IDEs-Friendly Interprocedural Analyser

Ilya Nozhkin
Saint Petersburg State University
St. Petersburg, Russia
nozhkin.ii@gmail.com

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
semen.grigorev@jetbrains.com

ABSTRACT

TODO: ABSTRACT

ACM Reference Format:
Ilya Nozhkin and Semyon Grigorev. 2019. IDEs-Friendly Interprocedural Analyser. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analyses are important part of modern development tools. They take care of verifying correctness of some program's behaviour freeing a programmer from this duty. By used scope of program, an analyze can be classified as intraprocedural or interprocedural, i.e. as those which make decisions based on only one current procedure or based on the whole program respectively. And interprocedural analyses, in theory, can be more precise due to amount of available information.

```
class A {
  [Source]
  int Source;
}

class B {
  [Filter]
  static int Filter(int d);
}

class C {
  [Sink]
  void Sink(int d);
}
```

```
class D {
  void Process(A a) {
    int d = Read(a);
    int f = B.Filter(d);
    Consume(d);
    AnotherConsume(f);
  }

  int Read(A a) {
    return a.Source;
  }

  void Consume(int d) {
    C c = new C();
    c.Sink(d);
  }

  void AnotherConsume(int d) { ... }
}
```

Figure 1: Sample code

For example, let's consider the listing in C#at fig. 1. It is known that method *Sink* is vulnerable to invalid arguments and so, it is marked by appropriate attribute. The method *Filter* validates its argument and possibly modifies it to ensure that returned data is definitely valid. Methods with such behaviour are marked with attribute *Filter*. The most common example of such method is the one that adds escape characters into the string to avoid affecting internal behaviour of the system by user input. And the field *Source* is known as potentially tainted that is also indicated by attribute. Class D extracts data using class A as a source, then validates it using class B then performs some computations involving class C. So, there might appear an issue that leads to usage of data that are not been validated yet as it happens during the invocation of method *Consume*. The problem is to find all such issues, i.e. we want an analysis that finds all possible ways how data from source can reach sink bypassing filters.

This problem is a special case of label-flow analysis, so there are several approaches of solving such problems. One of them is CFL-reachability. (TODO: ADV: performance, DISADV: expressive power, non-obvious structure, CITATIONS) Another is abstract interpretation. (TODO: vice versa) We propose to combine these two approaches to achieve acceptable performance and expressive power. I.e. the program is translated into a graph as it is in CFL-r, but constraints that specifiy what paths is needed to be accepted are set by pushdown automaton which transition relation simulates the semantics of original program. Let's take a closer look at these two components that define an analysis in conjunction.

2 ANALYSIS DEFINITION

2.1 Graph extraction

The graph that is explored during analysis is an aggregate of control-flow graphs of each method. The one that corresponds to our example is shown at fig. 2.

Each edge contains an operation that represents a statement in the source code and in the same time the target of the edge indicates where to jump after execution of the operation. For example, there exist three different types of operations: invocations, assignments and returns. Each of them is an image of some source instruction. Invocations are produced from call sites and have the same information as ones in original code. I.e. it contains a reference to the entity which method is called, the name of target, a set of arguments and the information about returned values. Each assignment corresponds to real assignment and has references to the source entity and the target one. And return just indicates the end of a method. It can be not present explicitly in the source code but is still needed to be added to inform analyser about return point. However, the number of instruction types is not fixed and some analysis-specific

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

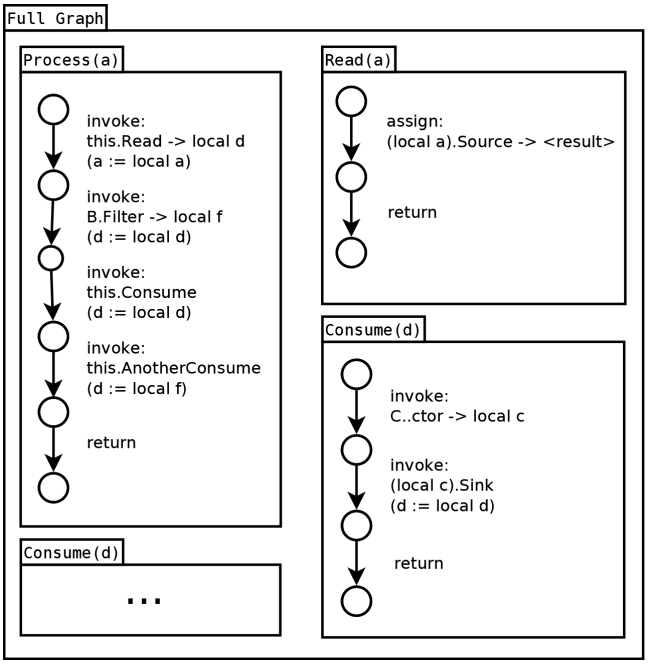

Figure 2: Sample graph

operations can be added if necessary. Nodes have no any data and correspond to positions between instructions in the source code.

