a synthetic class that implements the functional

interface of the target and delegates to the method containing the logic of the lambda expression. After creating the delegate, the call site is linked, and the bootstrap method is not required again unless something changes. The synthetic class contains any captured values passed as part of the invokedynamic arguments. To the best of our knowledge, this class cannot be instrumented, requiring a symbolic model that correctly handles the parameters passed to the lambda method.

```
final class Helper$$Lambda$207 implements Function

Constant Pool:
 #13 Method      Object."<init>":()V
 #15 Field       arg$1:I
 #19 class       Integer
 #25 Method      Helper.lambda$ex$0:(ILInteger;)LInteger;
{
    private final int arg$1;

    // $FF: synthetic class
    private Helper$$Lambda$208(int);
        Code:
           0: aload_0
           1: invokespecial #13
           4: aload_0
           5: iload_1
           6: putfield      #15
           9: return

    public Object apply(Object);
        Code:
           0: aload_0
           1: getfield      #15
           4: aload_1
           5: checkcast     #19
           8: invokestatic  #25
          11: areturn
}
```

Figure 4.10: Synthetic class generated at runtime by the JVM for the lambda expression in Figure 4.8 (line 2). For readability, the Constant Pool is significantly simplified. Entries not explicitly used in the code are omitted. Entries in the pool are dereferenced, and the fully qualified class names are shortened. The class is generated using a modified version of ASM [20]. The class prepares the arguments required to call the static lambda method (lambdaex0) shown in Figure 4.9. The apply method from the synthetic class is called and, in turn, calls the actual lambda method.

The symbolic handling requires two stages because the captured values are prepared at the location of the invokedynamic instruction. However, the invocation that leads to the

execution of the instrumented lambda method is decoupled from the initial invokedynamic call (compare Figure 4.9 (offset 3 and offset 14)). The two instructions are connected by the instance of the synthetic class generated during bootstrapping. The synthetic class that is initialized during the invokedynamic and consumed where the lambda is invoked. The dependency can also be observed in Figure 4.9, where the Constant Pool reveals that the return value dynamic invocation is an instance of Function (see entry #7) that is consumed by the invokeinterface (see entry #17). We take advantage of this dependency and augment the symbolic object returned by the dynamic invocation with the additional captured values that are passed to the lambda method. When the lambda invocation is reached, we retrieve the additional values during the preparation of the symbolic frame. We can thus correctly modify the method's frame with the corresponding values. While handling captured values represents the general procedure, other conditions, for example, when the parent reference is required inside the method, are handled accordingly.

Multi-threading

Popular frameworks for developing web services in Java spawn a new thread per request, allowing multiple requests to be received simultaneously. Hence a symbolic execution engine is required to keep track of multiple threads in parallel. Symbolic execution engines built on top of the JPF-JVM [31] can symbolically model multi-threaded behavior and even detect locks in inter-thread communications. Multi-threaded lock detection is not available in any instrumentation-based symbolic engine. CATG [64], in particular, allows for no multi-threading because the shadow state models one JVM stack (Figure 2.7) and has no way of differentiating from what thread an instruction originated. SWAT uses an adapted shadow state where thread-specific information is stored per thread. The executing thread is identified when an instruction is executed and the appropriate shadow state is advanced. Effectively, the module responsible for tracking symbolic instructions and advancing the shadow state is fully stateless and operates on the correct state based on the current thread. Having the symbolic state stored per thread and a stateless symbolic driver allows parallel symbolic execution of any number of threads. However, we assume no intercommunication between the threads; for example, when two threads access and modify the same static variable, the symbolic constraints are currently not correctly adapted. Nevertheless, if the static information is only read or is not part of the constraints, the symbolic state is correctly tracked.

4.4 Symbolic Initalization

During symbolic execution, a subset of variables must be marked as free variables. The SMT solver could not find new values without any free variables when all values are bound. We call these free variables *symbolic variables* because the symbolic executor tracks them in abstract form, not their concrete value. Marking values as symbolic that are not manipulable by the user or the harness makes little sense. Hence specifying which values should be symbolically tracked depends on the target application. SWAT offers several adapters

to track specific methods and variables symbolically automatically. Two generic drivers are described below. Because SWAT is focused on enabling web services to be tracked symbolically, web-specific adapters are integrated to allow the developer to choose an appropriate technique for automatically tracking values symbolically. Web-specific drivers are introduced at the end of this section. CATG [64] requires the user to modify the application's source code to add method calls that mark variables as symbolic. For example, when an integer should be symbolically tracked, the symbolic value has to be initialized by calling $\mathrm{int\ x = CATG.readInt(2);}$. To add these calls manually, one requires access to the source code or knowledge of how to add calls directly in bytecode, making the approach impractical. The manual placement of calls also prevents symbolic initialization in libraries without manipulating the library code before loading it. Additionally, the manual approach is time-consuming when the target application contains many variables that should be symbolically tracked.

A method that marks the beginning of the symbolic scope can often be identified. Mostly, the parameters of these methods should be symbolically tracked. Such a method can also be identified in the examples used by Tanno et al. [64] and in benchmark datasets such as OWASP Benchmark [51] (Section 5.2). We exploit this observation to offer automatic initialization of symbolic variables. The user can specify methods or a regular expression to match a set of methods that should aid as symbolic initializers. Automatic initialization is facilitated by adding an additional instrumentation pass, during which we detect methods that match the specified signatures. If a match is found, the method body is retransformed to mark all parameters symbolically. Automatic transformation to enable systematic symbolic initialization eases the usage significantly and mitigates the requirement for the user to add additional logic to the source code.

Aside from manually specifying symbolic values in the code and the method adapter described above, SWAT also offers the option to automatically turn values returned from user-specified function calls symbolic. Using a user-specified signature and configuration option, SWAT automatically adds the required logic to make values that match the descriptions symbolic by performing an additional instrumentation pass. Such symbolic return values can, for example, be seen in the benchmark for Java-focused verification tools, SV-Comp [15] (Section 4.6). The benchmark assumes all values returned from methods satisfying the following regular expression to be free variables: $\mathrm{r'org/sosy\_lab/sv\_benchmarks/Verifier/nondet.*'}$. Providing the regular expression, the instrumentation engine can identify all invocations of the specified method in instrumented classes with no further action required by the user.

Aside from specifying what variables should be symbolically tracked, the scope of the symbolic execution also needs to be defined. In particular, entry and exit points for symbolic tracking need to be specified. While a user could manually add these calls to the target application (see CATG), SWAT allows the user to specify methods that can be seen as the harness for the symbolic execution. We automatically add required calls at the beginning of all methods matching the specification to initialize the symbolic tracking and add handling before each return statement contained in the method to terminate

symbolic tracking. These calls are added using an instrumentation pass. Such a method could, for example, be the main method of a program. When the main method is specified, the symbolic executor begins tracking when the method is entered, so when the JVM startup sequence is complete, and exit just before termination of the program or just before the method returns. Each method that correlates with an endpoint could be specified in the web scenario to enable symbolic tracking when the request is handled. Specifying multiple such methods that can be invoked, for example, by a web server, allows the symbolic engine to track multiple threads in parallel. During the symbolic initialization of a particular variable, its type is determined, and constraints are built that reflect the lower and upper bound of the symbolic variable. By default, the bounds are enforced by the JVM, which is done to ensure compatibility with solvers that have potentially larger bounds. Otherwise, the solver could find solutions outside the actual value range. Bounds on symbolic variables can also be used to limit the search space by supplying tighter bounds for variables. These bounds can be specified as a configuration option by the user.

Spring endpoint

A popular framework widely deployed for developing and serving web applications is the Spring framework [68]. The Spring Web MVC framework[5] allows developers to easily create RESTful web services by providing annotated controller classes.

```
0 @PostMapping("/request/{id}")
1 @ResponseBody
2 public String postController(
3         @RequestBody LoginForm loginForm,
4         @PathVariable("id") int id) {
5     ...
6 }
```

Figure 4.11: Example of a Spring [68] controller that is called by the framework when a POST request is received at the path "{base}/request/{id}". The base path ({base}) is determined by the parent class. The id {id} is an HTTP path variable that is passed as a parameter to the method. The request's body is automatically deserialized into an object of the LoginForm class. The return value of the method is sent back as the response body (@ResponseBody).

An example of a method signature invoked on a specific post request is seen in Figure 4.11. The method that handles the logic between request and response is part of the codebase of the target. By specifying the @PostMapping("path") annotation, the method is registered by the Spring library as a method that listens for POST requests on the specified path. The method has to be part of an annotated controller class with a base path. The

[5] https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

handling between the receiving of the request in the web server to the correct invocation of the specified method with its parameters is handled by the library and is not instrumented. Using the previously described technique to wrap methods symbolically, we automatically identify and wrap all methods with the correct signature and parent class. User-specified values that are part of the request are passed as parameters. Hence we also use the previously described technique to mark all variables as symbolic. Spring offers automatic deserialization of the request's body into class objects such as the `LoginForm` given in the example. We also provide support for full symbolic tracking of these data objects. When such an object is encountered during the runtime and is not yet instrumented, we instrument the class to mark all class variables as symbolic. To summarize, we give the user the possibility to specify that the system under test is a Spring-based web service. SWAT then automatically identifies all exposed endpoints and wraps them symbolically during instrumentation. The user can then use any system to send requests to the target, including the symbolic explorer (Section 4.5). After the request is successfully handled, the symbolic constraints are automatically sent to the explorer for further investigation.

