Dynamic Symbolic Execution for Polymorphism

Lian Li[1,2]
[1]Oracle Labs, Australia
[2]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
lianli@ict.ac.cn

Yi Lu
Oracle Labs, Australia
yi.x.lu@oracle.com

Jingling Xue
UNSW Australia
jingling@cse.unsw.edu.au

Abstract

Symbolic execution is an important program analysis technique that provides auxiliary execution semantics to execute programs with symbolic rather than concrete values. There has been much recent interest in symbolic execution for automatic test case generation and security vulnerability detection, resulting in various tools being deployed in academia and industry. Nevertheless, (subtype or dynamic) polymorphism of object-oriented programAnalysiss has been neglected: existing symbolic execution techniques can explore different targets of conditional branches but not different targets of method invocations. We address the problem of how this polymorphism can be expressed in a symbolic execution framework. We propose the notion of *symbolic types*, which make object types symbolic. With symbolic types, *various targets of a method invocation can be explored systematically* by mutating the type of the receiver object of the method during automatic test case generation. To the best of our knowledge, this is the first attempt to address polymorphism in symbolic execution. Mutation of method invocation targets is critical for effectively testing object-oriented programs, especially libraries. Our experimental results show that symbolic types are significantly more effective than existing symbolic execution techniques in achieving test coverage and finding bugs and security vulnerabilities in `OpenJDK`.

Categories and Subject Descriptors D [2]: 5Symbolic Execution

Keywords Concolic Testing, Object-Oriented Programs

1. Introduction

Symbolic execution [26] is a powerful technique for automatic test case generation [11], especially when used for finding security vulnerabilities in complex software applications. Program inputs are represented with symbolic values, and program instructions are interpreted as operations that manipulate symbolic values. *Dynamic symbolic execution* (DSE) can be seen as auxiliary execution semantics of programs, in which a program is executed with concrete and symbolic input values. The concrete values induce a program execution, from which a *path condition* is generated using the symbolic values of the variables in the program. The path condition represents a set of constraints that must be satisfied by any concrete input that results in the same execution path.

The ability to automatically generate concrete test inputs from a path condition makes symbolic execution attractive for automated testing and bug detection. A new test case is generated from an existing one by *mutating* the path condition of that existing execution and then solving the mutated path condition with an off-the-shelf constraint solver. Recent research has demonstrated that symbolic execution techniques and test case generation can automatically generate test suites with high coverage [10, 54], and that these techniques are effective in finding deep security vulnerabilities in large and complex real-world software applications [10, 22].

This paper addresses (subtype or dynamic) polymorphism in symbolic execution. When analyzing object-oriented programs symbolically, it is critical to explore different targets of method invocations, i.e., the targets that depend on the types of their receiver objects. Mutating a constraint related to the type of an receiver object should therefore generate a set of new constraints encompassing all other possible types of the receiver object. However, such mutation has been overlooked. Existing symbolic execution techniques [16, 22, 54] for object-oriented programs do not generate such polymorphism-related constraints. While being capable of exploring different targets of conditional branches, where constraints on symbolic values are introduced for different branch conditions, the state-of-the-art techniques [16, 22, 54] cannot explore different targets of method invocations, which depend on the types rather than values of receiver objects. This limitation has largely limited their ability in generating test cases effectively for object-oriented programs.

To address this problem, we propose *object-oriented symbolic execution* (OO-SE), which leverages classic symbolic execution and extends it with *symbolic types*. A symbolic type represents all possible concrete types of a receiver object, thereby enabling its mutation in test case generation. During symbolic execution, constraints on symbolic types are added to the path condition and type-related operations such as casting, type testing (e.g., `instanceof` in Java™) and *virtual invocation* are tracked symbolically. As a result, we have enriched the expressiveness of path conditions, by considering not only the constraints on symbolic values as before but also the constraints on symbolic types as in this work.

Figure 1 depicts a test case generation framework, built by exploiting symbolic types, for object-oriented libraries. Our approach applies also to applications when their individual parts are separately tested. When testing an object-oriented library, the following three steps are repeated until a certain goal has been met:

- First, a concrete test input for an API method $M = c\ m(\overrightarrow{t\ x})$ $\{\ldots\}$ in the library is generated and a harness program is syn-

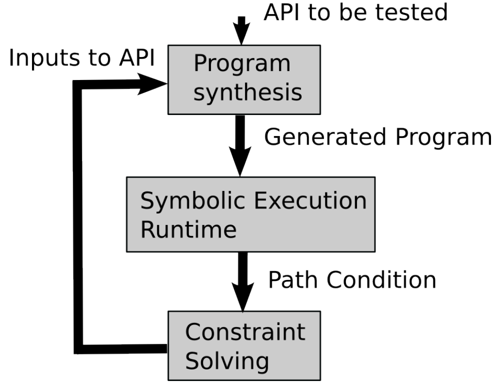

Figure 1. Architecture of a dynamic symbolic execution tool for object-oriented libraries by exploiting symbolic types.

thesized to invoke the API with the test input. When testing method m, we can assume symbolic inputs for its formal parameters as well as the formal parameters of the constructor called on its corresponding receiver object. Let x be such a formal parameter. Two cases are considered. If x is of a primitive type, then x has a symbolic (input) value as before [16, 22, 54]. If x is of a reference type, then x has a symbolic (input) type T, or more precisely, the (input) object o referenced by x has a symbolic type T. Note that o may be of any subtype of the declared type d of x, denoted $T \trianglelefteq d$. We write $l^T \mapsto o^c$ to represent the fact that o, which is constructed with a concrete type c, has a symbolic type T. Here, l^T is a reference to o, such that $x = l^T$. Thus, the program is executed with object o being of the concrete type c and with constraints on its symbolic type T being collected whenever the reference l^T to o is used in type-related operations, such as type casting, `instanceof` and method invocations.

- Second, the program is executed symbolically by keeping track of a path condition, which comprises constraints on not only symbolic values but also symbolic types at run time.
- Third, the path condition is mutated and passed to a constraint solver to generate a new concrete test input and a harness program to test $M = c\ m(\overrightarrow{t\ x})\ \{\ldots\}$, so that a different program path can be explored. For every (symbolic) formal parameter x, as discussed above, the new test case provides a concrete value for x if x is of a primitive type and a concrete type for the object referenced by x if x is of a reference type.

This paper focuses on neither test case synthesis nor constraint solving. Existing symbolic execution techniques on unit testing [31, 54] can synthesize interactions among different methods in the same class. They can achieve high coverage by exercising different method invocation sequences on the same object to reach an object state that exposes different behaviors of the class. Our work is orthogonal. By enriching the expressiveness of path conditions, we can accelerate test coverage and improve test effectiveness.

The contributions of this work are summarized as follows:

- We propose *OO-SE* (Object-Oriented Symbolic Execution), by addressing polymorphism effectively in terms of symbolic types and providing a formalization of *OO-SE*. To the best of our knowledge, *OO-SE* is the first that executes method invocations symbolically, by mutating their targets based on the constraints on the symbolic types of their receiver objects.
- We have developed a test case generation framework for object-oriented libraries based on *OO-SE*. To test an API method in a library, this framework generates automatically different test cases and their associated harness programs for the API.
- We have implemented our test case generation framework in Java. Our experimental results on `OpenJDK` show that symbolic types are not only effective in improving test coverage but also essential in finding bugs and security vulnerabilities.

The rest of the paper is organized as follows. Section 2 motivates our approach by example. Section 3 formalizes our extension of classic symbolic execution with symbolic types to support polymorphism. Section 4 discusses our implementation details. Section 5 evaluates our approach using the `OpenJDK` library. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2. Motivating Example

Figure 2 shows a code snippet extracted from class `java.net.InterfaceAddress` of `OpenJDK 1.6.0_03-b05`. This example shows that existing symbolic execution techniques [16], which do not mutate method invocation targets, are insufficient to produce expressive path conditions for test case generation and may thus miss bugs that can be found systematically with symbolic types.