So, we have a bunch of graphs each of which represents the content of one method. Next step is to interconnect them to have an opportunity to perform interprocedural jumps during invocations. There are several possible way to do that. First of them is to expand invocations statically, i.e. add a pair of edges for each target of each invocation. One to represent a jump from the call site to the entry point of target and one to emulate return from the final node of the method to the caller. But this approach leads to the need to update all these additional connections if some method is removed or its body is changed. So, we propose to resolve invocations dynamically right during an analysis. It allows us not to modify initial graphs structure at all. Nevertheless, it is still needed to have a mechanism that can perform the resolution, i.e. there must be an entity that can collect all targets of any invocation using references stored there.

To implement such mechanism, called resolver, we offer to accumulate some meta-information about the program besides graphs themselves. The relations in required data is shown at fig. 3.

Firstly, it is important to keep the hierarchy of inheritance to support polymorphic calls and invocations of methods of a basic class. Secondly, it is needed to know which methods are contained in each class to find the method by its name and its location. Thirdly, methods can have local functions and it is necessary to keep their hierarchy too to support, for example, anonymous function invocations, delegates passing and so on. And finally, methods themselves has references

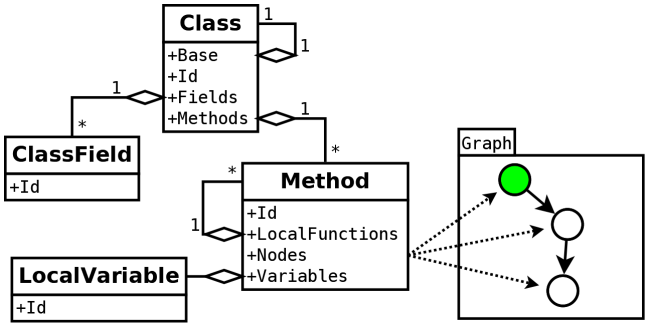

Figure 3: Metadata

to nodes they own which is used to find the entry point and update the graph when the body of method is changed. This structure also contains such data as class fields and local variables of a method which can be referenced by operations. So, the resolver takes the class name and the identifier of a method or a field and walks through the hierarchy tying to find all suitable entities.

2.2 PDA construction

Formally, nondeterministic pushdown automaton is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where Q, Σ and Γ are finite sets of states, input symbols and stack symbols respectively, $q_0 \in Q$ and $Z_0 \in \Gamma$ are initial state and stack symbol, $F \subseteq Q$ is a set of final states and $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is a transition relation which computes new state and stack by current ones and input symbol. We also add following restriction on transition relation. The resulting stack must differ from the source one by no more than one top symbol. I.e. only one symbol can be pushed or popped during the transition.

Next, we propose to take the set of all edges in the control-flow graph as Σ. So, the transition relation can be understanded as a structural operational semantics that defines how configuration is changed during execution of a statement. All other sets can be taken arbitrary.

However, there is one more problem. Semantics of invocation contains the need to make a jump from the current position to the entry point of the target instead of just going to the node that is pointed by the current edge. To support such behaviour we propose to change the codomain of δ such that $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to \mathcal{P}(Q \times \Gamma^* \times N \cup \{\nu\})$, where N is the set of nodes of the graph and ν is the dummy value that means that there is no need to jump and PDA just goes to the next node.

For example, let's construct the PDA performing that taint tracking analysis described in the introduction. Let $Q := V \cup \{q_0, q_f\}$, where V is set of all local variables of all methods and q_0 and q_f are dummy initial and final state, so $F := q_f$. $\Gamma := I \cup \{Z_0\}$, where $I \subset \Sigma$ is set of all edges containing an invocation and Z_0 is dummy initial stack symbol. And δ is

defined by the case analysis (1).