HTTP Servlet

Servlets provided by the `javax` package are another technique often used for serving HTTP endpoints. A servlet is a class that is responsible for request handling. Each servlet serves a unique resource identifier (URI) and offers methods for all *CRUD* (create, read, update, delete) functionalities. The servlet container is responsible for receiving requests from the web server and calling the correct target servlet. The application under test is the user-supplied code inside the servlet, not the underlying framework. Hence, instrumenting the entire framework would significantly increase the time spent executing parts of the application that are not under observation. An example of a method responsible for serving the `POST` requests is listed in Figure 2.9. We can automatically match all classes and methods listening to the servlet container using an instrumentation pass. Using the above-discussed approaches of making the parameters symbolic does not directly transfer here because the parameters are objects inheriting the `HttpServletRequest` and `HttpServlerResponse` classes for the request and response object, respectively. The request and the response object are deeply nested within the framework; thus, making these objects entirely symbolic is only feasible when symbolically tracking the entire framework. However, the request object contains the values of interest, such as the cookies, headers, and body of the request. One approach is to specify all methods that the objects can have and mark all values these methods return symbolically. While the general approach of marking the values symbolic that are retrieved from the request object is promising, manually specifying all possible methods among different implementations of the `HttpServletRequest` is time-consuming and error-prone.

To mitigate the need to specify all methods manually, we introduced a new behavior in SWAT that allows generic objects to be marked as symbolic. The symbolic flag is transitively propagated when an object is marked symbolic and not instrumented. Propagation is done by either method invocations or setting fields of the object. Transitivity is

achieved by marking all objects retrieved from the symbolic object as symbolic. When a primitive value is retrieved from a symbolic object, it is marked as a symbolic variable. In the case of servlets, the transitivity allows easy symbolic tracking of invocation chains such as $\mathrm{request.getHeaders().getHeaderNames().nextElement()}$ that can be scattered across several lines in different classes. Symbolic objects also allow us to mark previously concrete values, primitive or object, as symbolic when they are stored in the symbolic object. We modify the routine developed by CATG that is responsible for fetching concrete values, used to keep the stack up to date, to check if a value should be symbolically flagged. The flag is set during instrumentation, and the appropriate propagation is handled during the runtime.

4.5 Symbolic Explorer

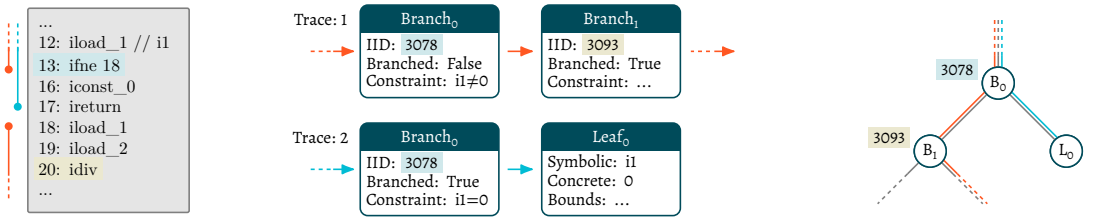
The symbolic executor (Section 4.3) is responsible for generating traces containing path constraints for the instructions visited during execution. One trace is generated per execution of the symbolic executor, and the processing of the execution traces is completely decoupled from the symbolic executor. The symbolic explorer is an independent component that exposes several endpoints to receive information from the symbolic executor. The independence from the symbolic executor allows the symbolic explorer to be located on a separate host, increasing the performance and enabling the target application to be hosted on multiple machines in parallel.

The symbolic explorer receives traces, described below, from the symbolic executor and adds the trace to the corresponding execution tree. After the trace is received, the HTTP connection is terminated before the trace is processed to minimize the time the symbolic executor spends waiting. The asynchrony is possible because the symbolic executor does not require any information from the explorer since it does not keep track of the state of symbolic execution. After a trace is successfully sent, all information regarding the trace is discarded. The explorer also offers endpoints to query for inputs leading to previously unexplored branches. To facilitate the symbolic execution of a target with multiple symbolic entry points, i.e., multiple endpoints, each trace contains an id that identifies the symbolic entry point. The explorer uses that id to distinguish between different execution trees stored in the database. The traces received from the symbolic executor contain all information required to sequentially build the execution tree, determine unexplored branching points and create an SMT instance that represents the path.

A *trace* $\mathcal{T} = (T, I)$ contains a list of execution elements T and a set of inputs I. Each *execution element* contains an instruction id. There are two types of execution elements; *branch* elements additionally contain information on whether execution branched at the specific instruction and, if available, the constraint(s) that model the branching condition. All instructions that can actively diverge the control flow and instructions with a symbolic model for their exception behavior are represented in the trace using a *branch* element. $\mathrm{Special}$ elements model all instructions that could cause an exception that has no symbolic model (see Table A.1) and instructions that can cause a class to be loaded

and the static initializer to be executed. Classes can be loaded either by invoking a static method (invokestatic), creating a new instance (new), or setting/ retrieving a static field (put/getstatic) of a class that is not yet stored in memory. While the symbolic explorer cannot determine new inputs for these elements, they are required to build a consistent execution tree. Without special elements, different execution traces could not be combined and inserted into a binary tree.

The trace $\mathcal{T}$ also contains a set of *inputs I* that contains one input element for each symbolic variable encountered during the execution. Each input element contains the mapping between the symbolic name (used in the solver context) and the concrete value seen during execution. The concrete information is required, especially when a satisfying assignment does not contain all variables. Input elements also store the variable's type information and the associated constraints for bounding the value to the allowed value range. The trace is serialized using JSON to allow easy interoperability between different explorers and executors in the future.



(a) Program (b) Two partial execution traces (c) Partial execution tree

Figure 4.12: Example for trace and execution tree generation. The program snippet (a) loads an integer (symbolic integer i1) and branches based on the value of the integer. One branch returns, while the other performs an integer division. The part of a trace correlating to the snipped is shown in (b). Trace 1 contains an execution that did not branch, while trace 2 branched. The IID for each branch element is the id for the corresponding instruction. The corresponding part of the execution tree is shown in (c).

All execution trees start with a root node representing the specific entry point (such as a method) into the symbolically observed code region. With every new trace received, the tree gradually expands as the number of paths the symbolic executor has visited increases. Each tree is a binary tree where each node has, at most, two children, and no cycles are present. The binary structure without cycles is inherited from the structure of the execution traces, as each trace is a linear sequence of elements that contains no cycles. While the program that is analyzed may contain cycles, the trace only contains the unrolled loop with a fixed number of iterations after execution. The instruction ids are used to assemble the tree. All instructions that can diverge the control flow are represented as inner nodes. Each leaf represents the inputs that were recorded along the respective path. Figure 4.12 visualized the steps from a code snippet (Figure 4.12a), via traces (Figure 4.12b) to the execution tree (Figure 4.12c). The program excerpt loads an

integer, assumed to be symbolic, compares it against zero, and either returns zero or performs a division. The excerpt is chosen to highlight the creation of a branch based on a conditional jump (ifne) or based on exception modeling (idiv). Parts from two traces are highlighted that correspond to the code shown. Trace 1 did not branch at offset 13, while trace 2 did. The constraints are listed in the trace elements based on the branching behavior. Trace 2 returned at offset 17 and hence has a Leaf element as its last trace element that stores the symbolic and concrete values of the inputs and their bounds. The combination of both traces based on their instruction ids (IID) is shown in the execution tree. The dotted edges and arrows symbolize the continuation of the tree or trace outside the scope of the program section.

To find new paths to explore, branching points with symbolic conditions and an unexplored path need to be determined. As briefly discussed in Section 3.1, work exists to optimize the selection of new branches. While optimized heuristics are also planned for SWAT, a breadth and depth-first (BFS/ DFS) search is currently supported. Using BFS, nodes closer to the root are preferred, and thus branches that are earlier in the execution trace are preferred, whereas, for DFS, branches closer to a leaf are selected first. In combination with a fuzzer, selecting branches retrieved using BFS may enable the fuzzer to fuzz large new subtrees while using DFS we may be able to guide the fuzzer towards deeper, and more specific, parts of the state space. We can observe the number of times the driver has passed a branching point. Weighing nodes retrieved using BFS by the number of times they were seen may also prove beneficial in combination with a fuzzer. All nodes with symbolic constraints and only one child are selected as possible branches during searching. Before a new input can be determined, all path constraints that need to hold to reach the point in the execution need to be added to the SMT instance. The constraints are accumulated by traversing the tree from the selected node up toward the root, collecting all encountered constraints, and adding them to the SMT instance. The constraint that should be solved for is negated to find an instantiating that leads the execution into the unexplored areas. Additionally, the bounds for all symbolic inputs are added to the instance before the solver is queried. If the solver does not find a satisfying assignment within a configurable time, the next branch is selected, and the process is repeated. If a new solution is found, it is stored in the tree, similar to the inputs, as a leaf at the position where the values should lead. Storing the solutions as leaves prevents the solver from accidentally solving for the same branch multiple times and allows the solver to constantly solve for new branches without actual execution of the inputs. The leaf is replaced with the actual trace when it is encountered.

Currently, the solver is only used to find any instance that satisfies the constraints. However, using an optimization solver to find minimal or maximal assignments will be explored in the future. To facilitate machine learning guided state selection in the future we offer additional metadata per branching point. The additional information that is currently stored per branch includes the size of the current method stack, the locals size, and the depth of the call chain. This information is also stored as part of the execution tree.

4.6 Evaluation

Before we discuss specific features of SWAT for finding vulnerabilities in web services, we evaluate the efficiency and effectiveness of SWAT as a symbolic execution engine compared to other symbolic execution engines and tools for verification of Java software. This evaluation serves as a reference to quantify the performance of SWAT and allows classification and placement of SWAT compared to the current state of the art. CATG [64] is not included in the evaluation because of its condition. Without significant modifications, the system cannot run at all, and with the number of missing symbolic datatypes, it cannot reasonably compete on the benchmark we present below.

SV-Comp [15] is a yearly competition held at the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), providing an established set of verification tasks for comparing tools focused on software verification. The competition has several tracks, including a Java track. The tasks are all compiled using Java 7 and do not test for modern language features such as lambda expression. However, the benchmark provides an independent and comparable dataset with known properties that can be used to compare and classify the performance of verification tools. Each test case consists of several classes and a known entry point. The task of the verifier is to evaluate whether the code contains any $\mathrm{assertions}$ that are reachable and do not hold. The benchmark assumes all values returned from methods that match $\mathrm{r'org/sosy\_lab/sv\_benchmarks/Verifier/nondet.*'}$ are free variables. As previously described, SWAT provides an adapter that easily enables symbolic tracking of all these values. However, methods provided by the benchmark return a random value instead of a controllable value. To enable symbolic execution, we provide a separate class, which reimplements the methods and is automatically substituted during instrumentation. The benchmark contains a total of 473 test cases, out of which we compared SWAT on 297 cases. The tests we used for evaluation are the regression tests from three popular verification tools, including two symbolic execution engines, JDart [43] and JPF [31], and JBMC, a bounded model checker [24]. Other test classes were omitted because of (currently) unsupported features such as recursion, multi-threaded lock detection, or unsupported looping behavior. The regression tests are comparatively small test cases that evaluate specific functionality.

