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, 2 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] 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 directly in the source code but is needed to be added to

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

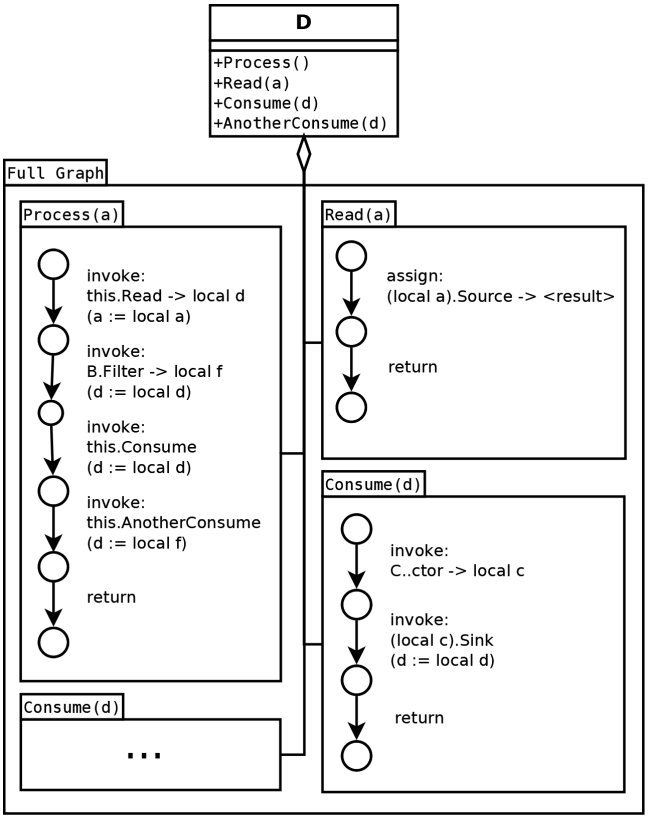

Figure 2: Sample graph

2.2 PDA construction

3 SOLUTION

4 EVALUATION

TODO: EVALUATION

5 CONCLUSION

TODO: CONCLUSION