For class `InterfaceAddress`, its instance field `address` can never be null but its instance field `broadcast` may be `null`. In line 7, due to the badly written checks, `broadcast.equals()` may throw a null pointer exception, as highlighted. This error is reported as bug 6628576 in `http://bugs.java.com`. The bug can be exposed only if both `obj instanceof InterfaceAddress` at line 1 and `address.equals(cmp.address)` at line 4 hold.

Suppose we would like to test the API `InterfaceAddress.equals()`. To trigger the error, it is essential to change the type of the object referenced by the field `address` of its receiver object and the type of the object referenced by its formal parameter `obj`.

Figure 3 shows the symbolic execution tree with different execution paths for `InterfaceAddress.equals()`. At line 1, if `obj instanceof InterfaceAddress` fails, the execution returns at line 2. Otherwise, the execution continues by taking a different path from line 3. When invoking `address.equals(cmp.address)` at line 4, we may execute one of the three target methods, `InetAddress.equals()`, `Inet4Address.equals()` and `Inet6Address.equals()`, denoted m1, m2, and m3, respectively. Note that m1, i.e., `InetAddress.equals()` always returns `false` (line 10). Thus, line 7, where the null pointer error occurs, can be reached only if either m2 or m3 is invoked, requiring the object referenced by field `address` to be of type `Inet4Address` or `Inet6Address`.

Traditional symbolic execution techniques [16, 22, 54] may miss this error since they always execute a method invocation concretely. To overcome this limitation, we propose to handle polymorphism systematically with symbolic types. Figure 4 depicts how our object-oriented symbolic execution approach is applied to generate test cases automatically to trigger the null pointer error. As `InterfaceAddress.equals()` is the API being tested, there are three input objects: (1) its receiver object (O_0), (2) the object referenced by its input parameter `obj` (O_1), and (3) the object referenced by the formal parameter `addr` of `InterfaceAddress`'s constructor (O_2). As the field `address` of O_0 points to O_2, it suffices to focus on O_1 and O_2 for the purposes of understanding how the null pointer error is exposed. Let the symbolic types of O_1 and O_2 be T^1 and T^2, respectively. Thus, O_1 and O_2 may be constructed with different concrete types, c_1 and c_2, during the test case generation. In our notation, these facts are represented as $l^{T_1} \mapsto O_1^{c_1}$ and $l^{T_2} \mapsto O_2^{c_2}$.

As shown in Figure 4(a), initially the two objects O_1 and O_2 are constructed with their concrete types being their declared types: O_1 is constructed with type `Object` and O_2 is constructed with type `InetAddress`. The API method being tested is then executed sym-

```
   public class InterfaceAddress{
     private InetAddress address;
     private InetAddress broadcast;
     ...

     InterfaceAddress(InetAddress addr){
       address = addr;
     }

     public boolean equals(Object obj){
1      if(!(obj instanceof InterfaceAddress))
2         return false;
3      InterfaceAddress cmp=
               (InterfaceAddress) obj;
4      if(!address.equals(cmp.address))
5        return false;
6      if((broadcast!=null &&
           cmp.broadcast==null)||
7        (!broadcast.equals(cmp.broadcast)))
8        return false;
        ...
     }
   }

   public class InetAddress {
9   public boolean equals(Object obj){
10     return false;
     }
   }

   public class Inet4Address
          extends InetAddress{
11   public boolean equals(Object obj){
12     ...
13     return true;
     }
   }

   public class Inet6Address
          extends InetAddress{
14   public boolean equals(Object obj){
15     ...
16     return true;
     }
   }
```

Figure 2. Code taken from `java.net.InterfaceAddress` from OpenJDK 6 with some simplifications.

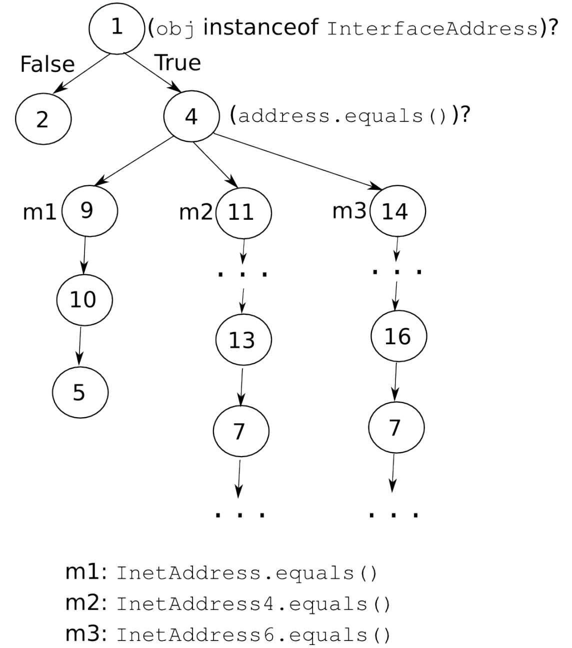

Figure 3. Symbolic execution tree for the example in Figure 2.

bolically by following the execution path from line 1 to line 2, resulting in the path condition being $\neg(T^1 \trianglelefteq$ `InterfaceAddress`).

Next, we negate the path condition and solve the thus mutated path condition $T^1 \trianglelefteq$ `InterfaceAddress` to generate a new test case. Figure 4(b) shows a solution: O_1 is constructed now with a different type, `InterfaceAddress`. As a result, the test in line 1 fails this time, generating a constraint $T^1 \trianglelefteq$ `InterfaceAddress`. After that, we execute symbolically `address.equals()` at line 4, thereby obtaining another constraint, $T^2 \overset{\text{equals}}{=}$ `InetAddress`, which asserts that the target called is `InetAddress.equals()`.

In general, mutating a constraint related to the type of the receiver object at a method invocation will generate a set of mutations, each representing a new target method yet to be invoked. This allows us to try all its feasible targets systematically. In our example, there are three target methods invokable at `address.equals()` (line 4). Thus, mutating $T^2 \overset{\text{equals}}{=}$ `InetAddress` yields two new constraints, $T^2 \overset{\text{equals}}{=}$ `Inet4Address` and $T^2 \overset{\text{equals}}{=}$ `Inet6Address`, with their solutions given in Figures 4(c) and 4(d), respectively. In either case, the null pointer error at line 7 can be exposed.

Note once again that existing symbolic execution techniques [16, 22, 54] do not mutate method invocation targets. In [16], type variables are introduced to support subtyping. While collecting type-related constraints for `instanceof` operations, this earlier work still executes a method invocation concretely. In our motivating example, it can expose the null pointer error at line 7 only if all possible types for O_1 and O_2 are enumerated exhaustively. This is impractical for large object-oriented libraries such as `OpenJDK`, as will be discussed in Section 5.

3. Object-Oriented Symbolic Execution with Symbolic Types

This section presents our approach on object-oriented symbolic execution and test case generation for object-oriented libraries. As mentioned earlier, our approach is applicable to applications as well when its parts are separately tested. The semantics of the dynamic symbolic execution is described formally using a small object-oriented language abstracted from Java.

3.1 Program Representation

The syntax of our core language is given in Figure 5, where the top half provides the abstract syntax for the source language and the bottom half provides the additional syntax used by the dynamic symbolic execution. The core language closely models conventional object-oriented languages, including dynamically allocated mutable objects, class inheritance and dynamic dispatch. Static fields and static methods are omitted since they are simpler cases of instance fields and methods. For simplicity, we do not model exception handling and concurrency features in order to focus on symbolic types and values while avoiding semantic complications. We also assume that any program in the core language is type-safe.

