Reflection Analysis for Java: Uncovering More Reflective Targets Precisely

Jie Liu[1], Yue Li[1], Tian Tan[1], and Jingling Xue[1,2]
[1]School of Computer Science and Engineering, UNSW, Australia
[2]State Key Laboratory of Software Engineering, Computer School, Wuhan University, China

***Abstract*—Reflection, which is widely used in practice and abused by many security exploits, poses a significant obstacle to program analysis. Reflective calls can be analyzed statically or dynamically. Static analysis is more sound but also more imprecise (by introducing many false reflective targets and thus affecting its scalability). Dynamic analysis can be precise but often miss many true reflective targets due to low code coverage.**

We introduce MIRROR, the first automatic reflection analysis for Java that increases significantly the code coverage of dynamic analysis while keeping false reflective targets low. In its static analysis, a novel reflection-oriented slicing technique is applied to identify a small number of small path-based slices for a reflective call so that different reflective targets are likely exercised along these different paths. This preserves the soundness of pure static reflection analysis as much as possible, improves its scalability, and reduces substantially its false positive rate. In its dynamic analysis, these slices are executed with automatically generated test cases to report the reflective targets accessed. This significantly improves the code coverage of pure dynamic analysis. We evaluate MIRROR against a state-of-the-art dynamic reflection analysis tool, TAMIFLEX, by using 10 large real-world Java applications. MIRROR detects 12.5% – 933.3% more reflective targets efficiently (in 362.8 seconds on average) without producing any false positives. These new targets enable 5 – 174949 call-graph edges to be reachable in the application code.

I. Introduction

As one of the most widely adopted programming languages [1], Java has been a popular attack target. Java suffers still from serious security issues, with 87% of attack vectors for web exploits in 2013 [2] and 91% in 2014 [3]. A large variety of exploited vulnerabilities are related to reflection, a dynamic feature widely used in Java applications to enable their runtime behaviors to be examined or modified at runtime, which is abused by 45% of all exploits in the wild [4].

In practice, program analysis tools are invaluable for ensuring software quality and reliability. However, "you can't check code you don't see" [5]. Without analyzing reflection, bug detectors and security analysers may miss important program behaviors, because these tools do not have a complete view of the code (as many reflectively induced call-graph edges are missing). Therefore, reflection poses a major obstacle to bug detection and security analysis [6]–[9].

Reflective calls can be analyzed either statically or dynamically. Static analysis [6], [7], [10]–[15], which discovers reflective targets accessed at reflective calls via type inference, is often imprecise by reporting many false targets (and consequently, impairing scalability for some large applications). In contrast, dynamic analysis [16], [17], which instruments and records reflective targets accessed at reflective calls during program execution, can be both precise and efficient. As a result, bug detectors on finding, for example, data races [18], deadlocks [19] and property violations [20]), and security analysers on finding, for example, privacy leaks [21] and malicious functionalities [22], often resort to dynamic reflection analysis. However, analyzing reflection dynamically often misses many true reflective targets (due to low code coverage). This is especially the case when GUI applications are analyzed. For example, we observe that TAMIFLEX [17], the state-of-the-art dynamic reflection analysis, fails to find any new reflective target after a long sequence of GUI operations has been performed (on, for example, `findbugs-1.2.1`).

In this paper, we introduce MIRROR, the first automatic reflection analysis for Java that combines program slicing (static analysis) and test case generation (dynamic analysis) to uncover more reflective targets precisely. MIRROR is designed to assist dynamic reflection analysis (e.g., TAMIFLEX) to resolve more reflective targets with low false positives. Thus, MIRROR can discover effectively reflective targets that would otherwise be missed by TAMIFLEX in real-world applications and improve the code coverage of dynamic reflection analysis.

In MIRROR, its static analysis applies a novel reflection-oriented slicing technique to focus on the parts of the program relevant to a reflective call. Unlike traditional slicing [23], [24], which hardly scales to large object-oriented programs [25], [26], MIRROR identifies a small subgraph of the program's call graph that likely affects the execution of a reflective call and then computes a small number of small path-based slices in the subgraph so that potentially true yet different reflective targets are likely exercised at the reflective call along these different paths. This preserves the soundness of pure static reflection analysis as much as possible, improves its scalability, and reduces substantially its false positive rate.

In MIRROR, its dynamic analysis executes each path-based slice with automatically generated test cases to exercise the path and record the reflective targets accessed. This increases the code coverage of pure dynamic reflection analysis.

We have evaluated MIRROR against TAMIFLEX [17] by using a set of 10 large real-world Java programs. MIRROR detects 12.5% – 933.3% more reflective targets efficiently (in 362.8 seconds on average) with no false targets. These new reflective targets result in 5 – 174949 call-graph edges reachable in the application code of these programs.

With MIRROR, more reflective targets can be found pre-

2332-6549/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSRE.2017.36

cisely and quickly, making many previously missing call-graph edges visible to a variety of analysis tools. This enables bug detectors and security analyzers, for example, to identify more bugs and vulnerabilities effectively.

In summary, this paper makes the following contributions:

- We introduce MIRROR, the first automatic Java reflection analysis framework that combines static and dynamic analysis to resolve reflective calls in large codebases.
- We describe a reflection-oriented slicing technique that improves the scalability of traditional slicing for object-oriented programs (w.r.t. a reflective call). This technique is also potentially useful for supporting bug detection, program understanding, and verification.
- We describe a dynamic analysis technique for resolving reflective calls by combining automatic test case generation and program execution on path-based slices.
- We evaluate MIRROR (implemented in 10 KLOC of Java) by using 10 large real-world Java programs. MIRROR is critical in enabling their reflective calls to be analyzed efficiently and precisely with good soundness.

II. A MOTIVATING EXAMPLE

Our motivating example is a Java program given in Figure 1, which is abstracted from real-world applications `freecs-1.3` and `pmd-4.2.5` (used in our evaluation in Section IV). We focus on resolving the reflective target methods that may be called at `getM.invoke(o, null)` in line 42. While MIRROR works on the Jimple IR (Intermediate Representation) in SOOT [27], we illustrate it by using the high-level statements in Java in order to ease understanding.

In line 17, a class metaobject pointed to by `this.clzObj` is created by calling `Class.forName(cName)` to represent the class named by `cName`. Here, `cName` is either "`content.HTTPRequest`" (line 5) or a non-constant string read from the command line (line 7). In line 41, an object `o` is created reflectively as an instance of `this.clzObj`. Then a method object, pointed to by `getM`, is created by calling `getMethod()` in line 37 or 39 to represent a method from `this.clzObj` with its name being "`toString`" or "`getUrl`". In line 42, this method is called reflectively on the receiver object `o` with the actual argument null. There are five potential reflective targets: `getUrl()` in class `content.HTTPRequest` and the four `toString()` methods in the four subclasses, `SimpleRenderer`, `XMLRenderer`, `CSVRenderer` and `VSRenderer`, of interface `pmd.cpd.Renderer`.

A. *Existing Approaches*

Static analysis [6], [7], [11], [28]–[30] attempts to resolve reflective targets of `getM.invoke(o, null)` in line 42 by conducting type inference. By keeping track of string constants, we know that `getM` represents a method named `toString()` (line 37) or `getUrl()` (line 39). In addition, as `o` may be an instance of class `content.HTTPRequest` (line 5), `toString()` and `getUrl()` from class `content.HTTPRequest` are the

```
 1 public class Server {
 2   Class clzObj;
 3   boolean infoChanged;
 4   public static void main(String args[]) {
 5     String className = "content.HTTPRequest";
 6     if (args.length > 0) {
 7       className = args[0];
 8     }
 9     if (args.length > 5)
10       System.out.println("No config info");
11     Server srv = new Server();
12     srv.changeInfo(className);
13     srv.readConfig();
14     srv.initServer();
15   }
16   public void changeInfo(String cName) {
17     this.clzObj = Class.forName(cName);
18     this.infoChanged = false;
19   }
20   public void readConfig() {
21     loadCommand();
22     this.infoChanged = true;
23     FileMonitor.getFileMonitor().addReloadable(this);
24   }
25   public void initServer() {
26     created();
27   }
28   public void created() {
29     loadCommand();
30   }
31   public void loadCommand() {
32     String cmdStr = null;
33     if (this.infoChanged == false) {
34       Method getM = null;
35       Class clz = this.clzObj;
36       if (pmd.cpd.Renderer.class.isAssignableFrom(clz))
37         getM = clz.getMethod("toString");
38       else if (content.HTTPRequest.class.isAssignableFrom(clz))
39         getM = clz.getMethod("getUrl");
40       else throw new ErrorClassTypeException(...);
41       Object o = clz.newInstance();
42       cmdStr = (String) getM.invoke(o, null);
43     }
44   }
45 }
```

Fig. 1: A Java program.

two potential targets in line 42. Finally, as `o` may also be an instance of any unknown class (line 7), `getUrl()` or `toString()` in any class is also a possible target. Thus, a dilemma emerges. Including all these targets improves the soundness of the analysis but introduces many false targets, making the analysis imprecise or unscalable. Conversely, ignoring all these targets may cause some true targets to be missed. In MIRROR, we avoid this dilemma by reporting only the reflective targets observed dynamically on the path-based slices that are statically computed for a given reflective call.