To provide comparable results, BenchExec, a framework for reliable benchmarking and resource measurement [16], is used to orchestrate the task distribution and provides a wrapper for each evaluation run. We evaluated all tools using a 60 second time limit and a memory limit of 3000 MB with one core per task against the same subset of tasks to provide comparable results. The evaluations were performed inside the BenchExec framework on an AMD EPYC 3151 4-Core Processor running Ubuntu 20.04.4 LTS with 128 GB of RAM. To enable the classification of SWAT compared to the current state of the art, we include all competitors from the Java track at SV-Comp 2021 [15] in our evaluation. The competitors include several symbolic execution engines. Namely, SPF [55], COASTAL [36], JDart [43], and Java Ranger [61] a symbolic execution engine built on top of SPF that enables path-merging using veritesting for Java. Veritesting combines static and dynamic symbolic execution [5] to limit the effects of path explosion by summariz-

ing paths using static symbolic execution. Additionally, JBMC [24], a bounded model checker, and JayHorn [37], a model checker based on Horn clauses, are included.

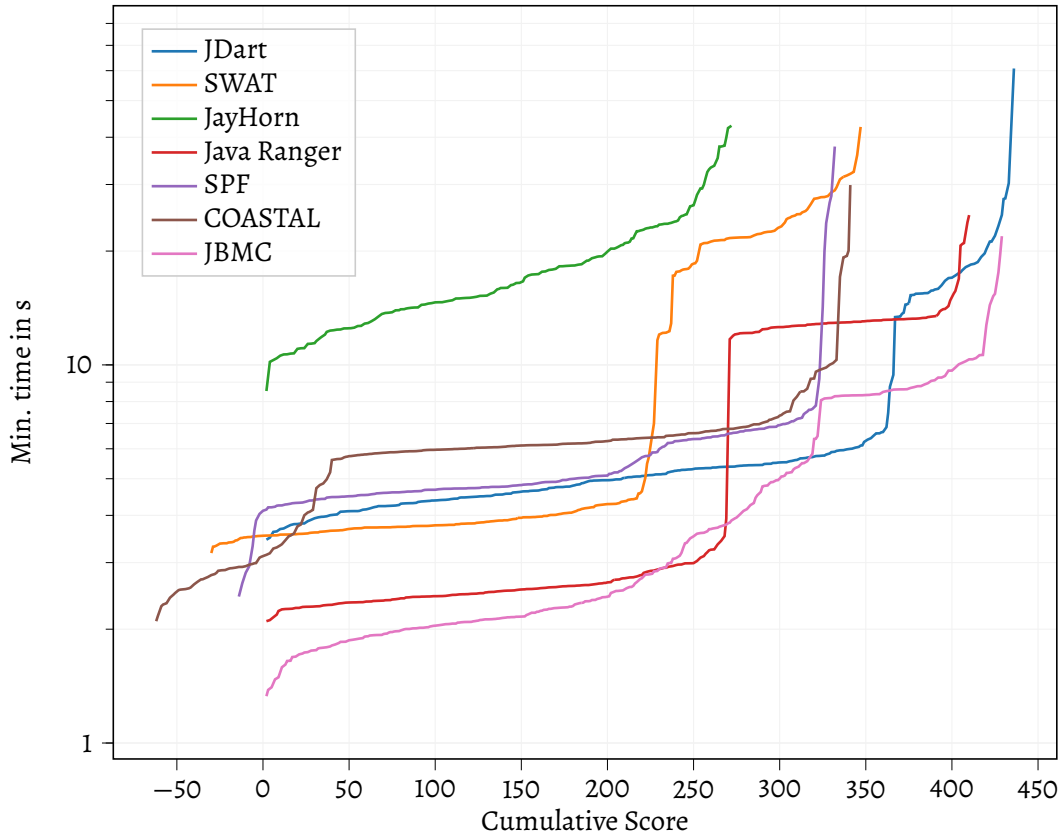


Figure 4.13: Score-based quantile plot [14] for qualitative analysis of verification results. Tools evaluated on a subset of tasks from the SV-Comp 2021 [15] Java track using the benchmarking framework BenchExec [16]. Each verification task is ordered by the runtime as a cumulative score. The y-axis shows the time taken for each task on a log scale. The x-axis shows the cumulative score for each tool.

Each tool can classify a test case as either *True*, *False* or *Unknown*. Test cases labeled as *True* contain no violation. That is, no instantiation of the variables exists, so an assert statement inside the test case is violated. Analogously, for test cases labeled *False*, an instantiation of the variables exists, leading to an assertion violation. Correctly identified test cases without a violation are awarded two points, while misclassified test cases without a violation are penalized by −32 points. Test cases containing a violation are rewarded with one point when correctly identified and penalized with −16 points otherwise. If a violation is present the test case is weighed less because it is generally easier to prove the presence of a violation (by finding the correct) inputs than to prove the absence of one. Tools can also classify a test case as unknown, resulting in neither gained nor lost points. When we detect that a call leaves the instrumented area and no symbolic model for the function is available, and no instantiation proves the presence of an assertion, we label the test case unknown.

To understand the comparative evaluation results, Figure 4.13 plots score-based quantile functions for each participant. The plot is analogous to the quantile plots used in the SV-Comp competition results by Beyer [15]. The figure plots time in log scale on the y-axis against the cumulative score on the x-axis. Each function maps the minimum required time it took to achieve a given score. The left end of the graph visualizes the total amount of incorrect work, while the right end highlights the tools achieving the highest score. If a tool reaches the same number of points as another tool while maintaining a lower runtime for the slowest test case, its y value is lower. Hence the rightmost tool with the lowest y value is the best-performing tool regarding effectiveness and efficiency. Overall, in the cumulative score, SWAT scores fourth place behind Java Ranger [61] in third place, JBMC [24] in second, and JDart [43] in the first place. This places SWAT as the second highest-scoring purely dynamic symbolic execution engine and the highest-scoring instrumentation-based dynamic symbolic execution engine in front of COASTAL [36]. JayHorn scores last here; however, in 66 cases, it ran into a timeout. JayHorn's time issues are already apparent from Figure 4.13 as JayHorn requires the most time for each quantile. The runtime for the other tools remains almost the same until they reach approximately 200 points. SWAT is the first tool to see a significant increase in runtime, followed by Java Ranger and SPF. However, the way the test cases are designed, without a symbolic harness that can drive the function, each test case requires a restart of the JVM for each new input. SWAT does not provide such a harness by design because the system is focused on testing applications that stay online (web applications) between runs. The design of the benchmark implies that the efficiency of SWAT evaluated by this benchmark is not transferable to the targets SWAT is designed for. Additionally, because each test is run in a separate container, we have to run the symbolic explorer for each test case adding additional overhead that is not required in an actual deployment. Furthermore, the benchmark is designed to be compiled using Java 7. Hence, the benchmark does not consider modern language features such as lambda expressions that were introduced in Java 8.

A detailed breakdown of the results is listed in Table 4.14. The results are split into three groups. Test cases that were correctly classified appear in the first group. Incorrect classifications are listed in the second group, and all other cases are accumulated in the third group. Undetermined test cases include timeouts, out-of-memory errors, and unknown verdicts of the verifiers. Each group lists the total number of cases in the group and a differentiation between test cases that contain no violations (True) and cases that contain a violation (False). Lastly, the overall score achieved by the respective tool is given. e SWAT achieved a score of 347, and correctly identified 87.54% of all test cases and 92.76% of the violations, while 82.07% of test cases without a violation were correctly identified. In addition, 70% of the undetermined test cases contain no violation. The results indicate that SWAT is more effective at finding violations than proofing their absence. SWAT missed ten test cases containing a violation, and all ten of these cases contained methods for which no symbolic model exists yet. Out of the 24 test cases containing no violation classified as unknown by SWAT, 29.17% were due to missing symbolic methods.

Table 4.14: Detailed results from SV-Comp 2021 [15] benchmark with all competitors and SWAT on 297 out of 473 test cases. Results include all regression tests from JBMC [24], JDart [43] and JPF [31]. Results are grouped by correct, incorrect, and undetermined results. Undetermined results contain timeouts, out-of-memory errors, and unknown scores by the verifier. Each group is split into test cases that contain no violations (True) and cases that contain violations (False).

	JDart	JBMC	Java Ranger	SWAT	COASTAL	SPF	JayHorn
Correct Results	293	289	271	260	271	226	172
Correct True	143	140	139	119	134	122	100
Correct False	150	149	132	141	137	104	72
Incorrect Results	0	0	0	1	2	1	0
Incorrect True	0	0	0	1	2	0	0
Incorrect False	0	0	0	0	0	1	0
Undet. Results	4	8	24	36	24	70	125
Undet. True	2	5	4	26	11	22	45
Undet. False	2	3	20	10	13	48	80
Score	436	429	410	347	341	332	272

Various reasons cause the remaining cases; 8.33% are caused by either a timeout or out-of-memory error. Timeouts can occur for several reasons. The execution of the target or constraint solving may not finish in the required time. Alternatively, a combination of several factors may result in insufficient time to explore all branches. Other unknown verdicts are either caused by an unexpected error during execution or a combination of constraints that are not solvable due to the limitations of the solver. We have only rarely observed unsolvable constraints. The ones we observed were caused by either large expressions over strings or repeated transformations between numerical and bit vector theories. A coverage analysis is not provided on the benchmark because to consistently identify a test case as safe, we need to be able to evaluate the entire state space of the program. Otherwise, the faulty assumption could be located in the unevaluated code region, causing an incorrect result. This was the case for one test that is discussed in detail below.