In the source language, a class C is defined by extending another class. A special class Object is predefined to serve as the root of the class hierarchy, for simplicity, with no field or method. The

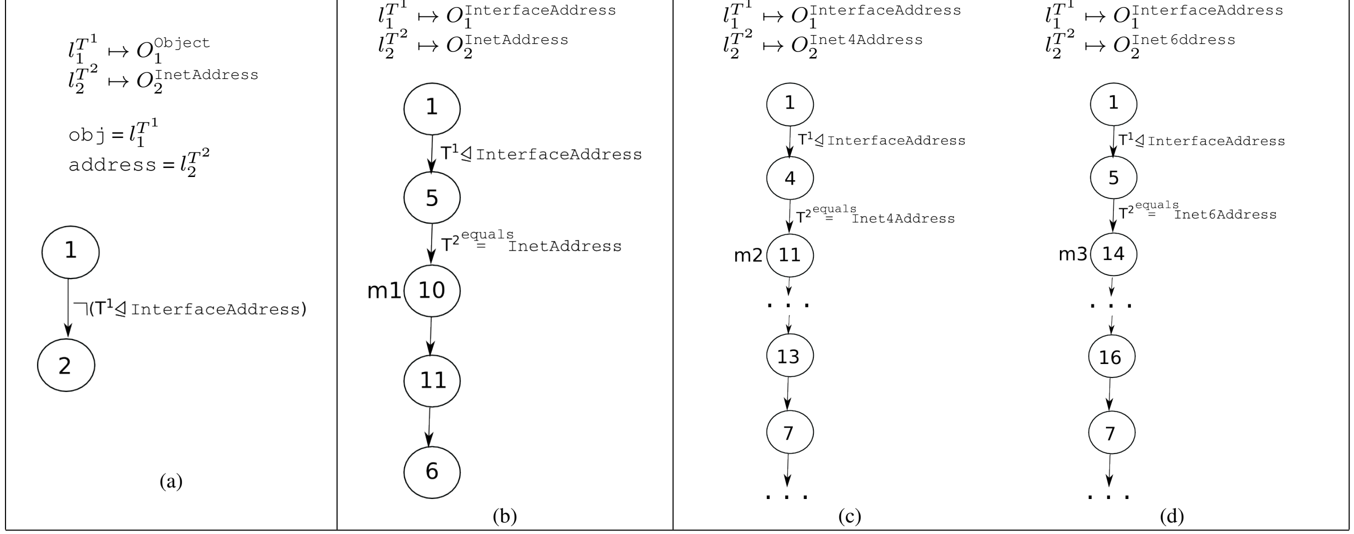

Figure 4. Dynamic object-oriented symbolic execution for the example in Figure 2.

Programs	P	::=	$\overline{C}\ e$
Classes	C	::=	class $c \lhd c\ \{\overline{t\ f};\ \overline{D}\}$
Methods	D	::=	$t\ m(\overline{t\ x})\ \{e\}$
Expressions	e	::=	$x \mid v \mid$ new $c^{T} \mid e.f \mid e.f = e \mid e.m(\overline{e})$ $\mid$ let $x = e$ in $e \mid$ if $e\ e\ e \mid e\ op\ e \mid e$ instanceof $c \mid$ halt
Types	t	::=	$c \mid$ bool $\mid$ int $\mid$ string
Values	v	::=	r^{X}
Literals	r	::=	null $\mid b \mid n \mid s$
Booleans	b	::=	true $\mid$ false
Integers	n	::=	$-1 \mid 0 \mid 1 \mid 2 \mid \ldots$
Strings	s	::=	"a" $\mid$ "bc" $\mid$ "abc" $\mid \ldots$
Operators	op	::=	$+ \mid - \mid == \mid != \mid \ldots$
Identifiers	c, f, m, x, X, T		

Heaps	H	::=	$\overline{l \mapsto o^{c}}$
Objects	o	::=	$\overline{f \mapsto v}$
Values	v	::=	$\ldots \mid r^{\varsigma} \mid l^{T}$
Symbolic representations	ς	::=	$X \mid r \mid T \trianglelefteq c \mid \varsigma\ op\ \varsigma$
Constraint lists	Σ	::=	$\emptyset \mid \Sigma, \varsigma \mid \Sigma, \neg\varsigma \mid \Sigma, T \overset{m}{=} c$
Evaluation contexts	E	::=	$[\![\cdot]\!] \mid E.f \mid E.f = e \mid v.f = E \mid E.m(\overline{e}) \mid v.m(E) \mid \overline{v}, E, \overline{e}$ $\mid$ let $x = E$ in $e \mid$ if $E\ e\ e \mid E\ op\ e \mid v\ op\ E \mid E$ instanceof c
Locations	l		

Figure 5. Abstract syntax for source language (top) and symbolic execution (bottom).

symbolic values and types, identified by X and T, respectively, are marked in gray to indicate that they are optional—introduced only in the synthesized harness program for invoking a public API of the library under test. All input values to the API are symbolic: object construction is labeled with a symbolic type and a non-object value v is a literal labeled with a symbolic value. In addition, a variable of a reference type in the harness program refers to symbolic objects or null. Thus, we can execute the API symbolically with these symbolic inputs introduced in the harness program.

In the additional syntax used in the symbolic execution, the heap H maps memory locations to objects with a given type. An object in memory maps fields to values, where the syntax of value v has been extended to include also a runtime memory location possibly labeled with a symbolic type to represent symbolically the type of the object at the location. During dynamic symbolic execution, both concrete and symbolic executions are performed together. Thus, we must keep track of both the concrete and symbolic information for a value v. Specifically, for a non-object value r^{ς}, r is its concrete value and ς is its symbolic value. For an object at location l^{T}, c is its concrete type, where $H(l) = o^{c}$, and T is its symbolic type. A symbolic representation ς can be either a symbolic identifier (X), a literal (r), a subtype expression ($T \trianglelefteq c$), or a binary operation of two symbolic expressions ($\varsigma\ op\ \varsigma$). The symbolic execution accumulates all constraints collected during execution and places them in a list Σ, which may contain a symbolic representation ς or its negation $\neg\varsigma$, or a *method identity constraint* $T \overset{m}{=} c$, which will be explained below.

[CONTEXT]
$$\frac{H_1\ \Sigma_1\ e_1 \longrightarrow H_2\ \Sigma_2\ e_2}{H_1\ \Sigma_1\ E[\![e_1]\!] \longrightarrow H_2\ \Sigma_2\ E[\![e_2]\!]}$$

[NEW]
$$\frac{\mathit{fields}(c) = \overline{f} \qquad l \notin dom(H)}{H\ \Sigma\ \mathsf{new}\ c^{\ T} \longrightarrow H, l \mapsto (\overline{f \mapsto default})^c\ \ \Sigma\ \ l^{\ T}}$$

[UPDATE]
$$H\ \Sigma\ l^{\ T}.f = v \longrightarrow H[l \mapsto H(l)[f \mapsto v]]\ \ \Sigma\ \ v$$

[LOOKUP]
$$H\ \Sigma\ l^{\ T}.f \longrightarrow H\ \Sigma\ H(l)(f)$$

[LET]
$$H\ \Sigma\ \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow H\ \Sigma\ e[v/x]$$

[OP]
$$\frac{r_1\ op\ r_2 = r}{H\ \Sigma\ r_1^{\ \varsigma_1}\ \ op\ r_2^{\ \varsigma_2}\ \longrightarrow H\ \Sigma\ \ r^{\ \varsigma_1\ op\ \varsigma_2}}$$

[INSTANCEOF-TRUE]
$$\frac{H(l) = o^{c_1} \qquad c_1 \trianglelefteq c}{H\ \Sigma\ l^{\ T}\ \mathsf{instanceof}\ c \longrightarrow H\ \Sigma\ \mathsf{true}^{\ T \trianglelefteq c}}$$

[INSTANCEOF-FALSE]
$$\frac{H(l) = o^{c_1} \qquad c_1 \ntrianglelefteq c}{H\ \Sigma\ l^{\ T}\ \mathsf{instanceof}\ c \longrightarrow H\ \Sigma\ \mathsf{false}^{\ T \trianglelefteq c}}$$

[INSTANCEOF-NULL]
$$H\ \Sigma\ \mathsf{null}^{\ \varsigma}\ \mathsf{instanceof}\ c \longrightarrow H\ \Sigma\ \mathsf{false}^{\ \mathsf{false}}$$

[TRUE]
$$H\ \Sigma\ \mathsf{if\ true}^{\ \varsigma}\ e_1\ e_2 \longrightarrow H\ \ \Sigma\,,\varsigma\ \ e_1$$

[FALSE]
$$H\ \Sigma\ \mathsf{if\ false}^{\ \varsigma}\ e_1\ e_2 \longrightarrow H\ \ \Sigma\,,\neg\varsigma\ \ e_2$$

[CALL]
$$\frac{H(l) = o^{c_1} \qquad method(c_1, c.m) = t\ m(\overline{t\ x})\ \{e\}}{H\ \Sigma\ \ l^{\ T}.m(\overline{v}) \longrightarrow H\ \Sigma\,,T \overset{m}{=} c\ \ e[\overline{v/x}]}$$

Figure 6. Operational semantics of dynamic symbolic execution with symbolic types.