Dynamic analysis [16], [17] instruments and records the reflective targets invoked by `getM.invoke(o, null)` in line 42 during program execution. Unless all test inputs are exhausted, which is impractical for many applications, especially GUI applications, some reflective targets will be missed. For example, `pmd-4.2.5`, from which our example is partly abstracted, does not come with any test case for exercising

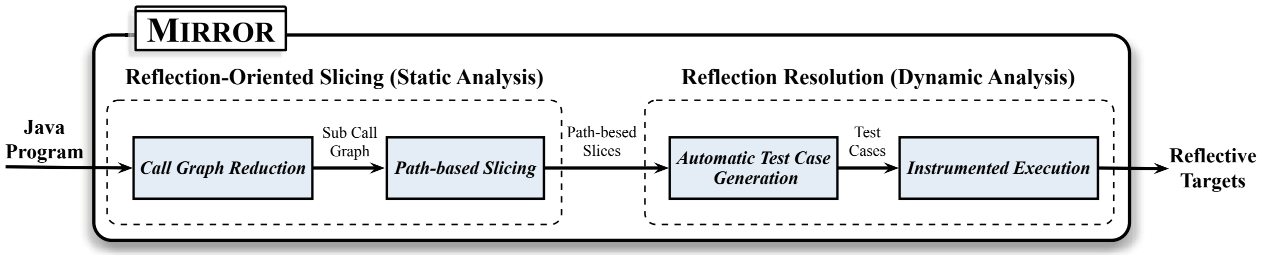


Fig. 2: The MIRROR framework for resolving reflective calls by combining static and dynamic analysis.

`className` (line 7). Therefore, a dynamic analysis tool may fail to find the four `toString()` target methods provided in the four aforementioned subclasses of `pmd.cpd.Renderer`. In general, dynamic analysis cannot resolve reflective calls that are not encountered during program execution. In MIRROR, however, we can still handle such reflective calls by combining automatic test case generation and program execution.

B. The MIRROR *Approach*

Figure 2 gives an overview. Given a Java program, MIRROR analyzes its reflective calls individually. Below we describe our approach by focusing on `getM.invoke(o, null)` in line 42 of Figure 1. We will highlight the functionalities of its four stages, with the first two forming the static analysis phase ("Reflection-Oriented Slicing") and the last two forming the dynamic analysis phase ("Reflection Resolution"). We will also explain some challenges faced, our solutions, the motivations behind, and the tradeoffs made.

1) **Reflection-Oriented Slicing (Static Analysis):** As discussed earlier, pure static reflection analysis may introduce many false reflective targets, making it unscalable for some programs. Given a reflective call to resolve, one straightforward remedy is to restrict the analysis to its backward slice comprising the statements on which the reflective call data- or control-depends. Unfortunately, such traditional slicing [23], [24] does not scale to large object-oriented programs [28], [31]–[33], as it operates on the entire call graph of the program. For the reflective call in line 42, its backward slice consists of all the methods in Figure 1, comprising all the statements except the three in lines 9, 10 and 23.

Ideally, we would like to find the smallest backward slice for a reflective call so that its different paths trigger its different reflective targets. Achieving such soundness and precision efficiently is too challenging to be practical. In this paper, we introduce reflection-oriented slicing that strikes a good balance among efficiency, soundness and precision by leveraging the following key observation about reflection usage.

Observation 1. *Given a reflective call $\mathcal{R}$ in the program, the variables that appear in all the conditionals affecting the execution of $\mathcal{R}$ are usually defined in the same set of methods that contain the du-chains (i.e., def-use chains) for $\mathcal{R}$.*

Let $D_{\mathcal{R}}$ be the set of du-pairs (i.e., def-use pairs) of the form $s \Rightarrow s'$, where a variable defined at s is used at s', such that these data dependences form the du-chains for $\mathcal{R}$. In reflection-oriented slicing, we obtain first a subgraph of the program's call graph that contains $D_{\mathcal{R}}$ (by considering data dependences) and then a small number of path-based slices for $\mathcal{R}$ on this subgraph (by considering also control dependences).

a) **Stage 1. Call Graph Reduction:** Instead of operating on the entire call graph of the program as in traditional slicing [23], [24], MIRROR restricts itself to a small subgraph.

For the reflective call in line 42, we have:

$$D_{42} = \{4 \Rightarrow 7, 7 \Rightarrow 12, 5 \Rightarrow 12, 12 \Rightarrow 17, 17 \Rightarrow 35, 35 \Rightarrow 37, \\ 35 \Rightarrow 39, 35 \Rightarrow 41, 37 \Rightarrow 42, 39 \Rightarrow 42, 41 \Rightarrow 42\} \quad (1)$$

where each statement is identified by its line number. In Figure 1, all these statements, which affect the execution of line 42 in a data-dependent manner, are underlined. The definition of `getM = null` in line 34 is disregarded since it cannot trigger any target at the reflective call in line 42.

Given $D_{\mathcal{R}}$, a subgraph, denoted $G_{\mathcal{R}}$, is built so that, for every du-chain in $D_{\mathcal{R}}$, $G_{\mathcal{R}}$ contains a sequence of method calls along which the du-chain holds. In our example, its call graph is given in Figure 3(a) and the subgraph G_{42} is given in Figure 3(b).

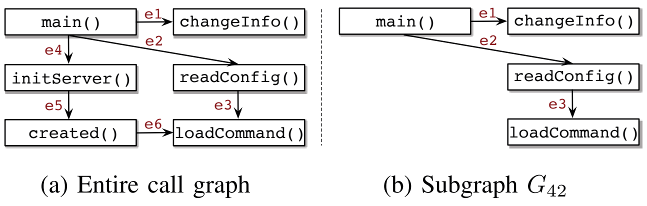


Fig. 3: Call graph reduction for Figure 1.

There can be many such subgraphs to choose from. For the subgraph in Figure 3(b), replacing its call-graph edges $e2$ and $e3$ by $e4$, $e5$ and $e6$ in Figure 3(a) yields another larger subgraph. We do not aim to find the smallest $G_{\mathcal{R}}$. Instead, we will build $G_{\mathcal{R}}$ by performing BFS in the program's call graph in order to keep $G_{\mathcal{R}}$ as small as possible, thereby improving the scalability of the subsequent path-based slicing stage. This tradeoff may still make call-graph reduction unscalable for some reflective calls, affecting the soundness of MIRROR (due to the inherent complexity of slicing, in general).

b) **Stage 2. Path-based Slicing:** For a reflective call $\mathcal{R}$, its different targets may be defined along different paths. We are therefore motivated to partition $D_{\mathcal{R}}$ into different du-chain groups so that one group contains exactly one definition for every variable used at $\mathcal{R}$. Clearly, all the paths that contain the same du-chains for $\mathcal{R}$ must trigger the same set of reflective targets at $\mathcal{R}$. Therefore, only one representative path needs to be selected for each du-chain group. As the partition obtained statically this way is not guaranteed to be the coarsest at run time, different du-chain groups may still trigger the same set of reflective targets at $\mathcal{R}$.

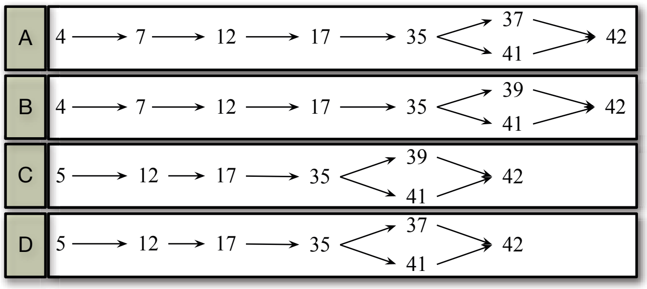


Fig. 4: Partitioning of D_{42} into du-chain groups.