One test case that contains a violation was incorrectly classified by SWAT. To evaluate why we misclassified this case, the corresponding method under test is shown in Figure 4.15. In the test case, the $\mathrm{addition}$ method is wrapped by the main method, where m and n are initialized symbolically. After some checks, the values can only reach the addition method if they satisfy $0 < \mathrm{m,n} < 10000$. The value c is not symbolically tracked and is always initialized to 0. Hence the only way the branching condition in line 4 could be symbolically evaluated is if the implicit data flow between the recursion and the incrementation of the value c is recorded. SWAT has no functionality to track such implicit dataflow between symbolic and non-symbolic variables (i.e., n and c) and hence has no symbolic formula for the branch.

Without a symbolic model for implicit flows, it leads to a full exploration of the symbolic space without triggering the assert statement and a wrong classification.

```
0  public static int addition(int m, int n, int c) {
1      if (n == 0) {
2        return m;
3      }
4      if (c >= 150) {
5        assert false;
6      }
7      if (n > 0) {
8        return addition(m + 1, n - 1, ++c);
9      } else {
10       return addition(m - 1, n + 1, ++c);
11     }
12   }
```

Figure 4.15: Method taken from test case jdart-regression/addition01 from SV-Comp [15]. n and m are symbolic variables with $0 < n,m < 10000$. The test case is labeled *False* because the assertion in line 5 can be triggered.

To summarize, the evaluation places SWAT in the midfield of evaluated state-of-the-art tools achieving fourth place overall. However, the evaluation was only performed on a subset of the test cases, and while that highlights limitations, the evaluated test cases underline the system's performance. Considering the relative immaturity of our system and missing features such as symbolic peers, the evaluation indicates the potential of SWAT, a loosely coupled instrumentation-based symbolic execution engine, being the best-ranking instrumentation-based symbolic execution engine.

5

Vulnerability Detection

The previous chapter has introduced the architecture and different modules of SWAT and demonstrated the potential of a loosely coupled instrumentation-based symbolic execution engine to find and detect violations in a benchmark designed to compare software verification tools. This chapter explores the applicability of SWAT on web services for detecting vulnerabilities.

5.1 Identifying Vulnerabilities

Symbolic execution provides the possibility to systematically explore the state space of a program while generating variable assignments that correspond to each explored branch of the execution tree. Hence symbolic execution is well-suited as a carrier for evaluating the program's properties under test. Unless symbolic execution is used for self-validating properties such as the existence or absence of crashes, additional functionality must be added to evaluate the properties under test. For example, in the SV-Comp [15] benchmark (Section 4.6) that was previously discussed, additional assert statements were added to the source code. Asserts are compiled as branching instructions that throw an exception depending on the value. These statements allow symbolic execution to both symbolically check the property (as it is compiled as a standard branch) and see if it is violated based on whether an exception is thrown. While a similar approach might be possible for detecting vulnerabilities, it would require the manual addition of comparable statements into the application's source code. However, SWAT does not require access to the source code of an application and is purposefully designed to require little manual labor to use the system. This chapter explains how SWAT can automatically detect vulnerable statements and dynamically evaluate if a vulnerability is present. In addition, the effectiveness of SWAT in finding vulnerabilities is evaluated using a state-of-the-art benchmark maintained by OWASP [51] for comparing different vulnerability detection tools.

The types of vulnerabilities present in code can vary significantly depending on the type of code and the interfaces exposed to potentially malicious actors. Hence the type of handling required to detect a set of vulnerabilities also differs. Vulnerabilities typically associated with compiled languages like C are not of concern for JVM-based languages be-

cause no direct memory management by the code is possible. Hence, SWAT does not consider memory safety vulnerabilities like stack- or buffer overflows. Instead, we focus on vulnerabilities typically present in web applications because SWAT is geared towards web services. In particular, we focus on injection attacks, including SQL injection, cross-site scripting attacks, and command injection attacks, as described in Section 2.3. OWASP ranks injection attacks among the top ten application security risks [53].

Injection vulnerabilities are data flow dependent and typically rely on *source-to-sink* flow of data. Sources include method calls that load user-controlled data into the program, and sinks are methods that execute some functionality vulnerable to malicious input. One example of an SQL injection is shown in Figure 2.9. In the scope of this method, line 4 is considered the source because it loads a user-controlled header value into the context of the method. Line 8 contains a method call that executes an SQL statement and is considered the sink because if a malicious user controls the `sql` string, an SQL injection could be performed. The nature of injection attacks tailors well to symbolic execution, primarily because an injection attack contains a user-controlled input that can execute some unintended functionality. Because we consider method calls as sources that introduce user-controlled data into the program, we can mark these values as symbolic values. We can either reuse the custom adapters introduced in Section 4.4 for popular frameworks or mark particular objects or methods as symbolic, as described below. One possibility used in Jaint [49] is to add taint tracking capabilities on top of the symbolic execution to observe if a tainted value propagates to a sink. SWAT uses a similar approach; however, we utilize symbolic execution as the taint propagation engine. When a sink is reached, we check if it contains a symbolic value, and if that is the case, an injection vulnerability has been found.

Source detection

Correctly identifying potential sources for injection attacks is vital to achieving good detection results. However, what methods should be considered sources depends on the analyzed target. Hence we allow the user to specify a list of regular expressions or concrete methods that should be tracked as sources. SWAT then uses the functionality described in Section 4.4 to set a flag during instrumentation to each method call that matches the user's specification. Because SWAT offers symbolic value propagation even on un-instrumented objects, for the OWASP benchmark [51] used below for evaluation, it is sufficient to specify the following regular expression: `javax/servlet/http/HttpServletRequest.*(*)*`. This regular expression matches all methods (first wildcard) from the class `HttpServletReques` with all signatures (second wildcard) and all return values (third wildcard). To, for example, only match methods that return a string, the third wildcard could be replaced with `Ljava/lang/String;`. Without symbolic propagation, the user would have to specify every method that returns a primitive value under the user's control. Specifying all such methods would increase the risk of missing some methods and is a tedious process.

Sink detection

Depending on the type of injection attack, different methods need to be considered as possible sinks. What exact methods need to be considered is also dependent on the library that is used to, for example, communicate with a database. While we allow the user to specify all methods, as usual, using either the full name or a regular expression, we also include a comprehensive list of vulnerable sinks. To demonstrate versatility, we use a list[6] provided by FindSecBugs [56]. For each sink, we evaluate the parameters passed into the sink for symbolic variables. If a sink is reached and a parameter contains a free variable, it is considered vulnerable. Our assumption aligns with OWASP's definition of injection vulnerabilities, where if any user-controlled value reaches a sink, it is considered vulnerable. We add an additional instrumentation pass to detect sinks that check for all methods matching the specification. We must evaluate each parameter regarding its symbolic status if such a method is found. To evaluate the parameters, we insert a custom method call before each sink that expects the same arguments and is responsible for evaluating the symbolic variables. This method is added during instrumentation and requires duplication of the parameters of the original method. However, we cannot simply duplicate all method parameters because they have an arbitrary depth on the stack that we cannot reach. To achieve the duplication, we evaluate free slots in the locals during instrumentation and temporarily store all parameters in the locals to retrieve them twice.

5.2 Evaluation

To evaluate the effectiveness of SWAT at correctly identifying vulnerabilities, we use a benchmark developed by OWASP [51]. The benchmark is a runnable web application written in Java. The app contains a total of 2740 endpoints spanning eleven types of vulnerabilities. Each endpoint either contains one vulnerability or none. As this thesis's scope is limited to injection attacks, a total of 1572 cases across six categories remain for evaluation. We further remove test cases from the LDAP injection category because the test cases are corrupt. Due to version issues, the benchmark is unable to instantiate the LDAP connection. A detailed overview of the categories we used and their test cases can be seen in Table 5.1. Overall, 1513 test cases from five categories were used for evaluation. From each category, all test cases are used for evaluation. The benchmark is designed to provide a test suite that can be used to evaluate vulnerability detection tools on real-world issues represented in small self-contained test cases. The suite supports analysis by SAST, DAST, or IAST tools that support Java.

[6] https://github.com/find-sec-bugs/find-sec-bugs/tree/99814871f33ca0484b975f2fe51bae2bc1bcf40a/plugin/src/main/resources/injection-sinks

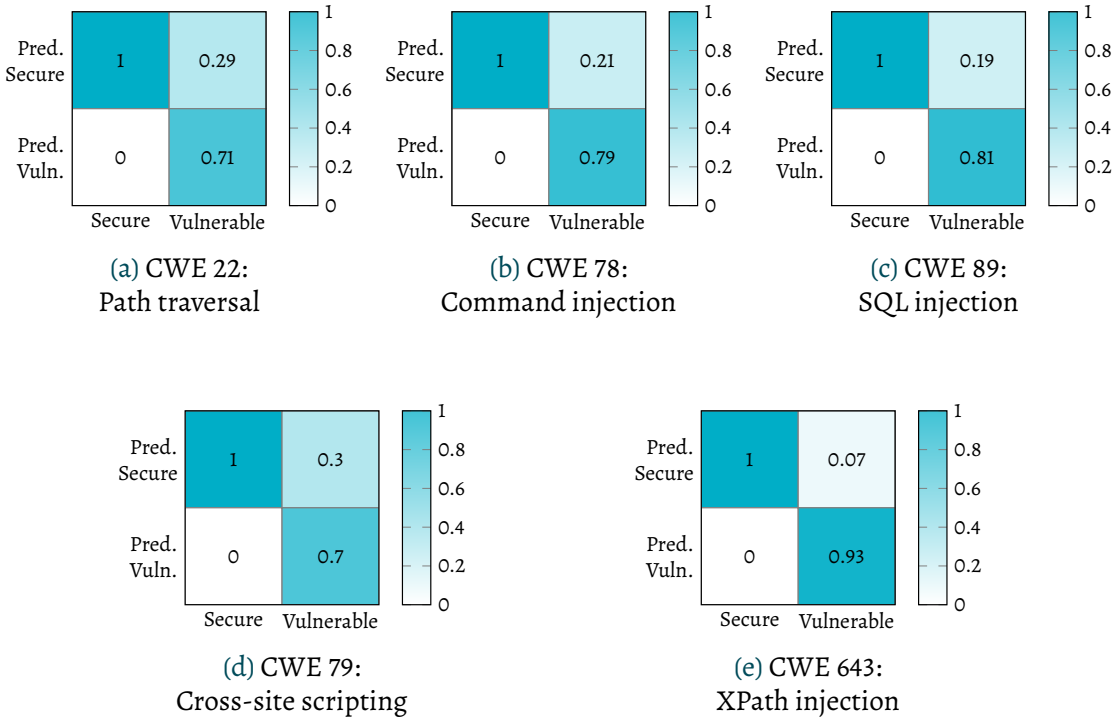
Table 5.1: Overview of number of test cases per category in the OWASP Benchmark [51]. Only the five categories used for evaluation are listed.