[FIELDS-DECLARED]
$$\frac{\mathsf{class}\ c \vartriangleleft c_1\ \{t_1\ f_1, ..., t_n\ f_n;\ ...\}}{\mathit{fields}(c) = f_1, .., f_n, \mathit{fields}(c_1)}$$

[FIELDS-OBJECT]
$$\mathit{fields}(\mathsf{Object}) = \emptyset$$

[METHOD-DECLARED]
$$\frac{\mathsf{class}\ c\ ...\ \{...\ t\ m(\overline{t\ x})\ \{e\}\ ...\}}{method(c, c.m) = t\ m(\overline{t\ x})\ \{e\}}$$

[METHOD-INHERITED]
$$\frac{\mathsf{class}\ c \vartriangleleft c_1\ \{...;\ \overline{D}\} \qquad t\ m(\overline{t\ x})\ \{e\} \notin \overline{D}}{method(c, c_2.m) = method(c_1, c_2.m)}$$

[METHOD-IDENTITY]
$$\frac{method(c_1, c_2.m) = t\ m(\overline{t\ x})\ \{e\}}{c_1 \overset{m}{=} c_2}$$

[SUBTYPE-REFLEXIVE]
$$c \trianglelefteq c$$

[SUBTYPE-OBJECT]
$$c \trianglelefteq \mathsf{Object}$$

[SUBTYPE-EXTEND]
$$\frac{\mathsf{class}\ c_1 \vartriangleleft c_2\ \{...\}}{c_1 \trianglelefteq c_2}$$

[SUBTYPE-TRANSITIVE]
$$\frac{c_1 \trianglelefteq c \qquad c \trianglelefteq c_2}{c_1 \trianglelefteq c_2}$$

Figure 7. Auxiliary definitions.

3.2 Operational Semantics

Figure 6 presents small-step operational semantics for dynamic symbolic execution with symbolic types. These rules apply to both the object-oriented library being tested and any harness program synthesized. Note that the grayed parts are optional and used only when there are symbolic representations involved. We write $e[v/x]$ to stand for a substitution of x with v in e. In addition, we write $H(l)$ to look up a mapping and $o[f \mapsto v]$ to update a mapping.

For completeness, we give the standard auxiliary definitions in Figure 7. The [FIELDS-*] and [METHOD-*] rules are introduced for field and method lookups. Methods are qualified with their defining classes; for example, $c.m$ denotes method m defined in class c. [METHOD-IDENTITY] is used in constraint solving to discover an identical method definition. The subtyping constraint is reflexive and transitive, defined by the [SUBTYPE-*] rules.

Each expression in Figure 6 is evaluated in the form of:

$$H_1\ \Sigma_1\ e_1 \longrightarrow H_2\ \Sigma_2\ e_2$$

where heap H may be updated by new object construction ([NEW]) and field update ([UPDATE]) and Σ (representing the path condition of the program) is an ordered list of constraints generated by evaluating conditionals ([TRUE] and [FALSE]) and virtual calls ([CALL]). Rule [NEW] creates a new object and initializes all its fields to default values. Rule [UPDATE] updates the value of an object field. For simplicity, we regard memory locations as constants and do not model heap updates ([UPDATE]) and heap lookups ([LOOKUP]) symbolically. We use the conventional form of evaluation contexts E to reduce the number of evaluation rules. Therefore, rule [CONTEXT] provides a context for each small-step reduction.

The initial state for evaluating a (harness) program is $\emptyset\ \emptyset\ e$, where e is the body of the main method of the program, and both the heap and the constraint list are initially empty. The evaluation of the program (in multiple steps) yields a complete list of constraints, Σ, that can be mutated to generate new test inputs:

$$\emptyset\ \emptyset\ e \longrightarrow^{*} H\ \Sigma\ v$$

3.2.1 Symbolic Expressions

New symbolic expressions of values are introduced in [OP] and the three [INSTANCEOF-*] rules. The [OP] rule manipulates existing symbolic representations, where $r_1\ op\ r_2$ represents an evaluation of an arithmetic or relational operation on literals. Given $1+2=3$, for example, we will have $H\ \Sigma\ 1^{\varsigma_1} + 2^{\varsigma_2} \longrightarrow H\ \Sigma\ 3^{\varsigma_1+\varsigma_2}$. Note that the unary negation operation $!e$ can be encoded as $e == \mathsf{false}$.

The three [INSTANCEOF-*] rules handle an `is-instance-of` test of the form `v instanceof c`. If `v` is a heap object represented by l^T, the test results in the subtype expression $T \trianglelefteq c$ ([INSTANCEOF-TRUE] and [INSTANCEOF-FALSE]). If v is null, then the test is always false ([INSTANCEOF-NULL]). In Java, every class is a subtype of `Object`. Hence, $T \trianglelefteq$ `Object` always holds. In our motivating example given Figure 2, we discussed earlier on testing the public API method `InterfaceAddress.equals()`. The object referenced by its formal parameter `obj` is represented as `obj` $= l^T$ with a symbolic type T. Initially, the object is constructed with the declared type `Object`, giving rise to $l^T \mapsto O^{\texttt{Object}}$. At line 1, the test `obj instanceof InterfaceAddress` is then translated to: $H\ \Sigma\ l^T\ \mathsf{instanceof}\,\texttt{InterfaceAddress} \longrightarrow H\ \Sigma\ \mathsf{false}^{T \trianglelefteq \texttt{InterfaceAddress}}$.

We also consider casting to be also handled by the [INSTANCEOF-*] rules, together with [LET]. For example, a Java statement $\{x = (c)\ y, \ldots\}$, where $y = l^T$, is modeled as $\{\mathsf{let}\ x = \mathsf{if}\ l^t == \mathsf{null}\ \mathsf{null}\ \mathsf{if}\ l^T\ \mathsf{instanceof}\ c\ l^T\ \mathsf{halt}, \cdots\}$. As a result, the subtype expression $T \trianglelefteq c$ generated by casting always needs to be satisfied. Otherwise, the program halts. Our definition is relatively abstract and generally simple through the introduction of an subtype expression. A more elaborate alternative for Java is described in the entry for `instanceof` in Chapter 6 of the Java Virtual Machine (JVM) Specification [36]. Specific casting rules for Java also depend on the JVM specification, in the entry for `checkcast` in [36].

3.2.2 Path Constraints

When a program path is followed, constraints are collected in evaluating conditionals ([TRUE] and [FALSE]) and virtual calls ([CALL]). A constraint represents a condition that needs to be satisfied for the same path to be taken during program execution. Thus, [TRUE] ([FALSE]) introduces a constraint for a conditional when its true (false) branch is taken. Solutions that satisfy the same conditional constraint will always cause the same branch to be executed.