Figure 4 gives a partition of D_{42} in Equation (1) into four du-chain groups, A, B, C and D. Note that $5 \Rightarrow 12$ and $7 \Rightarrow 12$ cannot appear in the same group since `className` is defined in lines 5 and 7 but used in line 12. Similarly, $37 \Rightarrow 42$ and $39 \Rightarrow 42$ cannot appear in the same group since `getM` is defined in lines 37 and 39 but used in line 42.

Given a du-chain group $X_{\mathcal{R}}$ of $D_{\mathcal{R}}$, we will find a representative path that contains all the du-pairs in $X_{\mathcal{R}}$ such that for every $s_1 \Rightarrow s_2 \in X_{\mathcal{R}}$, if $s'_1 \Rightarrow s_2 \in D_{\mathcal{R}} \setminus X_{\mathcal{R}}$, then s'_1 will not appear on the path. This ensures that s_1 is the only definition for s_2 on this path. We will achieve this by performing BFS on the ICFG (inter-procedural Control Flow Graph) of the program restricted to $G_{\mathcal{R}}$. If such a path does not exist, $X_{\mathcal{R}}$ is ignored. This can happen when $X_{\mathcal{R}}$ contains two du-pairs that appear in two mutually exclusive branches and thus cannot hold simultaneously during program execution.

A	4, 6-9, 11, 12, 17, 18, 13, 21, 32-37, 41, 42
B	4, 6-9, 11, 12, 17, 18, 13, 21, 32-36, 38, 39, 41, 42
C	4-6, 9, 11, 12, 17, 18, 13, 21, 32-36, 38, 39, 41, 42
D	4-6, 9, 11, 12, 17, 18, 13, 21, 32-37, 41, 42

Fig. 5: Representative paths for the du-chain groups in Figure 4.

Figure 5 gives the representative paths found for the four du-chain groups in Figure 4. In each case, there are two paths due to line 9. The shorter one that contains no statements in the **if** branch in lines 9 – 10 is selected. Note that for the du-chain groups C and D in Figure 4, with each containing the definition of `className` in line 5, the other definition of `className` in line 7 is skipped. Thus, the **else** branch of the **if** statement in lines 9 – 10 will be followed.

Finally, for each representative path selected for a du-chain group $X_{\mathcal{R}}$ partitioned from $D_{\mathcal{R}}$, we obtain a path-based slice by applying traditional slicing [23], [24] to $\mathcal{R}$. However, we only consider the data and control dependences for the statements on the path and restrict the slice to $G_{\mathcal{R}}$.

A	4, 6-8, ~~9~~, 11, 12, 13, 17, 18, 21, 22, ~~32~~, 33, ~~34~~, 35-37, 41, 42
B	4, 6-8, ~~9~~, 11, 12, 13, 17, 18, 21, 22, ~~32~~, 33, ~~34~~, 35, 36, 38, 39, 41, 42
C	4, 5, ~~6~~, ~~9~~, 11, 12, 13, 17, 18, 21, 22, ~~32~~, 33, ~~34~~, 35, 36, 38, 39, 41, 42
D	4, 5, ~~6~~, ~~9~~, 11, 12, 13, 17, 18, 21, 22, ~~32~~, 33, ~~34~~, 35-37, 41, 42

Fig. 6: Path-based slices for the paths in Figure 5.

Figure 6 gives the slices found for the four representative paths in Figure 5. For each path, the statements on the path that are irrelevant to line 42 with respect to this path are crossed out. In addition, line 22 is the only (new) statement added.

2) **Reflection Resolution (Dynamic Analysis):** For each path-based slice obtained for $\mathcal{R}$, we generate its test cases and discover its reflective targets at $\mathcal{R}$ by instrumented execution.

a) **Stage 3. Automatic Test Case Generation:** Each path-based slice has only one execution path. As in symbolic execution [34]–[38], a path condition is collected that comprises all the constraints affecting the execution of the path. However, unlike the prior work, MIRROR models a variety of object-oriented features such as `instanceof`, `isAssignableFrom` and type casts in order to generate test cases more comprehensively. In general, we will generate a set of different test inputs that satisfy the path condition so that different reflective targeted can be captured.

A	Args[] ∈ {{"pmd.cpd.SimpleRenderer"}, {"pmd.cpd.XMLRenderer"}, {"pmd.cpd.CSVRenderer"}, {"pmd.cpd.VSRenderer"}}
B	Args[] = {"content.HTTPRequest"}
C	No input needed
D	Infeasible path

Fig. 7: Test inputs generated for the slices in Figure 6.

Figure 7 give the test inputs generated. For slice A, four test cases for `Args[0]` at line 4 are generated due to `pmd.cpd.Renderer.class.isAssignableFrom(clz)` in its path condition. Here, `SimpleRenderer`, `XMLRenderer`, `CSVRenderer` and `VSRenderer` are the four subclasses of interface `pmd.cpd.Renderer`. For slice B, one test case is generated for `Args[0]` at line 4 due to `content.HTTPRequest.class.isAssignableFrom(clz)`. For slice C, no input is needed as all variables are well initialized. For slice D, its path is infeasible due to two inconsistent constraints, `className="content.HTTPRequest"` and `pmd.cpd.Renderer.class.isAssignableFrom(clz)`.

Let us now examine the implications of Observation 1 on the precision of MIRROR. If a conditional, say, `x == 0` that affects the execution of $\mathcal{R}$ involves a definition of x outside $G_{\mathcal{R}}$, then MIRROR will generate a value of x so that x == 0 holds. If this conditional never evaluates to true during program execution, then MIRROR may report some false reflective targets along this path. This has never happened to a set of 10 large Java programs evaluated, indicating that Observation 1 usually holds in real-world applications.

b) **Stage 4. Instrumented Execution:** For each path-based slice obtained for $\mathcal{R}$, we generate an executable program by adding some missing variable declarations. For example, every slice in Figure 6 misses the declaration for `getM`. We then execute each slice with its test cases generated earlier to resolve the reflective targets at $\mathcal{R}$ along its associated path.

Let us consider the reflective call in line 42. For slice A, the four `toString()` target methods in the four subclasses, `SimpleRenderer`, `XMLRenderer`, `CSVRenderer` and `VSRenderer`, of interface `pmd.cpd.Renderer` are found. For slice B, `content.HTTPRequest.getUrl()` is discovered. For the slice C, this same target is found.

III. The Mirror Algorithms

We describe the algorithms for realizing the four components in MIRROR (Figure 2). MIRROR operates on the ICFG of the program expressed in a three-address IR, by making use of the program's call graph G and def-use chains available.

Given a program, MIRROR analyzes the reflective calls reachable from its `main()` individually. For a given reflective call $\mathcal{R}$, $D_{\mathcal{R}}$ represents the set of du-chains for $\mathcal{R}$. Our call-graph reduction algorithm will reduce G to a substantially smaller subgraph $G_{\mathcal{R}}$ that contains a sequence of method calls to establish every du-chain in $D_{\mathcal{R}}$. Our path-based slicing algorithm will build a small number of small path-based slices on $G_{\mathcal{R}}$ with its paths leading potentially to all the possible targets accessed at $\mathcal{R}$. Our automatic test case generator generates the test cases for exercising a (feasible) path-based slice. Finally, our instrumentator instruments and executes a given slice to report the reflective targets accessed at $\mathcal{R}$.

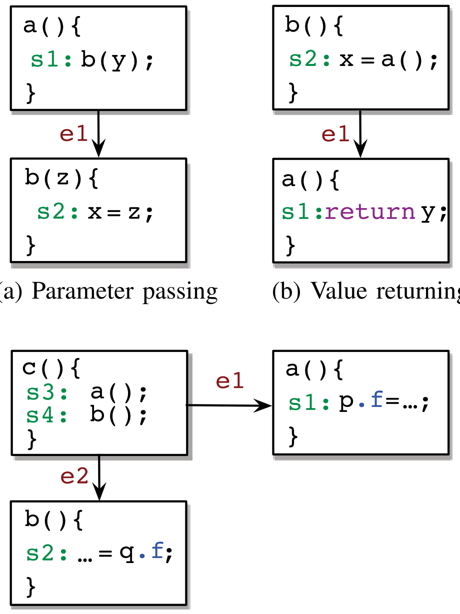
A. Call Graph Reduction

Given a reflective call $\mathcal{R}$, we will reduce the program's call graph G to a subgraph $G_{\mathcal{R}}$, which is substantially smaller but still allows every du-chain in $D_{\mathcal{R}}$ to hold. Thus, $G_{\mathcal{R}}$ suffices to trigger all the possible reflective targets at $\mathcal{R}$ while keeping the number of false targets reported to a minimum due to Observation 1. This also enables MIRROR to perform its subsequent reflection-oriented slicing on a small sub-call graph, thereby improving the scalability of traditional slicing (especially for large object-oriented programs), which is performed on the entire call graph of the program instead.