Name	CWE	Total	Vulnerable	Secure
Path traversal	22	268	133	135
Command injection	78	251	126	125
SQL injection	89	504	272	232
Cross-site scripting	79	455	246	209
XPath injection	643	35	15	20
Total	-	1513	792	721

For the OWASP evaluation, we used a first-generation Apple M1 MacBook Pro (A2338) with 8 GB of RAM running macOS 13.0 beta. We ran the Benchmark using the OpenJDK Runtime Environment (build 16+36-2231). Using the crawler provided by OWASP, SWAT can evaluate the entire test suite on average in 30 seconds over three runs. To put the required time into perspective, we compare our runtime to Jaint [49]. Jaint has a similar design but builds on top of SPF [55], so it utilizes the JPF-JVM [31]. Their artifacts are not public, so we cannot evaluate the two systems directly. However, Mues et al. report an average runtime of 1879 seconds on an Intel Core i9-7960X machine with 128 GB RAM. The difference in runtimes highlights the potential of SWAT in the context of vulnerability detection in web applications Especially considering that SWAT provides a simple deployment, as it runs in the native web application provided by OWASP without symbolic peers or a symbolic harness. To attach SWAT to the benchmark, we update the runtime configuration of the provided build file and add the location of the packaged engine as a JVM parameter. Additionally, we provide the configuration file.

Figure 5.2 visualizes confusion matrices for each category, to evaluate the accuracy of SWAT at detecting the vulnerabilities of different categories. The first column of every category shows that no test cases were falsely labeled as vulnerable, implying that SWAT has a precision of 1.0 or every test case reported as vulnerable is vulnerable. The second column reveals that SWAT can detect between 70% and 93% of the vulnerable cases with an average (arithmetic mean) detection rate of 75.81%.

We evaluated five open-source static vulnerability scanners on the same classification tasks, to put SWAT's performance in perspective. We used the presented scanners because they are provide run configurations for the benchmark and are supported by OWASP's evaluation. While other tools, especially IAST frameworks, would be interesting for comparison the ones that provide configurations require a licence.



(a) CWE 22: Path traversal

(b) CWE 78: Command injection

(c) CWE 89: SQL injection

(d) CWE 79: Cross-site scripting

(e) CWE 643: XPath injection

Figure 5.2: Evaluation of SWAT on 1513 injection vulnerabilities in OWASP benchmark test suite. Results are grouped by type of vulnerability as differentiated by the common weakness enumeration system (CWE) [47]. For each vulnerability type, a confusion matrix is plotted that shows the relative number of correctly or incorrectly classified samples into either vulnerable or Secure. Each matrix's first column contains all Secure test cases, and the second column contains vulnerable test cases. The rows summarize the cases classified as Secure (first row) and vulnerable (second row). The percentage of correctly classified samples (per class) can be read along the diagonal.

We first plot the results in the same format the OWASP benchmark presents. Figure 5.3 plots the true positive rate against the false positive rate. We achieve a recall of 0.76 and a false positive rate of 0.0. The further away a tool is from the red dashed line, the better the precision. All points below the line perform worse than guessing, while points above indicate better performance. Tools located closer to the origin, such as Insider [34], evaluate most cases as benign. Hence both true and false positive rates are low. Tools closer to the top right end of the spectrum, such as ShiftLeft Scan [62] and FindSecBugs [56], classify many cases as vulnerable, resulting in both a high true positive rate and a high false positive rate. SWAT can clearly outperform the presented tools with a high true positive and low false positive rate. However, Mues et al. report a perfect precision of 1.0 for Jaint [49]. Jaint is not included in the plot because we cannot evaluate the results as the tool has no public artifacts.

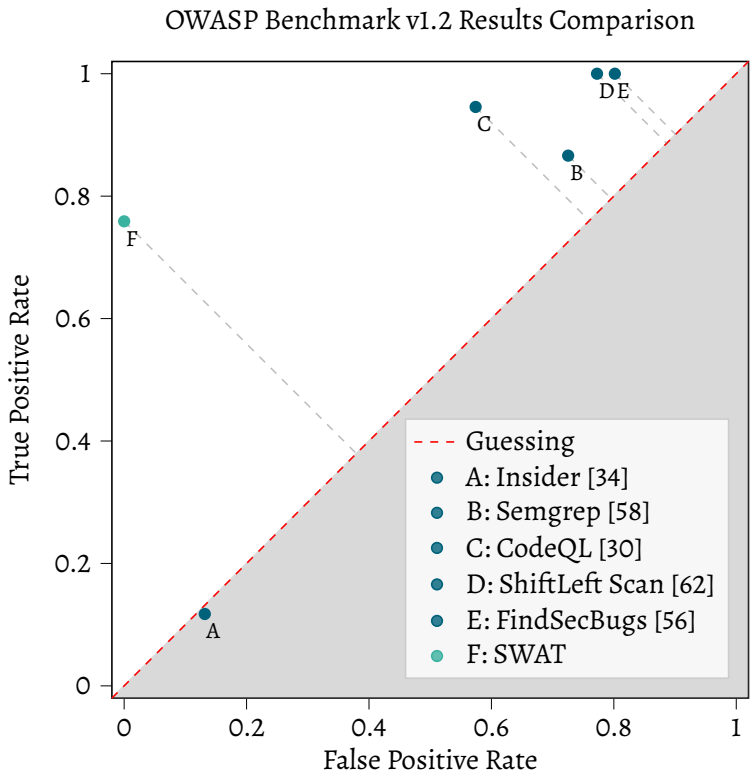


Figure 5.3: Evaluation adapted from the OWASP Benchmark evaluation. Each tool is evaluated regarding its true positive rate and false positive rate. Tools residing below the dashed line performed worse than guessing. The best-performing tools are located at the top left of the plot, where the true positive rate is maximized, and the false positive rate minimized. Evaluation is performed on all injection attack vulnerabilities, excluding LDAP injection attacks, because the benchmarks LDAP connections are erroneous.

True and false positive rates only paint a part of the overall picture. The F1 score combines recall and precision into a single metric by calculating their harmonic mean. An F1 score of one indicates perfect precision and recall, while an F1 score of zero indicates that both recall and precision are zero. F1 scores can be used to compare the performance of two classifiers and is more expressive than accuracy, especially if the dataset is not balanced. The F1 scores for the evaluated tools are plotted in Figure 5.4. Insider [34] performs the worst with an F1 score of roughly 0.2. Semgrep [58], CodeQL [30], ShiftLeft Scan [62], and FindSecBugs [56] all score between 0.69 and 0.77 while SWAT achieves an F1 score of 0.86 outperforming the evaluated tools. In general, we see static vulnerability scanners, such as the ones evaluated here, to perform better on vulnerability classes that do not contain data-flow dependent behaviour. Misconfigurations, for example tend to have higher scores on static tools. Mues et al. did not publish additional classification data; hence, no F1 score is available for Jaint [49].

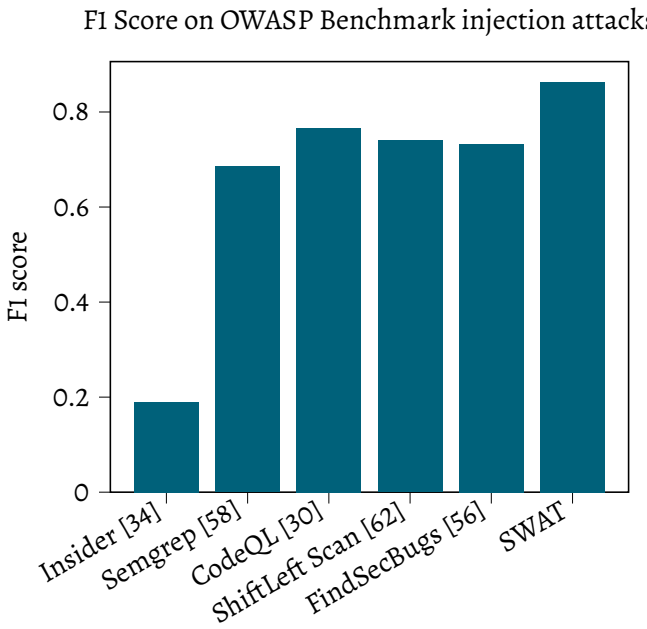


Figure 5.4: F1 scores of different vulnerability scanners and SWAT on the injection attacks present in the OWASP Benchmark evaluation. LDAP injection attacks are excluded because the benchmark's LDAP connections are erroneous.

Overall, SWAT outperforms the evaluated SAST tools on both the F1 score and OWASP's evaluation while achieving comparable runtime to static scanners on the OWASP benchmark. Compared to Jaint, a similar IAST approach, SWAT has slightly lower performance while reducing the runtime by a factor of 60 on a less capable processor. While the evaluation was only performed on a subset of vulnerabilities, the results highlight the potential of SWAT as a loosely coupled instrumentation-based symbolic execution engine.