Similarly, we introduce constraints on the types of receiver objects at method invocations ([CALL]). For a virtual call $l^T.m(\overline{v})$, where $H(l) = o^{c_1}$, the method identity constraint $T \stackrel{m}{=} c$ asserts that the invoked target must be method $c.m$, if $c = c_1$, where c_1 declares m ($method(c_1, c_1.m)$) or c_1 inherits (transitively) m from c ($method(c_1, c.m)$). Solutions that satisfy the same method identity constraint will always cause the same target to be invoked, although the type c_1 of the receiver object o^{c_1} may differ each time.

3.3 Test Generation and Constraint Solving

A program path is represented as a path condition Σ, which is an ordered list of constraints $\varsigma_0, \varsigma_1, \ldots, \varsigma_n$. Each constraint represents a condition for a particular branch to be taken along the path. During the test case generation, Σ is mutated and then solved by a constraint solver to generate new test cases to execute new paths.

As in prior work [22], we mutate one constraint in Σ at a time. Thus, a mutated Σ consists of one mutated constraint, together with the remaining constraints carried over unchanged from Σ. For example, if we mutate the constraints in $\Sigma = \varsigma_0, \varsigma_1, \ldots, \varsigma_n$ successively, we will obtain new path conditions: $\widetilde{\Sigma}_0 = \widetilde{\varsigma_0}$, $\widetilde{\Sigma}_1 = \varsigma_0, \widetilde{\varsigma_1}$, …, $\widetilde{\Sigma}_n = \varsigma_0, \varsigma_1, \ldots, \widetilde{\varsigma_n}$. Each mutation represents a new path, which diverges from the original path Σ at the mutated branch.

A constraint ς_0 introduced by [TRUE] or [FALSE] represents a conditional branch. Its mutation is simply the negation, i.e., $\widetilde{\varsigma_0} = \neg\varsigma_0$, which represents the other branch of the same conditional. A method identity constraint $T \stackrel{m}{=} c_0$ represents one target method for a virtual call. There may be multiple mutations, each representing a new target to be explored. Let $\{c_0.m, c_1.m, \ldots, c_n.m\}$ be the set of all such targets. The mutation $\widetilde{T \stackrel{m}{=} c_0}$ is the set of constraints $T \stackrel{m}{=} c_1, \ldots, T \stackrel{m}{=} c_n$. Hence, we have generated a set of new path conditions, allowing us to explore all possible targets for this particular virtual call. The set of all possible targets at a virtual call is estimated conservatively by static analysis. While an over-approximation affects neither the soundness nor the completeness of our approach, a precise estimation can help filter out some infeasible targets, accelerating constraint solving and test generation.

A mutated path condition is solved by a constraint solver to generate a new test input. All the constraints on symbolic values are handled in the standard manner [22]. There are two kinds of constraints on symbolic types. A subtyping constraint, $T \trianglelefteq c$ or $\neg(T \trianglelefteq c)$, is solved by applying the [SUBTYPE-*] rules. A method identity constraint $T \stackrel{m}{=} c$ is solved by [METHOD-IDENTITY].

By solving the method identity constraint $T \stackrel{m}{=} c$ at a virtual call and its mutations, we have effectively generated test cases to explore all its feasible invocation targets. Each test case will exercise a different target, with all the infeasible targets being filtered out by the constraint solver. In our current implementation, we use the classic class hierarchy analysis (CHA) [15] to find conservatively the potential targets for a virtual call. There is no noticeable performance difference if a more precise analysis, say, Andersen-style points-to analysis [3, 29, 30, 39, 50, 53], is used, because infeasible invocation targets are usually rejected quickly by the constraint solver at little cost.

3.4 Soundness and Completeness

The soundness and completeness of a symbolic executor are usually defined with respect to a bug finder [16, 21]. For soundness, if a bug is found by the symbolic executor, there must be a corresponding concrete execution path leading to this bug. For completeness, if there is a bug in the concrete execution, then it can be found by the symbolic executor given sufficient time. In other words, soundness implies that no infeasible paths are executed while completeness implies that all possible feasible paths can be explored.

Therefore, the proof for establishing the soundness of our approach in the core language is immediate. By performing a dynamic symbolic execution, all program paths executed are feasible.

The proof of completeness proceeds by induction on $\longrightarrow^*$ to show that each concrete execution path has a corresponding path condition solved by the constraint solver used. The completeness assumes that its underlying decision procedure of pruning infeasible paths does not exclude any feasible path. Our approach is parameterized on the underlying decision procedure used, as it is orthogonal to reasoning about virtual invocations with symbolic types. More importantly, our use of symbolic types to explore targets of virtual invocations does not introduce unnecessary incompleteness, as our approach rests on the assumption that any potential call target in a virtual call is (conservatively) resolved by static analysis, e.g., the class hierarchy analysis (CHA) [15].

4. Implementation

We have implemented our object-oriented symbolic execution approach in a DSE tool, as depicted in Figure 1. The objective for developing this tool is to find effectively security vulnerabilities in the `OpenJDK` library. We enable symbolic execution by instrumentation: `OpenJDK` is compiled using the SOOT [58] compilation framework and every SOOT instruction is instrumented with a cor-

responding function call, to collect constraints and update symbolic states accordingly. During the instrumented execution of an API method, invoked by a synthesized harness program, the constraints are generated and added to the path condition for the path being explored.

Our tool enumerates all the public API methods in OpenJDK, and for each API method, synthesizes automatically Java bytecode programs that execute different paths of the API method. The bytecode synthesizer is built on top of the ASM library [5].

As shown in Figure 1, each API method is tested iteratively, with a new input at each iteration in order to exercise a new path in the API method. To obtain an input for the next iteration, the path condition for the current input is mutated and solved for a solution by using a constraint solver. This solution is the next input, representing a new path to be explored. For each input, its synthesized bytecode program contains a `main` method, i.e., a harness that, when executed, will instantiates objects to be used as the arguments for the API method being tested. These objects are created by invoking the public constructors in the classes specified by the input.

For an argument of a reference type, the synthesizer can choose to use a new object of an appropriate type (as specified by the test input), null, or an existing object of an equivalent type. This simple approach accounts for all possible aliases among the input objects in [16, 60] but often suffer from the scalability problem. In our experiments on OpenJDK, no improvement on test coverage is observed. Therefore, aliases are not considered, by default.

Our constraint solver is implemented in SWIProlog [52]. It supports both type and linear arithmetic constraints as well as equality constraints on strings. For linear arithmetic, the solver resorts to the CLP(R) library of SWI Prolog, which is implemented based on the simplex algorithm, coupled with our own implementation of a backtracking algorithm for finding integral solutions.

In our implementation, we have also generalized symbolic types to handle reflection. In Java, reflection is realized by representing classes, methods and fields as objects of appropriate types, known as metaobjects: `java.lang.Class` for classes, `java.lang.reflect.Method` for methods and `java.lang.reflect.Field` for fields. Java also provides a reflection API for creating metaobjects by their (string) names and for making field accesses and method invocations reflectively. Table 1 gives a few API examples.

Table 1. Representative Methods in the Java Reflection API

API	Semantics
c = `Class.forName(`s`)`	returns a `Class` metaobject for the class named by the string s
$m = c$`.getMethod(`s`)`	returns a `Method` metaobject m for the method named by s in class c
m`.invoke(o,...)`	invokes method m reflectively on the receiver object o with the arguments "..."

Leveraging the notion of symbolic types, we represent `Method` and `Field` metaobjects symbolically with *symbolic methods* and *symbolic fields*, respectively. Given a reflective call m`.invoke()`, a symbolic method represents all possible target methods that are referenced by the `Method` metaobject m. Proceeding similarly as in the case of a regular virtual call ([CALL]), we introduce an identity constraint between a symbolic method and a reflective target. By mutating this identify constraint, we can explore different reflective targets at the reflective call m`.invoke()`. How do we find its possible targets statically and feed them to our symbolic executor? Presently, how to analyze Java programs in the presence of reflection soundly, precisely and scalably is a big challenge in its own right [33–35, 37]. As we focus on finding security vulnerabilities in OpenJDK, the set of targets at a reflective call is selected manually to include those that, if called, may pose potential security threats. Due to the type rules enforced during symbolic execution, infeasible targets will never be executed. Just like a symbolic method, a symbolic field represents all possible fields referenced by a `Field` metaobject and is dealt with similarly. Our handling of reflection enables the exploration of different reflective targets at a reflective call systematically during test case generation.