We will build $G_{\mathcal{R}}$ so that for every du-chain in $D_{\mathcal{R}}$, $G_{\mathcal{R}}$ contains a sequence of method calls for the du-chain to hold. It suffices to consider only the interprocedural du-pairs (i.e., the du-pairs spanning two different methods) in $D_{\mathcal{R}}$, since the resulting $G_{\mathcal{R}}$ will naturally include all the intraprocedural du-pairs in $D_{\mathcal{R}}$. To construct $G_{\mathcal{R}}$, we grow it incrementally by processing all the interprocedural du-pairs in turn. Therefore, how to find a sub-call graph to establish one interprocedural du-pair $s_1 \Rightarrow s_2$ is central to our algorithm. To this end, we make use of the following two functions:

- ***Connect***(s_1, s_2) returns a sub-call graph so that $s_1 \Rightarrow s_2$.
- ***Common***(s_1, s_2) returns a statement reaching s_1 and s_2.

There are three types of interprocedural du-pairs. We discuss how to build the two functions, as illustrated in Figure 8.



```
a(){
 s1:b(y);
}
```

```
b(z){
 s2:x=z;
}
```

(a) Parameter passing

```
b(){
 s2:x=a();
}
```

```
a(){
s1:return y;
}
```

(b) Value returning

```
c(){
s3: a();
s4: b();
}
```

```
a(){
 s1:p.f=…;
}
```

```
b(){
s2:…=q.f;
}
```

(c) Field access

Fig. 8: Building *Connect*(s_1, s_2) and *Common*(s_1, s_2) for three types of interprocedural du-pairs $s_1 \Rightarrow s_2$.

(1) Parameter Passing (Figure 8(a)). $s_1 \Rightarrow s_2$ denotes a parameter-passing dependence, where s_1 is a call statement in method $a()$ that passes an argument used at s_2 in method $b()$. As a result, *Connect*$(s_1, s_2) = \{a() \rightarrow b()\}$ and *Common*$(s_1, s_2) = s_1$.

(2) Return value (Figure 8(b)) $s_1 \Rightarrow s_2$ denotes a value-returning dependence, where s_2 is a call statement in method $b()$ that receives a value returned from s_1 in method $a()$. Thus, *Connect*$(s_1, s_2) = \{a() \rightarrow b()\}$ and *Common*$(s_1, s_2) = s_2$.

(3) Field Access (Figure 8(c)) $s_1 \Rightarrow s_2$ denotes a field-related dependence, where s_1 is a store $p.f = \cdots$ in method $a()$ and s_2 is a load $\cdots = q.f$ in method $b()$. Here, $p.f$ and $q.f$ are aliases as p and q may point to a common object. This also includes the special cases when s_1 appears directly in $c()$ (as if the call to $a()$ at s_3 is inlined) and/or when s_2 appears directly in $c()$ (as if the call to $b()$ at s_4 is inlined).

We compute *Connect*(s_1, s_2) by performing BFS on the program's call graph G backwards, starting from the use s_2. We will stop at the first method m such that

1) m is a direct or indirect caller of $b()$ or contains s_2 (as is the case when s_4 is replaced by s_2), and

2) m is a direct or indirect caller of $a()$ or contains s_1 (as is the case when s_3 is replaced by s_1).

Then *Connect*(s_1, s_2) comprises the two call-chains, one from m to $b()$ and one from m to $a()$, both computed by BFS. In Figure 8(c), m is found to be $c()$. So *Connect*$(s_1, s_2) = \{c() \to a(), c() \to b()\}$.

Common(s_1, s_2) is s_1 if m contains s_1 or is the call statement in m that calls (directly or indirectly) a method that contains s_1. In Figure 8(c), *Common*$(s_1, s_2) = s_3$. (If we modify the example by replacing s_3 by s_1, then *Common*$(s_1, s_2) = s_1$.)

In theory, *Connect*(s_1, s_2) is not the smallest. In practice, however, *Connect*(s_1, s_2) is nearly so due to BFS used, as all the reflection-related statements are typically used together.

Algorithm 1: Call Graph Reduction.

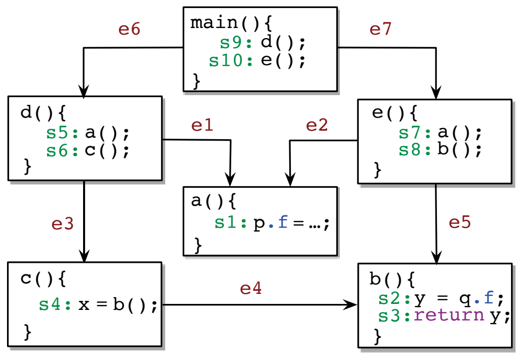
```
Input  : A reflective call site $\mathcal{R}$
Output: $G_\mathcal{R}$
Function BuildSubgraph()
    $G_\mathcal{R}$ := {the method containing $\mathcal{R}$};
    foreach $s_1 \Rightarrow s_2 \in D_\mathcal{R}$ do
        $visited(s_1 \Rightarrow s_2)$ := false;
    foreach $s \Rightarrow \mathcal{R} \in D_\mathcal{R}$ do
        RecBuild($\perp, s \Rightarrow \mathcal{R}$);
    return $G_\mathcal{R}$
Procedure RecBuild($u, s_1 \Rightarrow s_2$)
    $visited(s_1 \Rightarrow s_2)$ := true;
    if $s_1 \Rightarrow s_2$ spans two distinct methods then
        if $u == \perp$ then  $u := s_2$;
        $G_{s_1 \Rightarrow u} = Connect(s_1, u)$;
        $G_\mathcal{R} = G_\mathcal{R} \cup G_{s_1 \Rightarrow u}$;
        $c_{s_1 \Rightarrow u} = Common(s_1, u)$;
    foreach $s_3 \Rightarrow s_1 \in D_\mathcal{R}$ do
        if $!visited(s_3 \Rightarrow s_1)$ then
            RecBuild($c_{s_1 \Rightarrow u}, s_3 \Rightarrow s_1$);
    return;
```

Let us now describe *BuildSubGraph()*, in Algorithm 1, that builds $G_\mathcal{R}$ for a reflective call $\mathcal{R}$. We tag a du-pair as *visited* in the standard manner to deal with dependence cycles. One simple-minded but incorrect approach would compute:

$$G^{\text{err}}_\mathcal{R} = \bigcup_{s_1 \Rightarrow s_2 \in D_\mathcal{R} \text{ is interprocedural}} Connect(s_1, s_2) \qquad (2)$$

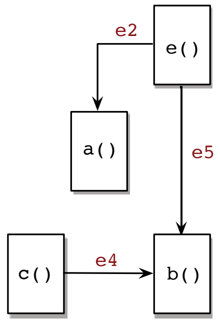
While $G^{\text{err}}_\mathcal{R}$ contains the statements in $D_\mathcal{R}$, some du-chains may no longer be preserved, as some caller-callee relations are missing. Consider Figure 9. The program's call graph is given in Figure 9(a), where s_4 symbolizes $\mathcal{R}$. Suppose $D_\mathcal{R} = \{s_1 \Rightarrow s_2, s_2 \Rightarrow s_3, s_3 \Rightarrow s_4\}$. For the du-chain $s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4$, $s_1 \Rightarrow s_2$ and $s_3 \Rightarrow s_4$ are interprocedural and $s_2 \Rightarrow s_3$ is intraprocedural. Due to the recursive nature of our algorithm, it suffices to use this du-chain to explain how our algorithm works and argue for its correctness.

$G^{\text{err}}_\mathcal{R} = Connect(s_1, s_2) \cup Connect(s_3, s_4) = \{e_2, e_5\} \cup \{e_4\}$ given in Figure 9(b) is incorrect. As $e()$ cannot reach $c()$ in $G^{\text{err}}_\mathcal{R}$, $G^{\text{err}}_\mathcal{R}$ does not contain a sequence of method calls that allows the du-chain $s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4$ to be established.

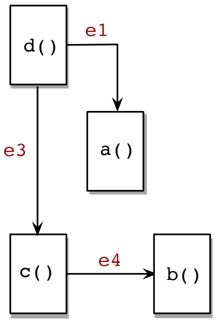


```
main(){
  s9: d();
  s10: e();
}
d(){
  s5: a();
  s6: c();
}
e(){
  s7: a();
  s8: b();
}
a(){
  s1: p.f = …;
}
c(){
  s4: x = b();
}
b(){
  s2: y = q.f;
  s3: return y;
}
```

(a) Call graph



(b) Incorrect $G^{\text{err}}_\mathcal{R}$



(c) $G_\mathcal{R}$

Fig. 9: Construction of incorrect and correct $G_\mathcal{R}$.