5.3 Transferability to Real World Applications

The evaluations presented in this thesis allow for a quantification of SWAT's effectiveness. However, both the SV-Comp [15] and OWASP benchmark [51] are synthetic benchmarks designed to provide quantitative results that can be used to classify the performance of the evaluated tools. The vulnerabilities present in the OWASP benchmark are based on observations in real-world applications to maximize the transferability of the results to the performance on real applications. Still, the individual tests are reduced to small cases showcasing the provided vulnerability.

However, SWAT is designed to be applicable to web service applications that run in their native (cloud) configuration where all required external services are available. The symbolic executor provides configurable symbolic adapters to integrate SWAT into projects using a supported framework automatically. We enable symbolic execution while the application is managed through the underlying web server. In particular, we replace the

need for an active harness by utilizing the request handling to initiate symbolic execution. To initialize requests, we require an external component to identify all exposed endpoints and automatically build the correct requests with all required fields. The symbolic explorer could be used to fulfill that task, but this requires manual specification of the endpoints and their format. To prevent manual specification, the external component would need to be able to identify the endpoints and their structure automatically. Automatic detection and request generation is available as part of the RESTler fuzzer [4], which utilizes the OpenAPI specification [65]. By integrating a module into the fuzzer that facilitates the output of the symbolic execution as a heuristic in the mutation process of the fuzzer, we can evaluate large-scale applications without manual specification of endpoints and request formats. The RESTler integration is developed, in parallel to this thesis, by Florian Sieck, Institute for IT-Security. To facilitate pure symbolic execution, one could build an external component similar to RESTler, but without the fuzzing core. This was not done as part of this thesis because we aim to integrate and facilitate symbolic execution-guided fuzzing in the future.

By combining a request generator, such as RESTler, and our symbolic engine, we will be able to automatically test web applications that rely on a supported framework such as Spring [68] or javax servlets. By combining fuzzing and symbolic execution, we hope to benefit from the efficiency of fuzzing and the effectiveness of symbolic execution to build a practically effective and efficient system for identifying real vulnerabilities in large-scale web applications. To measure the effectiveness of large applications, we can record instruction or branch coverage during symbolic and fuzzing execution. As indicated by the SV-Comp benchmark, we expect SWAT to be able to effectively evaluate a large portion of the target's state space. While limiting factors to the achievable coverage exist (Section 4.6), by providing symbolic capabilities for all primitive data types and strings, we aim to cover most branches that are affected by the behavior of the symbolic inputs.

6

Conclusion and Outlook

In this thesis, we introduced SWAT, a symbolic web application testing platform. SWAT is an instrumentation-based dynamic symbolic execution engine based on CATG [64] that is designed around a loosely coupled architecture. The symbolic executor and symbolic explorer are individual modules that communicate through HTTP requests. Our new symbolic backend uses JavaSMT [6] to build constraints in the standardized SMT-Lib format [9], allowing solver interchangeability. We used numerical theories to support all primitive datatypes symbolically, including correct overflow modeling and symbolic exception modeling for numerical exceptions. In addition to symbolic support for all primitive datatypes, SWAT provides symbolic peers for Java's built-in classes, such as the String or Integer class. Building on recent advancements in the Z3 [26] solver, we can model string operations and arrays using SMT-Lib's respective theories, significantly increasing symbolic string and array support. Using an improved instrumentation core, SWAT can symbolically model newer language features, such as lambda expressions, while optimizing the instrumentation footprint. To improve the usability of SWAT, we additionally introduced several adapters to initialize symbolic handling and determine symbolic variables automatically. The symbolic executor features a thread-specific symbolic expressions store and path constraints to enable parallel execution of threads. Traces sent between the symbolic executor and explorer are serialized using JSON and sent via an API to allow easy future interoperability between different executors and explorers. This thesis aimed to utilize symbolic execution to effectively and efficiently identify injection vulnerabilities in web applications. Building upon SWAT, we have introduced functionality to detect *source-to-sink* data flow from user-specified sources to sinks. Using additional instrumentation passes, we allow the user to specify what methods should be considered sources or sinks and automatically add the required handling. We demonstrated that, for the OWASP benchmark [51], we only need to specify a single regular expression to cover all sources and can import a list from FindSecBugs [56] without modification to track the sinks. In addition, to evaluate SWAT on the OWASP benchmark, we only need to specify the configuration file and add a parameter to the build file of the benchmark. No symbolic harnesses, drivers, or peers need to be implemented.

To unlock SWAT's full potential, we aim to increase the number of supported methods from Javas built-in classes. The remaining methods primarily present an engineering ef-

fort planned for the future. In the meantime, SWAT reports each method it encounters with no symbolic peer to the user to make the limitation transparent and allow targeted implementation of the next peers. When SWAT tracks multiple threads symbolically that interact with each other, the symbolic state is not appropriately advanced. For our target scenario, where multiple independent requests may be executed in several threads, general multi-threading support is vital, whereas, for example, lock detection between threads is not a major focus. To further extend the capabilities of SWAT, we aim to extend implicit information flow tracking in loops or recursions from the concrete model into our symbolic model. This will allow SWAT to test a target more thoroughly, increasing the coverage and likelihood of finding injection vulnerabilities. In the future, we plan to extend the capabilities to other types of vulnerabilities in a similar way to Jaint [49] yet without explicitly building a tainting engine. Instead, we aim to facilitate the required capabilities directly in the symbolic execution. Vulnerability classes, such as misconfigurations of cryptographic libraries, may be detectable by catching the respective initializations and evaluating whether the parameters provide a valid configuration. Evaluating variables would allow validation of both concrete values and symbolic variables.

We have evaluated the symbolic capabilities of SWAT by comparing our system against state-of-the-art verification tools for Java on a subset of the SV-Comp benchmark [15], where SWAT placed fourth overall and second behind JDart [43] comparing symbolic execution engines. In the evaluation, SWAT outperformed COASTAL [36], the other instrumentation-based symbolic execution engine. A wider variety of symbolic peers alongside implicit symbolic information flow would allow SWAT to achieve a higher score on the SV-Comp competition and expand the evaluation to more test cases from the competition. The SV-Comp evaluation underlines the potential of SWAT, especially under consideration of the current limitations. The evaluation on the injection category of the OWASP benchmark [51] highlights the effectiveness of SWAT, achieving an F1 score of 0.86, outperforming all tested tools. In addition, SWAT achieves perfect precision and a slightly lower true positive rate than reported by Jaint [49]. However, the runtime on the benchmark is reduced by a factor of 60. Overall we have shown the potential of SWAT as a practically effective vulnerability detection system designed to operate on web service architectures.

Bibliography

[1] Alhuzali, A., Gjomemo, R., Eshete, B., and Venkatakrishnan, V. N. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, pp. 377–392. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali.

[2] Arnold, K. and Gosling, J. *The Java Programming Language*. Addison-Wesley, 1996. ISBN: 0-201-63455-4.

[3] Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., and Ray, B. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. In: *CoRR* abs/2005.11498, 2020. arXiv: 2005.11498. URL: https://arxiv.org/abs/2005.11498.

[4] Atlidakis, V., Godefroid, P., and Polishchuk, M. RESTler: stateful REST API fuzzing. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by J. M. Atlee, T. Bultan, and J. Whittle. IEEE, 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083. URL: https://doi.org/10.1109/ICSE.2019.00083.

[5] Avgerinos, T., Rebert, A., Cha, S. K., and Brumley, D. Enhancing symbolic execution with veritesting. In: *Commun. ACM* 59(6):93–100, 2016. DOI: 10.1145/2927924. URL: https://doi.org/10.1145/2927924.

[6] Baier, D., Beyer, D., and Friedberger, K. JavaSMT 3: Interacting with SMT Solvers in Java. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 195–208. DOI: 10.1007/978-3-030-81688-9_9. URL: https://doi.org/10.1007/978-3-030-81688-9%5C_9.

[7] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A Survey of Symbolic Execution Techniques. In: *ACM Comput. Surv.* 51(3):50:1–50:39, 2018. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[8] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al. cvc5: A Versatile and Industrial-Strength SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by D. Fisman and G. Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24. URL: https://doi.org/10.1007/978-3-030-99524-9%5C_24.

[9] Barrett, C., Stump, A., and Tinelli, C. The SMT-LIB Standard: Version 2.0. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[10] Barrett, C. W., Barbosa, H., Brain, M., Ibeling, D., King, T., Meng, P., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., et al. CVC4 at the SMT Competition 2018. In: *CoRR* abs/1806.08775, 2018. arXiv: 1806.08775. URL: http://arxiv.org/abs/1806.08775.

[11] Barrett, C. W., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_14.

[12] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In: *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005, pp. 41–46. URL: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

[13] Beyer, D. Progress on Software Verification: SV-COMP 2022. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Ed. by D. Fisman and G. Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20. URL: https://doi.org/10.1007/978-3-030-99527-0%5C_20.

[14] Beyer, D. Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015). In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 401–416. DOI: 10.1007/978-3-662-46681-0_31. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_31.

[15] Beyer, D. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by J. F. Groote and K. G. Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24. URL: https://doi.org/10.1007/978-3-030-72013-1%5C_24.

[16] Beyer, D., Löwe, S., and Wendler, P. Reliable benchmarking: requirements and solutions. In: *Int. J. Softw. Tools Technol. Transf.* 21(1):1–29, 2019. DOI: 10.1007/s10009-017-0469-y. URL: https://doi.org/10.1007/s10009-017-0469-y.

[17] Binder, W., Hulaas, J., and Moret, P. Advanced Java bytecode instrumentation. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007, Lisboa, Portugal, September 5-7, 2007*. Ed. by V. Amaral, L. Marcelino, L. Veiga, and H. C. Cunningham. Vol. 272. ACM International Con-

ference Proceeding Series. ACM, 2007, pp. 135–144. DOI: 10.1145/1294325.1294344. URL: https://doi.org/10.1145/1294325.1294344.

[18] Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Kirda, E., and Egele, M. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/.

[19] Boute, R. T. The Euclidean Definition of the Functions Div and Mod. In: *ACM Trans. Program. Lang. Syst.* 14(2):127–144, Apr. 1992. ISSN: 0164-0925. DOI: 10.1145/128861.128862. URL: https://doi.org/10.1145/128861.128862.