$$
\begin{aligned}
&\delta(q_0, i@\text{invocation},\gamma) \coloneqq \\
&\qquad \{(q_0, i :: \gamma, s_0), \ldots, (q_0, i :: \gamma, s_n) : \\
&\qquad\ s_0, \ldots, s_n \in R(i)\} \\
&\delta(q_0, a@\text{assignment}(v_s, v_t), \gamma) \coloneqq \\
&\qquad \begin{cases} \{(v_t, \gamma, \nu)\}, & \text{if } \mathit{source}(v_s) \\ \{(q_0, \gamma, \nu)\}, & \text{otherwise} \end{cases} \\
&\delta(v, a@\text{assignment}(v, v_t), \gamma) \coloneqq (v_t, \gamma, \nu) \\
&\delta(v, i@\text{invocation},\gamma) \coloneqq \\
&\qquad \bigcup_{j=0}^{n} \{(v_{j0}, i :: \gamma, s_j), \ldots, (v_{jm}, i :: \gamma, s_j) : \\
&\qquad\ v_{jk} \in A(i, j, v)\}, s_j \in R(i) \\
&\delta(v, r@\mathit{return}, i :: \gamma) \coloneqq \\
&\qquad \begin{cases} \{(RV(i), \gamma, T(i))\}, & \text{if } \mathit{returned}(v) \\ \{\}, & \text{otherwise} \end{cases} \\
&\delta(q, \_, \gamma) \coloneqq \{(q, \gamma, \nu)\}
\end{aligned}
\tag{1}
$$

Where *source* checks if a variable is a source, *returned* checks if current variable is a return value of some method, T returns target node of an edge, RV returns the variable where the result of an invocation must be put, R is the resolver returning entry points of all possible targets of an invocation and A is defined by equation (2).

$$
A(i, j, v) \coloneqq \begin{cases} \{q_f\}, & \text{if } j\text{-th target of invocation } i \\ & \text{is sink and } v \text{ is its argument} \\ \{v_k : v \mapsto v_k\}, & \text{if } j\text{-th target of i} \\ & \text{is not filter} \\ \{\}, & \text{otherwise} \end{cases} \tag{2}
$$

Where $v \mapsto v_k$ means that v is passed as k-th parameter and becomes local variable v_k of the target after passing.

3 SOLUTION

The solution is an extensible infrastructure which is responsible for extracting graphs from the source code, aggregating them and their metadata into one database and finding paths in this database accepted by PDAs representing different analyses. Logically, it is divided into two separate entities which are shown on fig. 4.

The first entity, the core of the solution, is a backend implemented as a remote service running in a separate process and interacting with the frontend using a socket-based protocol. Architecturally, it is also divided into two subsystems. First of them is a database which provides the continuous incremental updating of the graph and its metadata, and supports dumping to a disk and further restoration in the beggining of next session. The second is responsible for execution of analyses. It contains an implementation of the resolver, the algorithm of PDAs running and the first extension point making the adding new analyses possible. The set of analyses contained in the backend can be extended by adding a new PDA as just a new implementation of appropriate generic abstract class. Furhter, it is possible to run this new anaysis

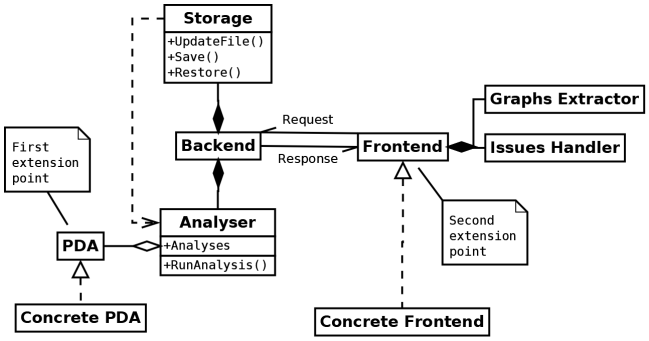

Figure 4: Solution structure

using existing internal algorithm of PDA simulation and get any finite subset of paths in the graph which are accepted by the PDA.

Second main entity is a frontend that is also divided into two subsystems. First of them is a graph extractor which parses source code, extracts graphs and metadata from it and sends collected data to the backend. The second is a results interpreter that receives the set of paths in the graph leading to an error, maps it into the source code and does something with this information. For example, it can highlight pieces of code which participate in the error producing. Also, the frontend is the second extension point because the backend, in general, can interact with any implementation supporting the protocol.

The protocol itself is based on request-response behaviour where the frontend acts as a master and the backend acts as a slave. I.e. the frontend informs the backend if there are some changes in the source code and asks it to update the database according to them. And when there is a need to get the results of some analysis, for example, when the IDE performs the code highliglighting, the frontend asks the backend for found issues and waits until it responses.

4 EVALUATION

In order to test the resulting solution we have implemented the frontend as a plugin using ReSharper SDK, so it can be installed into ReSharper, Rider and InspectCode. The source code is parsed by internal ReSharper tools and the result is used to produce graphs and meta-information. The issues found by the backend are shown using code highlighting.