Figure 9(c) gives the subgraph $G_\mathcal{R}$ constructed by our algorithm. We first compute $G_{s_3 \Rightarrow s_4} = Connect(s_3, s_4) = \{e_4\}$ and $c_{s_3 \Rightarrow s_4} = Common(s_3, s_4) = s_4$. We then compute $G_{s_1 \Rightarrow s_4} = Connect(s_1, s_4) = \{e_1, e_3\}$ and $c_{s_1 \Rightarrow s_4} = Common(s_1, s_4) = s_5$. By construction, the following two facts are true. (1) In $G_{s_3 \Rightarrow s_4}$, s_4 reaches the method $b()$ that contains s_3. (2) In $G_{s_1 \Rightarrow s_4}$, s_1 can reach s_4. As $s_2 \Rightarrow s_3$ is intraprocedural, s_2 and s_3 reside in the same method. Thus, in $G_\mathcal{R}$, s_1 can reach $b()$ that contains also s_2. By computing *Connect*(s_1, s_4) instead of *Connect*(s_1, s_2), $s_1 \Rightarrow s_2$ is also respected. As shown, $d()$ can trigger a sequence of method calls, $s_5 : a()$, $s_6 : c()$ and $s_4 : x = b()$, so that the du-chain $s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4$ holds.

Example 1. *D_{42} is given in Equation (1), which contains two interprocedural du-pairs, $12 \Rightarrow 17$ and $17 \Rightarrow 35$. By applying Algorithm 1, we obtain Connect$(17, 35) = \{e_1, e_2, e_3\}$ and Common$(17, 35) = \{12\}$. Finally, Connect$(17, 35) = \{e_1, e_2, e_3\}$ is the subgraph $G_\mathcal{R}$ obtained in Figure 3.* □

B. Path-based Slicing

Given $D_\mathcal{R}$ that contains the du-chains for a reflective call $\mathcal{R}$, we generate its path-based slices in three steps.

1) Partitioning $D_\mathcal{R}$: We partition $D_\mathcal{R}$ into du-chain groups so that in one group, every variable has exactly one definition, starting from the variables used at $\mathcal{R}$. This can be done easily by traversing the du-chains backwards from $\mathcal{R}$ by separating multiple definitions of a variable in different groups.

Example 2. *For the reflective call at line 42 in Figure 1, D_{42} in Equation (1) is partitioned into the four du-chain groups given in Figure 4, where each variable has one definition.* □

2) Finding Representative Paths: For a given du-chain group $X_\mathcal{R}$ partitioned from $D_\mathcal{R}$, all the paths sharing $X_\mathcal{R}$ will cause the same set of reflective targets to be accessed at $\mathcal{R}$. Thus, it suffices to consider just one of these paths. In MIRROR, we will find one representative path $P_{X_\mathcal{R}}$ for $X_\mathcal{R}$ in the program's ICFG restricted $G_\mathcal{R}$, which is a small subgraph of the program's call graph found in call graph reduction, by performing BFS, starting from $\mathcal{R}$. $P_{X_\mathcal{R}}$ is a path-specific to $X_\mathcal{R}$. For every statement in $X_\mathcal{R}$, the path includes only its definition in $X_\mathcal{R}$ but excludes its other definitions that appear in the other du-chain groups.

We can find $P_{X_\mathcal{R}}$ by applying *SelectPath()* in Algorithm 2. The basic idea is simple. During BFS, the shortest path p_s from $\mathcal{R}$ to any statement s visited in the program's ICFG restricted to $G_\mathcal{R}$ (line 21) is maintained (line 10). If $P_{X_\mathcal{R}}$ is found (lines 11 – 14), then we are done. If the statements in $X_\mathcal{R}$ that are not yet encountered cannot reach s, we give up this path (lines 15 – 17). If $s \Rightarrow u \in D_\mathcal{R} \setminus X_\mathcal{R}$ is downward-exposed for a statement u in p_s, then we are traversing along a wrong direction (lines 18 – 20), since $P_{X_\mathcal{R}}$ must include only the single definition of u in $X_\mathcal{R}$ rather than $D_\mathcal{R} \setminus X_\mathcal{R}$. In lines 21 – 23, we perform our traversal in BFS, by avoiding visiting control-flow cycles repeatedly. In line 21, *pred* represents the standard predecessor function for a directed graph. Finally, in line 5, we remove non-downward-exposed definitions, which are in $D_\mathcal{R} \setminus X_\mathcal{R}$, as they are redundant (as illustrated below).

Example 3. *For the four du-chain groups in Figure 4, we can apply SelectPath to find their representative paths in Figure 5. Let us consider the du-chain group A. Consider the code in Figure 1. For `getM` used in line 42, its definition is given in line 37. For `className` used in line 12, its definition is given in line 7. When constructing its representative path starting from line 42, we will not include line 39, since this will make the definition of `getM` in line 39 downward-exposed (to line 42). Towards the end of the BFS traversal, line 5 in Figure 1 will be visited. The representative path built so far is "6 – 9, 11, 12, 17, 18, 13, 21, 32 – 37, 41, 42". As the definition of `className` in line 7 is already included, the other definition `className` in line 5 is initially included by* RECSELECT *but removed in SelectPath since its not downward-exposed, i.e., redundant. Thus, the final path found is "4, 6 – 9, 11, 12, 17, 18, 13, 21, 32 – 37, 41, 42".* □

3) Generating Path-based Slices: For each representative path found for a reflective call $\mathcal{R}$, we apply traditional slicing [23], [24] to compute a backward slice from $\mathcal{R}$. However, only the data and control dependences for the statements in this path are considered, with the slice restricted to $G_\mathcal{R}$ only.

Algorithm 2: Representative Path Selection.

```
   Input  : A du-chain group $X_\mathcal{R}$ for $\mathcal{R}$
   Output: A representative path $P_{X_\mathcal{R}}$ for $X_\mathcal{R}$
1  Function SelectPath()
2      $P_{X_\mathcal{R}} = \emptyset$;
3      Enqueue($Q, \mathcal{R}$);
4      RECSELECT();
5      Remove all non-downwards-exposed definitions in
         $P_{X_\mathcal{R}}$;
6      return $P_{X_\mathcal{R}}$;
7  Procedure RECSELECT()
8      while $Q \neq \emptyset$ do
9          $s$ = Dequeue($Q$)
10         Let $p_s$ be the (BFS) path maintained for $s$;
11         if $p_s$ contains all statements in $X_\mathcal{R}$ then
12             $P_{X_\mathcal{R}} = p_s$;
13             $Q = \emptyset$;
14             return;
15         Let $T$ be the set of statements in $X_\mathcal{R}$ but not in
             $p_s$;
16         if $\exists\ t \in T$ : $t$ does not reach $s$ then
17             return;
18         if $\exists\ u \in p_s : s \Rightarrow u \in D_\mathcal{R} \setminus X_\mathcal{R}$ then
19             if $\nexists\ s' \in p_s : s' \Rightarrow u \in X_\mathcal{R}$ then
20                 return;
21         foreach $s' \in pred(s)$ in the ICFG restricted to
             $G_\mathcal{R}$ do
22             if $s'$ was not previously enqueued then
23                 Enqueue($Q, s'$)
24         RECSELECT();
25     return;
```

Example 4. *For the four representative paths in Figure 5, their corresponding path-based slices are given in Figure 6.* □

C. Automatic Test Case Generation

Each path-based slice exhibits one single path. A path condition that consists of all the constraints governing the execution of the path is collected and solved to find all the test inputs for exercising the path. There are some SMT solvers around [39]–[41]. However, we have written one ourselves in order to handle a variety of object-oriented constraints more comprehensively, including `instanceof`, `isAssignableFrom`, and type casts.

Example 5. *For the four path-based slices generated in Figure 6, their corresponding test cases are given in Figure 7. For the path-based slice named A, some path constraints are `args.length > 0`, `className = args[0]` (an assignment), `this.infoChanged = false` (an assignment), `this.infoChanged == false`, and `pmd.cpd.Renderer.class.isAssignableFrom(clz)`. Solving this path condition yields the four test inputs given in Figure 7.* □

D. Instrumented Execution

For each path-based slice generated for a du-chain group $X_{\mathcal{R}}$, we will create an executable program by adding some missing declaration statements to the slice. In addition, we will synthesize a pseudo `main()` method to call all the source methods (with no predecessors) in $G_{\mathcal{R}}$. By construction, for every du-chain in $X_{\mathcal{R}}$, its path-based slice always contains a source method that can make a sequence of method calls so that the du-chain holds (Section III-A).