[20] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: A code manipulation tool to implement adaptable systems. In: *In Adaptable and extensible component systems*. 2002.

[21] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by R. Draves and R. van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[22] Chen, P. and Chen, H. Angora: Efficient Fuzzing by Principled Search. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046. URL: https://doi.org/10.1109/SP.2018.00046.

[23] Cook, S. A. The Complexity of Theorem-Proving Procedures. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by M. A. Harrison, R. B. Banerji, and J. D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[24] Cordeiro, L. C., Kesseli, P., Kroening, D., Schrammel, P., and Trtík, M. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by H. Chockler and G. Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 183–190. DOI: 10.1007/978-3-319-96145-3_10. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_10.

[25] Dahm, M. Byte Code Engineering. In: *JIT '99, Java-Informations-Tage 1999, Düsseldorf 20./21. September 1999*. Ed. by C. H. Cap. Informatik Aktuell. Springer, 1999, pp. 267–277. DOI: 10.1007/978-3-642-60247-4_25. URL: https://doi.org/10.1007/978-3-642-60247-4%5C_25.

[26] De Moura, L. and Bjørner, N. Z3: An Efficient SMT Solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[27] Edalat, E., Sadeghiyan, B., and Ghassemi, F. ConsiDroid: A Concolic-based Tool for Detecting SQL Injection Vulnerability in Android Apps. In: *CoRR* abs/1811.10448, 2018. arXiv: 1811.10448. URL: http://arxiv.org/abs/1811.10448.

[28] Fraser, G. and Arcuri, A. EvoSuite: automatic test suite generation for object-oriented software. In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. Ed. by T. Gyimóthy and A. Zeller. ACM, 2011, pp. 416–419. DOI: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179.

[29] Fu, X. and Qian, K. SAFELI: SQL injection scanner using symbolic execution. In: *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), TAV-WEB 2008, Seattle, Washington, USA, July 21, 2008*. Ed. by T. Bultan and T. Xie. ACM, 2008, pp. 34–39. DOI: 10.1145/1390832.1390838. URL: https://doi.org/10.1145/1390832.1390838.

[30] GitHub *CodeQL*. https://codeql.github.com. 2022, Accessed: 2022.

[31] Havelund, K. and Pressburger, T. Model Checking JAVA Programs using JAVA PathFinder. In: *Int. J. Softw. Tools Technol. Transf.* 2(4):366–381, 2000. DOI: 10.1007/s100090050043. URL: https://doi.org/10.1007/s100090050043.

[32] He, J., Sivanrupan, G., Tsankov, P., and Vechev, M. T. Learning to Explore Paths for Symbolic Execution. In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Y. Kim, J. Kim, G. Vigna, and E. Shi. ACM, 2021, pp. 2526–2540. DOI: 10.1145/3460120.3484813. URL: https://doi.org/10.1145/3460120.3484813.

[33] Howar, F., Jabbour, F., and Mues, M. JConstraints: A Library for Working with Logic Expressions in Java. In: *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*. Ed. by T. Margaria, S. Graf, and K. G. Larsen. Vol. 11200. Lecture Notes in Computer Science. Springer, 2018, pp. 310–325. DOI: 10.1007/978-3-030-22348-9_19. URL: https://doi.org/10.1007/978-3-030-22348-9%5C_19.

[34] InsiderSec *Insider*. https://github.com/insidersec/insider. 2021, Accessed: 2022.

[35] Islam, M. and Csallner, C. Dsc+Mock: a test case + mock class generator in support of coding against interfaces. In: *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2010), WODA 2010, Trento, Italy, July 12, 2010*. Ed. by J. Cook and J. A. Jones. ACM, 2010, pp. 26–31. DOI: 10.1145/1868321.1868326. URL: https://doi.org/10.1145/1868321.1868326.

[36] Jaco Geldenhuys and Willem Visser *COASTAL: A Java program analysis tool built on concolic execution and fuzz testing*. https://deepseaplatform.github.io/coastal/. 2019, Accessed: 2022.

[37] Kahsai, T., Rümmer, P., Sanchez, H., and Schäf, M. JayHorn: A Framework for Verifying Java programs. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Ed. by S. Chaud-

huri and A. Farzan. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 352–358. DOI: 10.1007/978-3-319-41528-4_19. URL: https://doi.org/10.1007/978-3-319-41528-4%5C_19.

[38] Kersten, R., Luckow, K. S., and Pasareanu, C. S. POSTER: AFL-based Fuzzing for Java with Kelinci. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM, 2017, pp. 2511–2513. DOI: 10.1145/3133956.3138820. URL: https://doi.org/10.1145/3133956.3138820.

[39] Lattner, C. and Adve, V. S. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665. URL: https://doi.org/10.1109/CGO.2004.1281665.

[40] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. *The Java virtual machine specification*. Pearson Education, 2014.

[41] Livshits, V. B. and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In: *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. Ed. by P. D. McDaniel. USENIX Association, 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

[42] LLVM *Undefined behavior sanitizer*. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. 2021, Accessed: 2022.

[43] Luckow, K. S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., and Raman, V. JDart: A Dynamic Symbolic Analysis Framework. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by M. Chechik and J. Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 442–459. DOI: 10.1007/978-3-662-49674-9_26. URL: https://doi.org/10.1007/978-3-662-49674-9%5C_26.

[44] Luk, C., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K. M. Pin: building customized program analysis tools with dynamic instrumentation. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by V. Sarkar and M. W. Hall. ACM, 2005, pp. 190–200. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

[45] Man, H., An, J., Huang, W., and Fan, W. JSEFuzz: Vulnerability Detection Method for Java Web Application. In: *3rd International Conference on System Reliability and Safety, ICSRS 2018, Barcelona, Spain, November 23-25, 2018*. IEEE, 2018, pp. 92–96. DOI: 10.1109/ICSRS.2018.8688844. URL: https://doi.org/10.1109/ICSRS.2018.8688844.

[46] Michal Zalewski *American fuzzy lop*. https://lcamtuf.coredump.cx/afl/. 2022, Accessed: 2022.

[47] MITRE *Common Weakness Enumerations*. https://cwe.mitre.org/index.html. 2022, Accessed: 2022.

[48] Mues, M. and Howar, F. GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Ed. by D. Fisman and G. Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 435–439. DOI: 10.1007/978-3-030-99527-0_27. URL: https://doi.org/10.1007/978-3-030-99527-0%5C_27.

[49] Mues, M., Schallau, T., and Howar, F. Jaint: A Framework for User-Defined Dynamic Taint-Analyses based on Dynamic Symbolic Execution of Java Programs. In: *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*. Ed. by A. Koziolek, I. Schaefer, and C. Seidl. Vol. P-310. LNI. Gesellschaft für Informatik e.V., 2021, pp. 77–78. DOI: 10.18420/SE2021_27. URL: https://doi.org/10.18420/SE2021%5C_27.

[50] Oracle *Espresso: A meta-circular Java bytecode interpreter for the GraalVM*. https://github.com/oracle/graal/tree/master/espresso. 2022, Accessed: 2022.

[51] OWASP *OWASP Benchmark*. https://owasp.org/www-project-benchmark/#div-scoring. 2022, Accessed: 2022.

[52] OWASP *OWASP WebGoat*. https://owasp.org/www-project-webgoat/. 2022, Accessed: 2022.

[53] OWASP *Top 10 Web Application Security Risks*. https://owasp.org/www-project-top-ten/. 2021, Accessed: 2022.

[54] Pasareanu, C. S. and Rungta, N. Symbolic PathFinder: symbolic execution of Java bytecode. In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. Ed. by C. Pecheur, J. Andrews, and E. D. Nitto. ACM, 2010, pp. 179–180. DOI: 10.1145/1858996.1859035. URL: https://doi.org/10.1145/1858996.1859035.

[55] Pasareanu, C. S. and Rungta, N. Symbolic PathFinder: symbolic execution of Java bytecode. In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. Ed. by C. Pecheur, J. Andrews, and E. D. Nitto. ACM, 2010, pp. 179–180. DOI: 10.1145/1858996.1859035. URL: https://doi.org/10.1145/1858996.1859035.

[56] Philippe Arteau *FindSecBugs: Find Security Bugs*. https://find-sec-bugs.github.io. 2022, Accessed: 2022.

[57] Poeplau, S. and Francillon, A. SymQEMU: Compilation-based symbolic execution for binaries. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/.

[58] r2c *CodeQL*. https://semgrep.dev. 2022, Accessed: 2022.

[59] Rial, I. M. L. and Galeotti, J. P. EvoSuiteDSE at the SBST 2021 Tool Competition. In: *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021,*

Madrid, Spain, May 31, 2021. IEEE, 2021, pp. 30–31. DOI: 10.1109/SBST52555.2021.00013. URL: https://doi.org/10.1109/SBST52555.2021.00013.

[60] Ruaro, N., Zeng, K., Dresel, L., Polino, M., Bao, T., Continella, A., Zanero, S., Kruegel, C., and Vigna, G. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In: *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*. Ed. by L. Bilge and T. Dumitras. ACM, 2021, pp. 456–468. DOI: 10.1145/3471621.3471865. URL: https://doi.org/10.1145/3471621.3471865.

[61] Sharma, V., Hussein, S., Whalen, M. W., McCamant, S., and Visser, W. Java Ranger: statically summarizing regions for efficient symbolic execution of Java. In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by P. Devanbu, M. B. Cohen, and T. Zimmermann. ACM, 2020, pp. 123–134. DOI: 10.1145/3368089.3409734. URL: https://doi.org/10.1145/3368089.3409734.

[62] ShiftLeftSecurity *ShiftLeft Scan*. https://github.com/ShiftLeftSecurity/sast-scan. 2022, Accessed: 2022.

[63] Statista *Most used programming languages*. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. 2021, Accessed: 2022.

[64] Tanno, H., Zhang, X., Hoshino, T., and Sen, K. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. Ed. by A. Bertolino, G. Canfora, and S. G. Elbaum. IEEE Computer Society, 2015, pp. 717–720. DOI: 10.1109/ICSE.2015.231. URL: https://doi.org/10.1109/ICSE.2015.231.