We do not symbolically execute native methods and do not generate constraints for exceptions. As a result, values returned from and objects created in native methods are concretized, with no associated symbolic representations. A native method may modify the state of a Java object but this is not tracked. As a result, the symbolic representation of an object can sometimes be inconsistent with its runtime state, leading to the classic path-divergence problem [45], where the tested program runs an unpredicted path. For dynamic symbolic execution, path divergence affects completeness but not soundness. In other words, path divergence causes some paths to be missed but never unreachable code to be executed.

As with many other dynamic symbolic execution tools [22, 55], we can detect path divergence by comparing the predicted path of a test case with its real execution path. The execution path in the program can be traced precisely, even in the presence of native code and exceptions. In our implementation, we maintain a symbolic stack. When a native method is invoked, a pseudo stack frame (marked as native) is pushed onto the stack, which is popped out after the native method has completed its execution. When a Java method is executed, the symbolic stack is examined to check whether it is invoked from a native frame or from another Java method. Similarly, when an exception is thrown, we examine the symbolic stack to check from which method (native or Java) it is thrown, and by which method (native or Java) it is caught. The symbolic stack is then unwound accordingly.

5. Evaluation

The objective of our evaluation is to demonstrate that our approach (denoted *OO-SE*) is superior over the state-of-the-art (denoted CLASSIC-DSE) for testing object-oriented libraries. CLASSIC-DSE stands for a classic dynamic symbolic execution technique [16, 22, 54] that executes all virtual method invocations concretely. By exploring method invocation targets with symbolic types systematically, *OO-SE* can outperform CLASSIC-DSE in terms of both test coverage achieved and bug-finding ability demonstrated.

Note that the work of [16] models subtyping constraints from `instanceof` operations but stills executes virtual calls concretely. To test a public API method, this earlier work instantiates an input object for a formal parameter of a reference type with any subtype that satisfies its type constraints collected. This is impractical for large object-oriented libraries like OpenJDK 7, which has a total of 23,309 subclasses of `java.lang.Object`. For all the packages evaluated, except `j.applet`, this brute-force approach cannot run to completion within 24 hours: it often gets stuck at trying input objects of all possible subtypes for one particular formal parameter of type `java.lang.Object` without making progress, resulting in poor coverage. Thus, it will not be discussed any further.

Table 2. Tested Packages from OpenJDK 7 Update 6

Package	#LOC	#Classes	#Public Methods	#Basic Blocks	#Virtual Calls
j.applet	184	2	28	53	26
j.sql	2594	31	220	533	332
j.rmi	2452	60	195	762	390
j.beans	7425	139	376	3500	2254
j.text	9881	67	482	4512	2310

Table 3. Times on analyzing the classes and methods tested

Package	#Tested / #Total		#Test Cases Generated		Run Time (m:s)	
	Classes	Public Methods	Classic-DSE	OO-SE	Classic-DSE	OO-SE
j.applet	1 / 2	17 / 28	17	28	1m56s	2m47s
j.sql	13 / 16	65 / 69	92	110	5m12s	7m22s
j.rmi	18 / 35	49 / 62	64	102	21m18s	41m9s
j.beans	41 / 128	166 / 342	301	671	126m8s	665m9s
j.text	32 / 63	236 / 376	409	783	225m11s	726m10s

Therefore, we will compare CLASSIC-DSE and *OO-SE* by using the five packages from OpenJDK 7 Update 6, as shown in Table 2, where the packages are listed in the ascending order of the number of basic blocks possessed. The five packages are selected because in testing the five packages, we did not observe any path divergence and both CLASSIC-DSE and *OO-SE* are able to mutate all path conditions until termination within 24 hours. The other packages in OpenJDK either take too long to make any progress in symbolic execution or encounter frequently path divergences due to invocations to native methods. As a result, we do not report here the results on these packages, because their coverage data vary across different runs, and this non-determinism may lead to invalid conclusions.

For each package evaluated, as listed in Table 2, we give the number of uncommented lines of code as reported by the SLOCCount tool [61]. Blank, comment or whitespace-only lines are not considered. In addition, we also give the number of basic blocks and the number of virtual calls in a package. In `java.beans`, there is one reflective call to `java.lang.reflect.Method.invoke()`, which is analyzed symbolically by *OO-SE* (as discussed in Section 4) but only concretely by CLASSIC-DSE.

We instrument the OpenJDK library to perform symbolic execution functionalities. The instrumented OpenJDK library runs together with each synthesized harness program to generate path conditions during program execution. In our experiments, we have instrumented all the `java.*` packages except `java.util`. Our instrumentation of `java.util` tends to cause instability for the runtime of each tool, as the runtime itself depends heavily on the data structures implemented in `java.util`. It is sufficient to instrument the `java.*` only, since in OpenJDK, the packages that we experimented with depend mainly on the `java` package itself. For the actual instrumentation, we also exclude the classes `java.lang.Object`, `java.io.File` and `java.lang.invoke.FromGeneric.Adapter` due to stability issues.

We have conducted our experiments on an 8-core Intel Xeon 3.00 GHz system with 64GB memory running Linux 3.13.0.

In our evaluation, we address the following research questions:

1. RQ1: Is *OO-SE* still scalable relative to CLASSIC-DSE?
2. RQ2: Does *OO-SE* generate more *mutated* test cases than CLASSIC-DSE, resulting in improved test coverage?
3. RQ3: Is *OO-SE* more effective than CLASSIC-DSE in finding bugs and security vulnerabilities?

5.1 Efficiency

Table 3 gives the run times of CLASSIC-DSE and *OO-SE* elapsed in testing all the five packages. The run time spent by each tool in testing a package includes the times taken for synthesizing the harness programs for all the test cases, generating these test cases by the constraint solver, and executing these test cases symbolically. Each package is analyzed until the end of automatic test generation when no more new test cases can be generated.

Presently, our harness synthesizer can only instantiate objects by using public constructors. In OpenJDK, some classes do not have public constructors as they rely on factory methods exclusively for object instantiations. As a result, our synthesizer cannot successfully generate Java bytecode programs to test those methods whose receiver objects or input arguments cannot be instantiated. In Column 2 (Column 3), we give the number of classes (public API methods) that is actually tested, together the total number of classes (public API methods) available, in each package.

As *OO-SE* generates more test cases than CLASSIC-DSE (Columns 4 and 5), *OO-SE* is expected to run more slowly than CLASSIC-DSE, especially for the two largest packages, `java.beans` and `java.text`, evaluated (Columns 6 and 7). In general, constraint solving accounts for the majority of the run time for each package, with more than 90% for `java.text` and `java.beans`. When testing `java.text` under *OO-SE*, seven time-consuming path conditions are generated, costing the constraint solver over 5 minutes to solve each path condition for a solution.

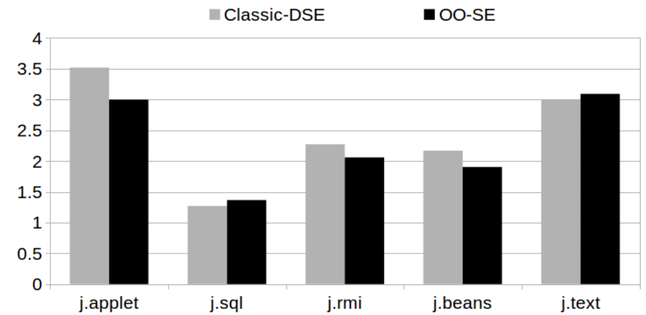

Figure 8. Run time per basic block (secs).