In Figure 9, the pseudo `main()` synthesized will call $d()$, since $d()$ reaches the other three methods $a()$, $c()$ and $b()$, so that the du-chain $s_1 \Rightarrow s_2 \Rightarrow s_3 \Rightarrow s_4$ will be established.

Example 6. *For the four path-based slices in Figure 6, the pseudo `main()` is simply set as the original `main()`, which happens to be the only source node in G_{42} (Figure 3). The declaration statements such as the ones for `className` and `getM` will be added in the final executables generated.* □

IV. Evaluation

Our evaluation addresses three research questions:

- **RQ1.** Can Mirror assist Tamiflex [17], a state-of-the-art dynamic reflection analysis tool, to resolve more (true) reflective targets in real-world applications efficiently and precisely while maintaining a good degree of soundness?
- **RQ2.** Is Mirror's reflection-oriented slicing capable of avoiding many irrelevant methods (statements) introduced by traditional slicing and including only the relevant reflection-related methods (statements)?
- **RQ3.** Can Mirror's reflection resolution solve path conditions effectively during automatic test case generation?

a) **Implementation:** We have implemented Mirror in Soot [27], a static analysis framework for Java, in 10 KLOC of Java code. Our analysis operates on the ICFG of the program expressed in the Jimple IR constructed by Soot with its demand-driven context-sensitive pointer analysis. We make use of the program's call graph and def-use chains built this way. For each sliced program, we transform it into a jar file and report the reflective targets detected under the test inputs automatically generated by Mirror.

b) **Benchmarks:** We consider a set of 10 large, widely-used open-source Java applications evaluated under a large Java library, JDK 1.6.0_45, and test each with 8GB heap space.

As shown in Table I, our 10 open-source programs are selected from a wide range of application areas: `batik1.7` (a SVG toolkit), `findbugs-1.2.1` (a bug detector), `freecs1.3` (a chat server), `gruntspud-0.4.6-beta` (a graphical CVS client), `h2-1.3.172` (a database management system), `jEdit-5.1.0` (a software text editor), `jfreechart-1.0.19` (a chart library), `jftp-1.6` (a network browser), `pmd-4.2.5` (a source code analyzer), and `xalan-2.4.1` (a XSLT processor).

For each program, we list the number of classes, methods and Jimple statements reported by Soot that are reachable from its `main()` in both the application and library code. We will focus on analyzing `Class.newInstance()`, `Method.invoke()` and `Constructor.newInstance()`, the three widely used Java reflection API in the application code of a program.

TABLE I: Program characteristics

Program	#Classes	#Methods	#Statements
batik1.7	5148	32347	581106
findbugs-1.2.1	4692	28857	472647
freecs1.3	3714	24000	418731
gruntspud-0.4.6-beta	4945	31684	527433
h2-1.3.172	4139	30208	518189
jEdit-5.1.0	5135	34402	586647
jfreechart-1.0.19	4511	30596	534615
jftp-1.6	5208	35739	659927
pmd-4.2.5	3936	25068	422054
xalan-2.4.1	3483	21755	373008
Total	44911	294656	5094357

c) **Computing Platform:** Our plaform is an Intel i5-4570 3.20 GHz machine (running Windows 7) with 16GB of RAM. The analysis time of a program is the average of 5 runs.

A. RQ1: Reflection Analysis

Table II contains our main results. Mirror can assist a state-of-the-art dynamic reflection analysis tool, Tamiflex [17], to find significantly more reflective calls and (true) reflective targets in real-world applications efficiently. For the 10 programs evaluated, Mirror reports no false reflective targets, demonstrating also the validity of Observation 1.

TABLE II: Comparing Mirror and Tamiflex in terms of reflective calls and (true) reflective targets found.

Program	Tamiflex		Mirror		Tamiflex ∩ Mirror	
	#Ref Calls	#Targets	#Ref Calls	#Targets	#Ref Calls	#Targets
batik	3	4	1	6	1	1
findbugs	3	3	3	4	2	2
freecs	6	6	6	59	3	3
gruntspud	5	8	7	7	2	2
h2	7	19	10	18	4	12
jEdit	4	40	6	12	1	1
jfreechart	4	6	13	13	1	1
jftp	4	10	16	28	1	5
pmd	1	2	2	10	1	2
xalan	11	56	3	8	1	1
Total	48	154	67	165	17	30

When running Tamiflex on a program, some test cases are needed. For the six GUI programs, `findbugs`, `gruntspud`, `h2`, `jEdit`, `jfreechart` and `jftp`, we exercise their GUI to invoke their main functionalities just like a user does, in 5 minutes each. Take `jfreechart` as an example. We exercise its GUI by designing different charts and exporting them to files. There are four non-GUI programs. For `batik`, `pmd` and `xalan`, we have designed 10 test cases each guided by their user tutorials in order to exercise their main functionalities. When applying Mirror to a program, we focus on the reflective calls that are identified by Soot and analyzed scalably under a budget (discussed below). For `freecs`, a chat server, we start it up for it to read several configuration files (without providing any explicit input from us).

TABLE III: Efficiency and effectiveness of Mirror. For efficiency, the last column gives the analysis time spent by Mirror (in all its four stages). The second last column gives the analysis time spent by Soot on performing its pointer analysis and building the call graph, ICFG and def-use chains. For effectiveness, the results in Table II are further analyzed. For each type of Java reflection API considered, the number of new reflective calls and reflective targets found by Mirror relative to Tamiflex are given. The number of additional call-graph edges reachable from these new reflective targets are also given.

Program	Reflection API	Tamiflex		Mirror		Increased Call-Graph Edges		Analysis Times (secs)	
		#Calls	#Targets	#Calls	#Targets	App	App + Lib	Soot	Mirror
batik	Class.newInstance	3	4	(+)0	(+)5	(+)74	(+)637	179.6	310.2
	Method.invoke	N/A	N/A	(+)0	(+)0				
	Constructor.newInstance	N/A	N/A	(+)0	(+)0				
	Total	3	4	**(+)0**	**(+)5**				
findbugs	Class.newInstance	2	2	(+)0	(+)0	(+)22328	(+)134769	117.2	237.9
	Method.invoke	N/A	N/A	(+)1	(+)2				
	Constructor.newInstance	1	1	(+)0	(+)0				
	Total	3	3	**(+)1**	**(+)2**				
freecs	Class.newInstance	3	3	(+)1	(+)1	(+)728	(+)60965	70.5	282.1
	Method.invoke	1	1	(+)2	(+)55				
	Constructor.newInstance	2	2	(+)0	(+)0				
	Total	6	6	**(+)3**	**(+)56**				
gruntspud	Class.newInstance	2	5	(+)0	(+)0	(+)5	(+)120887	117.4	395.8
	Method.invoke	3	3	(+)5	(+)5				
	Constructor.newInstance	N/A	N/A	(+)0	(+)0				
	Total	5	8	**(+)5**	**(+)5**				
h2	Class.newInstance	2	10	(+)2	(+)2	(+)22	(+)360300	103.9	451.7
	Method.invoke	4	8	(+)4	(+)4				
	Constructor.newInstance	1	1	(+)0	(+)0				
	Total	7	19	**(+)6**	**(+)6**				
jEdit	Class.newInstance	1	1	(+)3	(+)3	(+)174949	(+)498536	133.9	375.4
	Method.invoke	2	10	(+)2	(+)8				
	Constructor.newInstance	1	29	(+)0	(+)0				
	Total	4	40	**(+)5**	**(+)11**				
jfreechart	Class.newInstance	N/A	N/A	(+)2	(+)2	(+)97590	(+)1042689	122.7	244.2
	Method.invoke	3	5	(+)9	(+)9				
	Constructor.newInstance	1	1	(+)1	(+)1				
	Total	4	6	**(+)12**	**(+)12**				
jftp	Class.newInstance	1	7	(+)13	(+)21	(+)94	(+)121745	157.2	950.6
	Method.invoke	3	3	(+)2	(+)2				
	Constructor.newInstance	N/A	N/A	(+)0	(+)0				
	Total	4	10	**(+)15**	**(+)23**				
pmd	Class.newInstance	1	2	(+)1	(+)8	(+)176	(+)150519	70.8	77.2
	Method.invoke	N/A	N/A	(+)0	(+)0				
	Constructor.newInstance	N/A	N/A	(+)0	(+)0				
	Total	1	2	**(+)1**	**(+)8**				
xalan	Class.newInstance	7	28	(+)2	(+)7	(+)264	(+)121568	65.7	303.1
	Method.invoke	4	28	(+)0	(+)0				
	Constructor.newInstance	N/A	N/A	(+)0	(+)0				
	Total	11	56	**(+)2**	**(+)7**				
Average		5	15	**(+)5**	**(+)14**	(+)29623	(+)261261	113.9	362.8