Also, we have implemented the PDA constructed in the section 2 with some slight modifications allowing to process interactions with object fields. To provide more information an issue found by this analysis, the code higlighting accompanied by bulbs containing the full path of tainted variable from the source to the sink represented as the sequence of operations.

Let's look closer at properties of the resulting soluiton. All these properties are illustrated by screenshots taken exactly from the runned Rider IDE with some small relocations of bulbs to make them not to overlap the code.

Firstly, the solution ensures flow sensitivity. I.e. it processes flow of variables passed into methods and returned from them correctly. Which can be seen at fig 5.

```
17  class Program
18  {
19      [Tainted] private int A;
20      [Filter] private int Filter(int a) { return a; }
21      [Sink] private void Sink(int a) {}
22
23      private int PostSource() {
24          var b = A;
25          return b;
26      }
27
28      private int Brackets(int a) {
29          var b = a;
30          return b;
31      }
32
33      static void Main(string[] args) {
34          Program a = new Program();
35          var c = a.PostSource();
36          var d = a.Filter(c);
37          var e = a.Brackets(c);
38          var f = a.Brackets(d);
39          a.Sink(e);
40          a.Sink(f);
41      }
42  }
```

Tainted sink
source - Program.cs:24
assign - Program.cs:25
return <-
assign - Program.cs:35
pass -> Program.cs:37 (System.Int32)Brackets(System.Int32)
assign - Program.cs:29
assign - Program.cs:30
return <-
assign - Program.cs:37
sink -> Program.cs:39 (System.Void)Sink(System.Int32)

Figure 5: Snippet 1

In the example, line 39 contains the sink reachable from the tainted source, so it is highlighted and there also shown a full trace of instructions over which the tainted variable is passed. However, the sink on line 40 is not highlighted because data reaching it are passed through filter, despite it passes through method *Brackets* which also passed by a tainted variable.

Secondly, the solution has a limited context sensitivity. I.e. it allows to track propagation of objects that are tainted by assigning of some fields inside them both by their own methods and by outer code interacting with their fields directly. The first case is shown at fig 6.

There is a method *Store* which execution affects only that object by which reference the method is called.

Finally, the solution works with any type of recursion and does not fall into infinite cycles. It can be seen at fig. 7.

TODO: PERFORMANCE

5 CONCLUSION

TODO: CONCLUSION

```
17  class Container {
18      private int B;
19      public void Store(int a) { B = a; }
20  }
21
22  class Program {
23      [Tainted] private int A;
24      [Filter] private int Filter(int a) { return a; }
25      [Sink] private void Sink(Container c) {}
26
27      static void Main(string[] args) {
28          Program a = new Program();
29          var b = a.A;
30          var c = a.Filter(b);
31          var d = new Container();
32          var e = new Container();
33          d.Store(b);
34          e.Store(c);
35          a.Sink(d);
36          a.Sink(e);
37      }
38  }
```

Tainted sink
source - Program.cs:29
pass -> Program.cs:33 (System.Void)Store(System.Int32)
assign - Program.cs:19
return <-
sink -> Program.cs:35 (System.Void)Sink(TaintTrackingTests.Container)

Figure 6: Snippet 2

```
17  class Program
18  {
19      [Tainted]  public int A;
20      [Filter] private int Filter(int a) { return a; }
21      [Sink] private void Sink(int a) {}
22
23      private int Recursive1(int c, int d) {
24          var r = Recursive2( c: c - 1, d);
25          return r;
26      }
27
28      private int Recursive2(int c, int d) {
29          if (c == 0) return d;
30          var r = Recursive1(c, d);
31          return r;
32      }
33
34      static void Main(string[] args) {
35          Program a = new Program();
36          var b = a.A;
37          var c = a.Filter(b);
38          var d = a.Recursive1( c: 10,  d: b);
39          var e = a.Recursive1( c: 10,  d: c);
40          a.Sink(d);
41          a.Sink(e);
42      }
```

Tainted sink
source - Program.cs:36
pass -> Program.cs:38 (System.Int32)Recursive1(System.Int32,System.Int32)
pass -> Program.cs:24 (System.Int32)Recursive2(System.Int32,System.Int32)
assign - Program.cs:29
return <-
assign - Program.cs:24
assign - Program.cs:25
return <-
assign - Program.cs:38
sink -> Program.cs:40 (System.Void)Sink(System.Int32)

Figure 7: Snippet 3