Figure 8 compares CLASSIC-DSE and *OO-SE* in terms of their efficiency per block of code. In this experiment, we limit the depth of the symbolic reasoning to a fixed number of 20 basic blocks, which provides a reasonable comparison between the two tools. The run time per block varies little across the packages—1–3.5 seconds per block. There are no significant differences between CLASSIC-DSE and *OO-SE*. The high run time per block of code for `java.applet` can be attributed to its relatively small size, such that a small overhead has a relatively high impact. Otherwise, there appears to be a small tendency for the run time per block to increase with the size of the package (i.e., number of blocks in the package).

5.2 Coverage

Figure 9 compares CLASSIC-DSE and *OO-SE* in terms of block coverage achieved. In computing the coverage for a package, we exclude its methods that cannot be reached by the tested public API methods in the package, since the excluded methods are not tested.

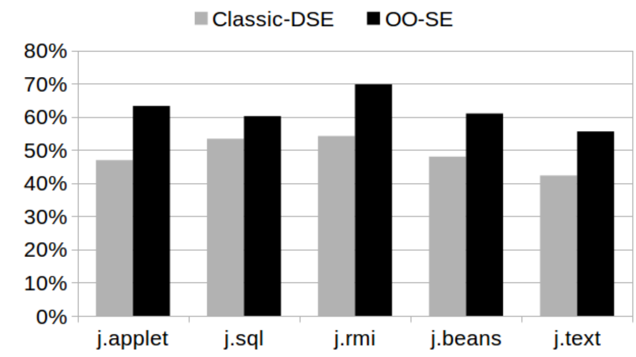

Figure 9. Block Coverage.

With symbolic types, *OO-SE* can automatically test `OpenJDK` more effectively than CLASSIC-DSE. For the five packages tested, the coverage improvements range from 12.8% at `java.sql` to 28.7% at `java.rmi` with an average of 24.3%. In particular, *OO-SE* has significantly improved CLASSIC-DSE in testing `java.rmi` by lifting its coverage from 54.2% to 69.8%. For all the five packages, except `java.text`, *OO-SE* is able to achieve a coverage of more than 60%. Some blocks are not covered due to exception handling and conditional branches that do not depend on symbolic inputs. In our current implementation, *OO-SE* does not generate constraints for exceptions. Hence, most exception handling blocks are not covered. In addition, *OO-SE* neither reasons symbolically about string operations nor represents symbolically values returning from native methods. As a result, just like CLASSIC-DSE, *OO-SE* cannot effectively explore different targets of a conditional branch that depends on some values with no symbolic representations.

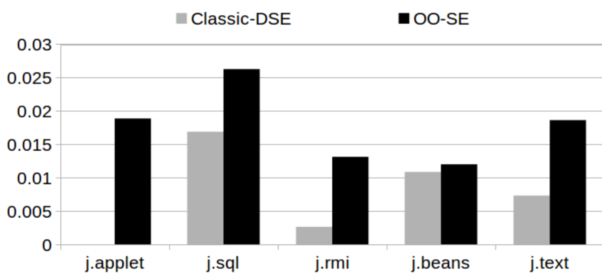

Figure 10. Number of mutated path conditions per basic block.

Figure 10 compares CLASSIC-DSE and *OO-SE* in terms of the number of path conditions, i.e., test cases mutated from another one per block of code. In this experiment, we also limit the depth of the symbolic reasoning to a fixed number of 20 blocks. Note that the first test input used for bootstrapping the symbolic execution of each public API method is not counted. For `java.sql` and `java.text`, *OO-SE* has produced a large number of mutated path conditions, i.e., new test cases. For `java.applet`, CLASSIC-DSE, without exploiting symbolic types, has failed to produce any new test case. In the case of `java.sql`, `java.rmi` and `java.text`, *OO-SE* has produced significantly more test cases than CLASSIC-DSE. With `java.applet` ignored, *OO-SE* has increased the number of mutated test cases per block produced by CLASSIC-DSE by 84.14%. However, this increase does not seem to have any correlation with the number of virtual invocations available in these packages (Table 2).

Figure 10 shows that *OO-SE* is able to generate significantly more tests than CLASSIC-DSE. In testing method `java.beans.Statement.execute()`, *OO-SE* has generated a total number of 47 tests, compared to only 2 tests generated by CLASSIC-DSE. This has greatly improved the ability of traditional symbolic execution techniques in finding bugs and security vulnerabilities, as discussed below.

5.3 Bug Checking and Vulnerability Detection

We describe a case study to show how our approach can be applied for bug checking and vulnerability detection. We examine the class `java.beans.Statement` and show we can automatically generate test cases that expose its bugs and vulnerabilities.

```
public class Statement {
  public Statement(Object target,
                   String methodName,
                   Object[] arguments) {
    ...
  }
  public void execute() throws Exception {
    invokeInternal();
  }
  private Object invokeInternal()
    throws Exception {
    if (target == Class.class &&
      methodName.equals("forName")) {
      ...
    }
    if (target instanceof Class) {
      if (methodName.equals("new"))
        ...
      }
    }
    ...
}
```

Figure 11. Code snippet from `OpenJDK` 7.

The class `java.beans.Statement` represents a primitive statement in which a single method is applied to a target and a set of arguments. Figure 11 gives a code snippet of its implementation. The constructor of this class takes three arguments: the receiver object (`target`), the method name (`methodName`), and a list of arguments (`arguments`). The public API `execute` can be invoked to execute the statement it represents. Thus, `new Statement(a,"foo",b).`
`execute()`, for example, is equivalent to `a.foo(b)`.

In `OpenJDK`, the directory `jdk/test` consists of a large set of regression tests, most of which are introduced due to a particular bug report filed previously. There are five manually created regression tests for this particular class, of which three tests, `Test6707226`, `Test4653179` and `Test6224433`, can be automatically generated by our technique. The other two check the behavior of this class for overloaded and overridden methods, which are beyond the scope of present-day test case generation techniques.

`Test6224433` is of particular interest, as it is directly related to a vulnerability in `JDK` (CVE-2013-0422), which allows users to access restricted classes. In its simplified form, this test is:

```
new Statement(Class.class, "forName",
        "sun.misc.BASE64Encoder")
        .execute();
```

In `OpenJDK`, there are security checks to prevent user access to restricted classes. As a result, the method call `Class.forName("sun.misc.BASE64Encoder")` will throw a security exception since class `sun.misc.BASE64Encoder` is restricted. However, if method `Class.forName` is invoked indirectly via the API `java.beans.`

`Statement.execute()`, such security checks may be bypassed. With all the restricted classes being annotated, we can apply our approach to generate test cases automatically to expose this vulnerability. We achieve this by testing the constructor `Statement` (`target`, `methodName`, and `arguments`) with its three formal parameters receiving appropriate symbolic representations.

For the vulnerability CVE-2013-0422 in class `java.beans.Statement.execute()`, *OO-SE* can find it in 29 minutes with 47 test cases being generated in total. In contrast, CLASSIC-DSE cannot expose this bug as only two test cases are generated.

5.4 Limitations

At this stage, we have not incorporated into our tool with performance-oriented optimizations to improve its efficiency. For example, sophisticated path selection algorithms [9, 32] can be used to achieve a desired level of test coverage more quickly. In addition, adding method identity constraints to the path condition ([CALL]) enriches the runtime information with calling contexts, bringing in new optimization opportunities. How to better make use of such calling context information to guide path selection, especially call target selection, is worth separate investigation.

Presently, our harness synthesizer instantiates an object of a given class type by invoking the public constructors in the class. We associate a symbolic representation with a field of a reference type in an instantiated object if the field is set in the constructor or can be set by a setter method. Sometimes, a field cannot be set directly, requiring its owner object to be at a specific state. To further improve test coverage, we may need to synthesize different method invocation sequences to instantiate an object with different states [54].

