

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]
    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, DIS-ADV: 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 specify 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2019 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

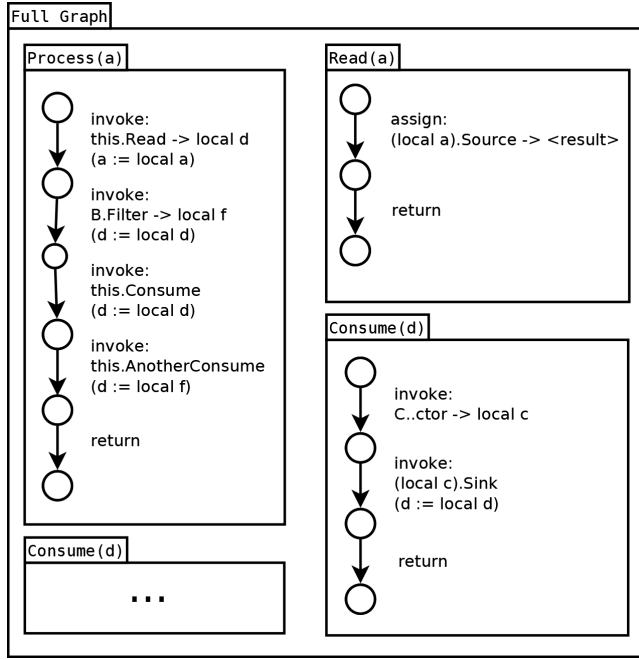


Figure 2: Sample graph

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

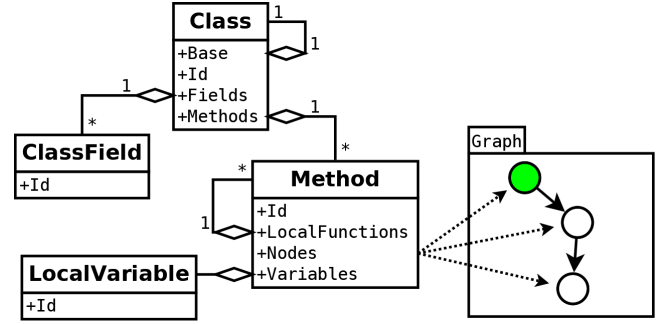


Figure 3: Metadata

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*

3 SOLUTION

4 EVALUATION

TODO: EVALUATION

5 CONCLUSION

TODO: CONCLUSION