[65] The Linux Foundation *OpenAPI Initiative*. https://www.openapis.org/. 2022, Accessed: 2022.

[66] Thomas Eisenbarth *CS4701 CoSyS: Advanced Code Analysis: Symbolic Execution*. 2021.

[67] Vallee-Rai, R. and Hendren, L. J. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

[68] VMware Tanzu *Spring*. https://spring.io. 2022, Accessed: 2022.

[69] Wang, F. and Shoshitaishvili, Y. Angr - The Next Generation of Binary Analysis. In: *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. IEEE Computer Society, 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14. URL: https://doi.org/10.1109/SecDev.2017.14.

[70] Watkins, C. J. C. H. and Dayan, P. Technical Note Q-Learning. In: *Mach. Learn.* 8:279–292, 1992. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[71] Wu, J., Zhang, C., and Pu, G. Reinforcement Learning Guided Symbolic Execution. In: *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. Ed. by K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou. IEEE, 2020, pp. 662–663. DOI: 10.1109/SANER48275.2020.9054815. URL: https://doi.org/10.1109/SANER48275.2020.9054815.

[72] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. One VM to rule them all. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by A. L. Hosking, P. T. Eugster, and R. Hirschfeld. ACM, 2013, pp. 187–204. DOI: 10.1145/2509578.2509581. URL: https://doi.org/10.1145/2509578.2509581.

[73] Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association, 2018, pp. 745–761. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

A

Appendix

Detailed Byte code contributions

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode.

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
aaload	✓	×	×	Branch	✓
aastore	×	×	×	Branch	✓
aconst_null	-	-	-	-	-
aload	-	-	-	-	-
aload_0	-	-	-	-	-
aload_1	-	-	-	-	-
aload_2	-	-	-	-	-
aload_3	-	-	-	-	-
anewarray	×	×	×	Branch	✓
areturn	-	-	-	Special	×
arraylength	×	✓	×	Special	×
astore	-	-	-	-	-
astore_0	-	-	-	-	-
astore_1	-	-	-	-	-
astore_2	-	-	-	-	-
astore_3	-	-	-	-	-
athrow	-	-	-	Special	×
baload	✓	✓	×	Branch	✓

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
bastore	×	✓	×	Branch	✓
bipush	-	-	-	-	-
breakpoint	-	-	-	-	-
caload	✓	✓	×	Branch	✓
castore	×	✓	×	Branch	✓
checkcast	-	-	-	Special	×
d2f	×	✓	✓	-	-
d2i	×	✓	✓	-	-
d2l	×	✓	✓	-	-
dadd	×	✓	✓	-	-
daload	✓	✓	×	Branch	✓
dastore	×	✓	×	Branch	✓
dcmpg	×	✓	✓	-	-
dcmpl	×	✓	✓	-	-
dconst_0	-	-	-	-	-
dconst_1	-	-	-	-	-
ddiv	×	✓	✓	-	-
dload	-	-	-	-	-
dload_0	-	-	-	-	-
dload_1	-	-	-	-	-
dload_2	-	-	-	-	-
dload_3	-	-	-	-	-
dmul	×	✓	✓	-	-
dneg	×	✓	✓	-	-
drem	×	×	✓	-	-
dreturn	-	-	-	Special	×
dstore	-	-	-	-	-
dstore_0	-	-	-	-	-
dstore_1	-	-	-	-	-
dstore_2	-	-	-	-	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
dstore_3	-	-	-	-	-
dsub	×	✓	✓	-	-
dup	-	-	-	-	-
dup_x1	-	-	-	-	-
dup_x2	-	-	-	-	-
dup2	-	-	-	-	-
dup2_x1	-	-	-		-
dup2_x2	-	-	-	-	-
f2d	×	✓	✓	-	-
f2i	×	✓	✓	-	-
f2l	×	✓	✓	-	-
fadd	×	✓	✓	-	-
faload	✓	✓	×	Branch	✓
fastore	×	✓	×	Branch	✓
fcmpg	×	✓	✓	-	-
fcmpl	×	✓	✓	-	-
fconst_0	-	-	-	-	-
fconst_1	-	-	-	-	-
fconst_2	-	-	-	-	-
fdiv	×	✓	✓	-	-
fload	-	-	-	-	-
fload_0	-	-	-	-	-
fload_1	-	-	-	-	-
fload_2	-	-	-	-	-
fload_3	-	-	-	-	-
fmul	×	✓	✓	-	-
fneg	×	✓	✓	-	-
frem	×	×	✓	-	-
freturn	-	-	-	Special	×
fstore	-	-	-	-	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
fstore_0	-	-	-	-	-
fstore_1	-	-	-	-	-
fstore_2	-	-	-	-	-
fstore_3	-	-	-	-	-
fsub	×	✓	✓	-	-
getfield	-	-	-	-	-
getstatic	-	-	-	Special	×
goto	-	-	-	-	-
goto_w	-	-	-	-	-
i2b	✓	✓	✓	-	-
i2c	✓	✓	✓	-	-
i2d	×	✓	✓	-	-
i2f	×	✓	✓	-	-
i2l	✓	✓	✓	-	-
i2s	✓	✓	✓	-	-
iadd	✓	✓	✓	-	-
iaload	✓	✓	×	Branch	✓
iand	×	✓	✓	-	-
iastore	×	✓	×	Branch	✓
iconst_0	-	-	-	-	-
iconst_1	-	-	-	-	-
iconst_2	-	-	-	-	-
iconst_3	-	-	-	-	-
iconst_4	-	-	-	-	-
iconst_5	-	-	-	-	-
iconst_m1	-	-	-	-	-
idiv	×	✓	✓	Branch	✓
if_acmpeq	×	×	×	Branch	-
if_acmpne	×	×	×	Branch	-
if_icmpeq	✓	✓	✓	Branch	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
if_icmpge	✓	✓	✓	Branch	-
if_icmpgt	✓	✓	✓	Branch	-
if_icmple	✓	✓	✓	Branch	-
if_icmplt	✓	✓	✓	Branch	-
if_icmpne	✓	✓	✓	Branch	-
ifeq	✓	✓	✓	Branch	-
ifge	✓	✓	✓	Branch	-
ifgt	✓	✓	✓	Branch	-
ifle	✓	✓	✓	Branch	-
iflt	✓	✓	✓	Branch	-
ifne	✓	✓	✓	Branch	-
ifnonnull	✓	✓	×	Branch	-
ifnull	✓	✓	×	Branch	-
iinc	✓	✓	✓	-	-
iload	-	-	-	-	-
iload_0	-	-	-	-	-
iload_1	-	-	-	-	-
iload_2	-	-	-	-	-
iload_3	-	-	-	-	-
impdep1	-	-	-	-	-
impdep2	-	-	-	-	-
imul	✓	✓	✓	-	-
ineg	✓	✓	✓	-	-
instanceof	-	-	-	Special	×
invokedynamic	-	-	-	Special	×
invokeinterface	-	-	-	Special	×
invokespecial	-	-	-	Special	×
invokestatic	-	-	-	Special	×
invokevirtual	-	-	-	Special	×
ior	×	✓	✓	-	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
irem	×	✓	✓	Branch	✓
ireturn	-	-		Special	×
ishl	×	✓	✓	-	-
ishr	×	✓	✓	-	-
istore	-	-	-	-	-
istore_0	-	-	-	-	-
istore_1	-	-	-	-	-
istore_2	-	-	-	-	-
istore_3	-	-	-	-	-
isub	✓	✓	✓	-	-
iushr	×	✓	✓	-	-
ixor	×	✓	✓	-	-
jsr_w†	-	-	-	-	-
jsr†	-	-	-	-	-
l2d	×	✓	✓	-	-
l2f	×	✓	✓	-	-
l2i	✓	✓	✓	-	-
ladd	✓	✓	✓	-	-
laload	✓	✓	×	Branch	✓
land	×	✓	✓	-	-
lastore	×	✓	×	Branch	✓
lcmp	✓	✓	✓	-	-
lconst_0	-	-	-	-	-
lconst_1	-	-	-	-	-
ldc	-	-	-	Special	×
ldc_w	-	-	-	Special	×
ldc2_w	-	-	-	-	-
ldiv	×	✓	✓	Branch	✓
lload	-	-	-	-	-
lload_0	-	-	-	-	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
lload_1	-	-	-	-	-
lload_2	-	-	-	-	-
lload_3	-	-	-	-	-
lmul	✓	✓	✓	-	-
lneg	✓	✓	✓	-	-
lookupswitch	-	-	-	Branch	-
lor	×	✓	✓	-	-
lrem	×	✓	✓	Branch	✓
lreturn	-	-	-	-	-
lshl	×	✓	✓	-	-
lshr	×	✓	✓	-	-
lstore	-	-	-	-	-
lstore_0	-	-	-	-	-
lstore_1	-	-	-	-	-
lstore_2	-	-	-	-	-
lstore_3	-	-	-	-	-
lsub	✓	✓	✓	-	-
lushr	×	✓	✓	-	-
lxor	×	✓	✓	-	-
monitorenter	-	-	-	Special	×
monitorexit	-	-	-	Special	×
multianewarray	-	-	-	Branch	✓
new	-	-	-	Special	×
newarray	-	-	-	Branch	✓
nop	-	-	-	-	-
pop	-	-	-	-	-
pop2	-	-	-	-	-
putfield	-	-	-	Special	×
putstatic	-	-	-	Special	×
ret†	-	-	-	-	-

Continued on next page

Table A.1: Overview of symbolic capabilities for SWAT and CATG [64] per opcode. (Continued)

	CATG	SWAT			
	Symbolic	Symbolic	Tested	Trace	Sym. Exception
return	-	-	-	Special	×
saload	✓	✓	×	Branch	✓
sastore	×	✓	×	Branch	✓
sipush	-	-	-	-	-
swap	-	-	-	-	-
tableswitch	-	-	-	Branch	-
wide	-	-	-	-	-