6. Related Work

Symbolic execution [26] is an important program analysis technique that executes programs with symbolic instead of concrete values. There have been a number of recent advances [2, 8, 10, 12, 21, 22, 32, 49, 56]. Researchers have applied symbolic execution to verification [13, 23, 42], impact analysis [43], debugging [4, 12, 47], program synthesis [6, 57], and automated testing [1, 17, 18, 20–22, 40, 46, 48, 49, 55, 60]. For some applications, symbolic execution tools can automatically generate test inputs that achieve better code coverage than manually-crafted ones [10].

Symbolic execution can be performed either dynamically (at runtime) or statically (as a form of abstract interpretation [14]). Static symbolic execution tools include KLEE [10], CLOUD9 [8], S2E [12], BitBlaze [51], and Symbolic Pathfinder [2, 25, 44, 46]. KLEE is a symbolic execution tool for C programs. It has been extended to support languages such as CUDA [28], C++ [27], and binary instructions [12]. Symbolic Pathfinder is based on Java Pathfinder [24], a model checker for Java programs developed at NASA. BitBlaze [51] is a symbolic execution tool for binaries.

Static tools can suffer from false positives due to their static approximation of program semantics. KLEE alleviates this problem by translating LLVM [38] instructions into constraints with minimal approximations (to achieve bit-level accuracy). For a library call whose semantics is statically unknown, user annotations are required. CLOUD9 [8] extends KLEE by providing more support for the POSIX library, which makes it applicable to more benchmarks than KLEE. S2E [12] adopts a different approach to addressing the unknown semantics of libraries, by providing a symbolic execution-enabled virtual machine for binary programs. Thus, there is no external library call with unknown semantics.

Dynamic symbolic execution (*DSE*)—also known as concolic (concrete-symbolic) execution [49]—performs symbolic execution at runtime together with concrete execution. This enables DSE tools to effectively check the correctness of symbolic semantics, by avoiding false positives (due to infeasible paths executed). This approach can simplify some hard-to-solve path conditions into simpler forms. It is more suitable than a static approach for testing `JDK`, given its complexity and its heavy usage on native libraries.

DART [21] is the foundation of DSE tools, pioneering the use of concrete values to simplify path constraints. SMART (Systematic Modular Automated Random Testing) [19] is an extension of DART, which uses function summaries and composition to scale up dynamic symbolic execution. CUTE and JCUTE [48, 49] are versions of DART for C and Java programs, respectively. The most significant improvement made in CUTE over DART is its ability in handling pointers, although this is limited to simple cases. CREST [9] is a re-engineering of CUTE that adds several path-exploration heuristics on top of the original depth-first search strategy. LATEST [41] is an extension of CUTE that uses abstractions of function calls from a designated function under test to simplify path conditions, allowing users to explore more relevant program parts. PEX [55] models the .NET library and performs DSE on .NET programs to check common errors. SAGE [22] is a DSE tool for binary programs used internally at Microsoft and has already found hundreds of security-related bugs in Microsoft products [7].

Constraints on types have been used in previous symbolic execution techniques. In [16, 59], type constraints are introduced to generate input heap shapes with pointer or reference inputs. PEX [55] encodes symbolically type constraints as universally quantified formulas. In this paper, we introduce symbolic types for polymorphism and mutate it to explore different invocation targets.

7. Conclusion

We have proposed object-oriented symbolic execution for software testing. For the first time, we support polymorphism by introducing constraints for method invocation targets via symbolic types. This enables a systematic exploration of method invocation targets, which is critical for testing object-oriented libraries, such as `JDK`, effectively. We have also generalized the notion of symbolic types to symbolic methods and symbolic fields to handle Java reflection symbolically. The basic idea can be applied to symbolic execution of programs in dynamic languages such as JavaScript.

We have implemented a dynamic symbolic execution tool with symbolic types for testing object-oriented libraries. Our results show that our approach is both effective in achieving test coverage and essential for finding security vulnerabilities in `OpenJDK`.

In future work, we will sharpen our tool with security checks to automatically find security vulnerabilities such as access control violations in `OpenJDK`. We also plan to extend our symbolic execution semantics to support the Java reflection 1.2 API effectively.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS '08*, pages 367–381.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS '07*, pages 134–138.

[3] L. O. Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen*, 1994.

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA '10*, pages 49–60.

[5] ASM. ASM 4.0: A Java bytecode engineering library. `http://asm.ow2.org`, 2012.

[6] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Comm. ACM*, 57(2):74–84, Feb. 2014.

[7] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE '13*, pages 122–131.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys '11*, pages 183–198.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08*, pages 443–446.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*, pages 209–224.

[11] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Comm. ACM*, 56(2):82–90, Feb. 2013.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS '11*, pages 265–278.

[13] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based bounded model checking for embedded ANSI-C software. In *ASE '09*, pages 137–148.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252.

[15] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*, pages 77–101.

[16] X. Deng, J. Lee, and Robby. Efficient and formal generalized symbolic execution. *Automated Software Engg.*, 19(3):233–301, Sept. 2012.

[17] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA '07*, pages 151–162.

[18] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09*, pages 474–484.

[19] P. Godefroid. Compositional dynamic test generation. In *POPL '07*, pages 47–54.

[20] P. Godefroid. Higher-order test generation. In *PLDI '11*, pages 258–269.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05*, pages 213–223.

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS '08*, 2008.

[23] J. Hatcliff, Robby, P. Chalin, and J. Belt. Explicating symbolic execution (xSymExe): An evidence-based verification framework. In *ICSE '13*, pages 222–231.

[24] K. Havelund. Java PathFinder, a translator from Java to Promela. In *SPIN '99*, pages 152–.

[25] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS '03*, pages 553–568.

[26] J. C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[27] G. Li, I. Ghosh, and S. P. Rajan. KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In *CAV '11*, pages 609–615.

[28] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPoPP '12*, pages 215–224.

[29] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *ESEC/FSE '11*, pages 343–353.

[30] L. Li, C. Cifuentes, and N. Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM '13*, pages 85–96.

[31] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE '09*, pages 515–519.

[32] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA '13*, pages 19–32.

[33] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP '14*, pages 27–53.

[34] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *SAS '15*, pages 162–180.

[35] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: slicing by sequential criteria. In *ECOOP '16*.

[36] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Inc., second edition, 1999.

[37] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotk, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Mller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58(2):44–46, 2015.

[38] LLVM language reference manual, 2013. Retrieved 3 September 2013.

[39] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. CC '13, pages 61–81.

[40] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07*, pages 416–426.

[41] R. Majumdar and K. Sen. LATEST: lazy dynamic test input generation. Technical report, Berkeley Digital Library SunSITE [http://sunsite2.berkeley.edu:8088/oaicat/OAIHandler] (United States), 2007.

[42] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *ICSE '10*, pages 355–364.

[43] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI '11*, pages 504–515.

[44] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE '10*, pages 179–180.

[45] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA '11*, pages 34–44.

[46] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing Nasa software. In *ISSTA '08*, pages 15–26.

[47] D. Qi, H. D. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *ESEC/FSE '11*, pages 278–288.

[48] K. Sen and G. Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *CAV '06*, pages 419–423.

[49] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE '05*, pages 263–272.

[50] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274.

[51] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, N. James, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS '08*, pages 1–25.

[52] SWI Prolog. `http://www.swi-prolog.org/`, 2014. Retrieved 24Mar.2014.

[53] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *SAS '16*, 2016.

[54] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA '11*, pages 189–206.

[55] N. Tillmann and J. De Halleux. Pex: White box test generation for .NET. In *TAP' 08*, pages 134–153, 2008.

[56] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In *OOPSLA '12*, pages 537–554.

[57] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *PLDI '13*, pages 287–296.

[58] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*, pages 13–.

[59] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test input generation for programs with pointers. In *TACAS '09*, pages 277–291.

[60] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA '04*, pages 97–107.

[61] D. A. Wheeler. SLOC Count User Guide. `http://www.dwheeler.com/sloccount/`, 2012. Retrieved 11 November 2012.