Looking at Table II, we find that Mirror is complementary to Tamiflex. Indeed, Mirror is actually designed to play this important role by filling a gap left by pure static and dynamic reflection analysis. For these two tools, neither is strictly more sound than the other. There are reflective calls that can be resolved by Tamiflex but not by Mirror (Tamiflex- Tamiflex ∩ Mirror). There are two reasons behind. First, there are 13 reflective calls that are invisible to Mirror, with 2 in `batik`, 1 in `findbugs` and 10 in `xalan`. This happens since Soot (and other static tools, in general) cannot soundly build the call graph for a program due to its inadequate modeling of dynamic class loading, reflection and native libraries, highlighting again the significance of reflection analysis in practice. Second, for the remaining reflective calls totaling 18 in Tamiflex- Tamiflex ∩ Mirror (visible to Mirror), Mirror's call-graph reduction is unscalable (due to the inherent complexity of program slicing), despite its significant better scalability than traditional slicing, as discussed later. This happens when their du-chains usually span across many methods, highlighting an important avenue for future research on reflection-oriented slicing.

Conversely, there are reflective calls that can be resolved by Mirror but missed by Tamiflex (Mirror- Tamiflex ∩ Mirror) since these calls are not reached during program execution. Mirror resolves more reflective calls in every application except `batik`. In particular, Mirror finds 56

TABLE IV: Comparing reflection-oriented and traditional slicing in terms of slice sizes and slicing times. For a program, a slice is measured by the number of methods/statements sliced for all its reflective calls considered (Column 4 of Table II). For the traditional slicer, "TO (x/y)" indicates that out of y reflective calls analyzed, x Times Out under the budget allocated.

Program	Budget (mins)	Reflection-Oriented Slicing			Traditional Slicing		
		#Mtds	#Stmts	Time (secs)	#Mtds	#Stmts	Time (secs)
batik	40	12	117	4.3	2080	16244	TO (1/1)
findbugs	55	12	91	3.4	3081	18893	TO (2/3)
freecs	90	71	746	78.9	2403	22032	TO (5/6)
gruntspud	50	11	129	33.6	9382	53304	TO (6/7)
h2	150	19	138	63.7	23757	162117	TO (7/10)
jEdit	130	12	124	7.1	7691	49452	TO (3/6)
jfreechart	65	21	211	3.6	40883	346618	TO (13/13)
jftp	150	63	1676	40.5	22956	189145	TO (12/16)
pmd	75	10	104	3.5	10	109	0.1 (0/2)
xalan	30	6	572	120.4	125	2295	TO (1/3)
Average	84	24	391	35.9	11237	86021	TO (5/7)

new targets in `freecs`, because it has activated a path with many reflective targets read from a file.

For all the 10 programs, TAMIFLEX finds 154 targets at 48 reflective calls and MIRROR finds 165 targets at 67 reflective calls. In total, MIRROR resolves 104.2% more reflective calls and 87.7% more reflective targets, measured by (((MIRROR − TAMIFLEX ∩ MIRROR)/TAMIFLEX) × 100)%. MIRROR discovers reflective targets that are not found by TAMIFLEX in all programs. For reflective calls, the percentage improvements range from 0.0% (`batik`) to 375.0% (`jftp`) with an average of 118.7%. For reflective targets, the percentage improvements range from 12.5% (`xalan`) to 933.3% (`freecs`) with an average of 208.9%.

Finally, Table III provides a breakdown of the results in Table II on the effectiveness of MIRROR, together with the analysis times of MIRROR for the 10 programs to demonstrate its efficiency. Due to the new reflective targets discovered, MIRROR enables between 5 (`gruntspud`) and 174949 (`jedit`) call-graph edges to be reached from these new targets in the application code. These numbers increase to 637 (`batik`) and 1042689 (`jfreechart`), respectively, if the library code is also included. For example, in `findbugs`, MIRROR finds only two new reflectively called methods, `findbugs.gui.FindBugsFrame.main()` and `findbugs.gui2.Driver.main()`. However, both methods can reach 22328 call-graph edges in the application code and 134769 call-graph edges if the library code is also considered. These call-graph edges found can significantly increase the code coverage of a variety of bug detection and security analysis tools.

As for efficiency, MIRROR spends an average of only 362.8 seconds (i.e., just above 6 minutes) on analyzing a program, by performing the fastest for `pmd` (in 77.2 seconds) and the slowest for `jftp` (in 950.6 seconds). It is not informative to compare TAMIFLEX and MIRROR in terms of their analysis times. MIRROR is fully automatic without requiring any user inputs. However, TAMIFLEX can only run under some given inputs. As discussed in Section I, when applying TAMIFLEX to analyze GUI applications, the user needs to spend a lot of human efforts to manually launch GUI operations (in order to generate user inputs for these applications).

B. RQ2: Reflection-Oriented Slicing

Table IV compares MIRROR's reflection-oriented slicer with a backward slicer that we have implemented in SOOT based on traditional slicing [23], [24] to reconfirm its known unscalability for object-oriented programs [25], [26]. For each program, Table II (Column 4) gives the number of reflective calls analyzed by MIRROR. For both slicers, the same time budget is allocated to a program (Column 2 in Table IV), which is calculated as follows. For each program, MIRROR computes path-based slices separately for its representative paths constructed at its reflective calls analyzed. Looking ahead in Figure 10, the number of representative paths per program ranges from 6 (`xalan`) to 30 (`h2` and `iftp`) with an average of 17. The average number of paths per reflective call ranges from 1 (`jfreechart`) to 8 (`batik`) with an average of 4. For MIRROR, each path is given a maximum of 5 minutes to be sliced. If a reflective call has N representative paths, then the traditional slicer is given a maximum of $5 * N$ minutes to slice from the reflective call.

As is clear from Table IV, MIRROR is far superior to the traditional slicer. On average, MIRROR spends 35.9 seconds while including *at least* 89.1% fewer methods and 87.4% fewer statements than the traditional slicer. Note the use of "at least" here. As indicated in Column "TO (x/y)", the traditional slicer times out in x out of y reflective calls analyzed in a program. Note that a slice produced by the traditional slicer includes the methods and statements obtained before it times out. The traditional slicer remains unscalable for these time-outed calls even if the budget is tripled, causing more methods and statements to be included in their slices.

Both slicers obtain similar results for `pmd` and `xalan`, since the reflective calls analyzed are close to `main()`. The traditional slicer is faster for `pmd` (with only one reflective call analyzed), since MIRROR needs to spend some time on building the sub-call graph even for analyzing just one call.

C. RQ3: Automatic Test Case Generation

Figure 10 evaluates the effectiveness of MIRROR's automatic test case generation. There are four bars for a program, with the first split into the last three. The first bar represents the number of representative paths in the program. The second and third give the number of feasible and infeasible paths detected, respectively. The last indicates the number of paths that cannot be solved by our constraint solver due to unmodeled constraints on file operations (reading from unknown files).

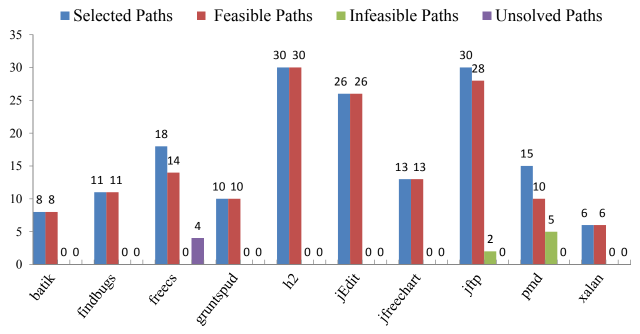


Fig. 10: Path solving during test case generation.

MIRROR is effective in solving path constraints effectively. In total, there are 167 paths. MIRROR fails to solve only 4 paths related to one reflective call, `getInstance.invoke(arg0)`, which resides in method `Freecs.Server.loadCommands()` in `freecs`, since they all involve reading some class names from an unknown file. However, MIRROR can successfully generate the test cases for 2 other paths related to the same reflective call, as the file name used is a string constant. For the remaining 161 paths, only 7 paths are found to be infeasible, indicating that our path selection strategy is viable.

MIRROR is efficient in its test case generation. MIRROR spends 7.66 seconds in `batik` and less than 2 seconds in each remaining one, with an average of only 1.3 seconds.

V. RELATED WORK

We review only the work mostly related to this paper.

a) **Static Reflection Analysis:** Several techniques exist for analyzing Java programs [6], [7], [10], [11], [15]–[17], [25], [42]. Livshits et al. [10] described the first static analysis for Java, which resolves reflective targets by tracking the flow of class/method/field names. Recently, Li. et al. introduced ELF [6] and SOLAR [7] to resolve reflective targets more effectively by applying sophisticated type inference.

Static reflection analysis has also been a subject of investigation for Android apps [43]–[45]. Li et al. [44] relied on string inference to resolve reflective calls in Android apps. Zhang et al. [45] presented RIPPLE, an approach to resolving reflective calls in Android apps in incomplete information environments.

The main challenge faced by static reflection analysis is how to balance the number of false reflective targets generated with the effectiveness realized in finding true targets. MIRROR tackles this challenge by statically analyzing only the program paths leading to a reflective call and dynamically detecting the reflective targets accessed along these paths.

b) **Dynamic Reflection Analysis:** There are some tools for Java programs [16], [17]. Hirzel et al. [16] proposed an online pointer analysis for monitoring the run-time behaviors of dynamic language features. Bodden et al. [17] introduced TAMIFLEX compared against in this paper.

The main challenge faced by dynamic reflection analysis is how to balance code coverage and analysis overhead. MIRROR tackles this challenge by focusing on the program paths exercising different targets at a reflective call and dynamically exercising these paths with automatically generated test cases.

c) **Static and Dynamic Reflection Analysis:** There are some prior studies for Android apps [46], [47]. STADYNA [47] interleaves static and dynamic analysis to reveal the program behaviors of dynamic class loading and reflection. However, this requires a modified Android virtual machine to log the side-effects of program behaviors (e.g., the reflective targets accessed) at runtime and involves human efforts (e.g., in preparing for test inputs), but with no guarantee for code coverage. HARVESTER [46] is designed to extract runtime values from Android apps, by executing a backward slice of an app on a stock Android emulator or real phone to log the values of interest, such as some class and method names.

MIRROR differs from STADYN and HARVESTER in several ways. First, MIRROR is the first for combining static and dynamic analysis in handling large Java applications. Second, MIRROR is fully automatically but STADYN is not. Third, MIRROR relies a novel reflection-oriented slicing technique to select and execute path-based slices in order to scale MIRROR for large object-oriented programs. In contrast, HARVESTER relies on traditional slicing to handle relatively small Android apps except that some environmental variables are modeled for the purposes of reducing slice sizes.

d) **Program Slicing:** Traditional slicing [23], [24] does not scale to large object-oriented programs [25], with the key bottleneck coming from handling of the heap [26]. Thus, existing techniques [31]–[33], [48] obtain their points of interest differently according to different goals. Path slicing [49] takes as input a program path leading to a target statement and eliminates all the statements that are irrelevant towards the reachability of the target statement. Thin slicing [26] improves scalability for object-oriented programs by ignoring control dependences and base pointer data dependences. Li et al. [28] introduced program tailoring to focus on the statements that pass through a sequence of API calls, including also the ones with no data or control dependences on the sequence. MIRROR differs from these by finding path-based slices confined to a set of du-chains in order to resolve reflective targets effectively.

VI. CONCLUSION

MIRROR is the first automatic Java reflection analysis that combines static and dynamic analysis to assist pure dynamic analysis tools to discover more reflective targets precisely and efficiently in large real-world Java applications.

Reflection analysis is challenging but significant. Currently, software industries lack adequate tools for handling reflection-heavy applications, causing many call-graph edges to be invisible to a variety of analysis tools. MIRROR represents a step forward towards developing effective reflection analysis tools.

MIRROR can be extended in several directions. First, a better call-graph reduction algorithm can be developed to enable more reflective calls to be sliced efficiently for large codebases. Second, its constraint solver can be improved to handle some file operations on some unknown file names. Third, some design patterns on reflection usage can be exploited to improve its efficiency and effectiveness further. Finally, some Android-specific analysis techniques (e.g., for callbacks and intents) can be added in order to handle Android apps.

ACKNOWLEDGEMENT

The authors wish to thank all the reviewers for their constructive comments. This research has been supported by an Australian Council Research Grant, DP170103956.

REFERENCES

[1] *About Java*, https://blogs.oracle.com/oracleuniversity/entry/why_is_java_the_most.
[2] *2013 cisco annual security report*, 2013, https://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2013_ASR.pdf.
[3] *2014 cisco annual security report*, 2014, http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf.
[4] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, "An in-depth study of more than ten years of Java exploitation," CCS, 2016, pp. 779–790.
[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[6] Y. Li, T. Tan, Y. Sui, and J. Xue, "Self-inferencing reflection resolution for Java," ECOOP, 2014, pp. 27–53.
[7] Y. Li, T. Tan, and J. Xue, "Effective soundness-guided reflection analysis," SAS, 2015, pp. 162–180.
[8] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1682–1699, 2016.
[9] D. Ye, Y. Su, Y. Sui, and J. Xue, "WPBOUND: Enforcing spatial memory safety efficiently at runtime with weakest preconditions," ISSRE, 2014, pp. 88–99.
[10] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for Java," APLAS, 2005, pp. 139–160.
[11] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer, "More sound static handling of Java reflection," APLAS, 2015, pp. 485–503.
[12] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection: Literature review and empirical study," ICSE, 2017, pp. 507–518.
[13] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.
[14] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," SAS, 2016, pp. 489–510.
[15] ——, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," PLDI, 2017, pp. 278–291.
[16] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 2, p. 11, 2007.
[17] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," ICSE, 2011, pp. 241–250.
[18] M. Eslamimehr and J. Palsberg, "Race directed scheduling of concurrent programs," PPoPP, 2014, pp. 301–314.
[19] ——, "Sherlock: Scalable deadlock detection for concurrent programs," FSE 2014, 2014, pp. 353–365.
[20] R. Purandare, M. B. Dwyer, and S. Elbaum, "Optimizing monitoring of finite state properties through monitor compaction," ISSTA, 2013, pp. 280–290.
[21] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," CODASPY, 2013, pp. 209–220.
[22] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," ICSE, 2017, pp. 300–311.
[23] M. Weiser, "Program slicing," ICSE, 1981, pp. 439–449.
[24] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.
[25] WALA, *T.J. Watson libraries for analysis*, http://wala.sf.net.
[26] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," PLDI, 2007, pp. 112–122.
[27] "Soot - a Java bytecode optimization framework," CASCON, 1999.
[28] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program tailoring: Slicing by sequential criteria," ECOOP, 2016, pp. 15:1–15:27.
[29] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.
[30] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Aliasing in object-oriented programming," 2013, ch. Alias Analysis for Object-oriented Programs, pp. 196–232.
[31] D. Binkley and M. Harman, "A survey of empirical results on program slicing," *Advances in Computers*, vol. 62, pp. 105 – 178, 2004.
[32] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.
[33] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys*, vol. 44, no. 3, pp. 12:1–12:41, 2012.
[34] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," CCS, 2006, pp. 322–335.
[35] K. Sen, G. Necula, L. Gong, and W. Choi, "MultiSE: Multi-path symbolic execution using value summaries," ESEC/FSE, 2015, pp. 842–853.
[36] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.
[37] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," ASE, 2010, pp. 179–180.
[38] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," ISSTA, 2011, pp. 34–44.
[39] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An interpolating SMT solver," SPIN, 2012, pp. 248–254.
[40] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT solver," TACAS, 2010, pp. 150–153.
[41] L. Li, Y. Lu, and J. Xue, "Dynamic symbolic execution for polymorphism," in *CC*, 2017, pp. 120–130.
[42] DOOP, *A sophisticated framework for Java pointer analysis*, http://doop.program-analysis.org.
[43] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents (T)," ASE, 2015, pp. 669–679.
[44] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," ISSTA, 2016, pp. 318–329.
[45] Y. Zhang, T. Tan, Y. Li, and J. Xue, "Ripple: Reflection analysis for android apps in incomplete information environments," CODASPY, 2017, pp. 281–288.
[46] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," NDSS, 2016.
[47] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: addressing the problem of dynamic code updates in the security analysis of Android applications," CODASPY, 2015, pp. 37–48.
[48] F. Tip, "A survey of program slicing techniques." Tech. Rep., 1994.
[49] R. Jhala and R. Majumdar, "Path slicing," PLDI, 2005, pp. 38–47.