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, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

TODO: INTRODUCTION

2 PRELIMINARIES

TODO: PRELIMINARIES

3 SOLUTION

3.1 Main idea

The solution involves using of the conception that is close to CFL-reachability to solve the problems mentioned above.

The classic CFL-r approach (TODO-CITATION) is a search of paths in a graph, edges concatenation of which is a word of a certain context-free language which is defined by grammar. In case of static analyses it means the following specialization. The analysed graph is the control-flow graph of the considered program such that its nodes represent states and edges contain statements which transfer program from one state to another respectively. The grammar, in turn, defines sequences of statements passing through which leads to an error. So, the analysis is a composition of a grammar and a set of rules that define a translation of existing program into control-flow graph. And the result of the analysis are a control-flow graph and a set of paths in it each of which corresponds to a sequence of operations that can be passed through during the execution of the original program.

However, such definition has a few drawbacks when it comes to static analyses. The first of them is that grammars have a slightly unnatural structure in comparison with program interpreter. For example, call-return edges pairing is defined using grammars as brackets that contain anything else between them. In interpreter semantics, in contrast, calls and returns are usually defined separately from each other using only stack concept. So, the main idea is to formulate analyses in terms of pushdown automata which has the same computational power as context-free grammars (TODO-CITATION) but can emulate original interpreter semantics in more natural way. Moreover, it is quite useful to define automata as abstract as possible to allow to use any objects as states, input and stack symbols and by this get closer to interpreter in contrast to grammars which are based on strings. NEED-HELP: GRAMMARS, IN GENERAL, ARE NOT LIMITED BY STRINGS AND THEIR REAL DISADVANTAGE IS THAT THEY ARE BASED ON EXACT MATCHES OF TERMINALS, BUT HOW TO EXPRESS IT MORE CLEARLY?

Therefore we define an analysis as a composition of PDA and the set of rules that generates a control-flow graph. The result of such analysis is still a set of paths in the graph each of which can be traversed during the usual execution and also accepted by automaton, thus it can cause an erroneous behaviour.

3.2 Example

Now, let's consider the construction of a simple analysis including definition of graph and PDA. Let the sample problem be a kind of taint tracking analysis, namely, we propagate tainted variables from marked sources to the vulnerable sinks simultaneously checking whether they pass through filters or not.

Firstly, we construct graphs of each method so that each edge represents one statement of original program. Next step is to provide a way of interaction between methods somehow.

TODO: PROGRAM AND PICTURE

One way of providing interprocedural connections is to replace invocations with calling and returning edges that are directed to the beginning or from the end of an invoked procedure respectively and then add different labels for each pair to distinguish one pair from another. Of course, such approach allows to perform further analysis but has some disadvantages. First of them is that when some procedure changes then all connections that have been added instead of invocations inside it should be removed and then new edges need to be added again. But the second one is more significant. It is not obvious how to support dynamically forming connections such as invocations of delegates.

Thus, instead of it, we offer to modify PDA concept and add an ability to jump to any point of input graph during the transition. Actually, it can be interpreted as adding of fake edges. So, there is no need to change produced control

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

flow graph at all and the only requirement is to have an opportunity to find the entry point of an invoked procedure during the simulation.

Now let's define PDA that performs the considered analysis. To simplify definition we give it in informal way which still can be translated into strict rules. Let set of states be set of variables with one dummy initial state, set of stack symbols be just edges with one dummy edge that indicates the bottom of the stack and the transition rules be following (SHOULD I REWRITE THEM IN PSEUDOCODE???):

- If current state is initial and next operation is assignment of some variable to tainted source then change state to this variable
- If current state is initial and operation is an invocation then push current edge to the stack and jump to the entry point of called procedure.
- If current state is a variable and operation assigns some variable to the current variable then change state to this new variable
- If current state is a variable and operation is invocation that passes this variable as argument then push current edge to the stack, switch state to the variable that corresponds to the argument and jump to the entry point of called procedure.
- If current state is a variable and operation is return of this variable from function then pop the edge from the stack, change state to the variable that is assigned by popped invocation and jump to the target of the popped edge.
- If current state is a variable and operation is a filter invocation then just drop further execution because since this point variable is already filtered.
- And finally, if there met some sink then accept the path.
- In all other cases just skip the operation

Therefore, if this PDA is runned from the whole programs's entry point then each accepted path is the sequence of operations that since the certain one pass some tainted variable from a source to a sink bypassing any filters.

So, since there is a definition of the analysis it is needed to have an engine that makes it possible to implement this rules using it and then get result of the computation. The solution described below meets exactly these requirements.

3.3 Solution structure

In order to provide the most common interface for interaction with IDE, the solution is offered to be divided into two separate entities. The first of them is the thin plugin for IDE that translates methods into the graphs, sends them to the second entity and then gets analysis results by request. The second one is the remote service that aggregates graphs into the full graph of the program, is able to update it incrementally and finally can perform any available analysis by request. This side also takes care about PDA simulation and results extraction and provides interfaces that require only the PDA implementation itself. Now, let's consider how to implement the analysis defined above using the presented entities.

3.4 Data representation

The first thing that is needed to be described is a representation of data extracted from the source code because it is used throughout the whole path of the analysis development. Let's start with the highest entities in the hierarchy which are present in fig 1

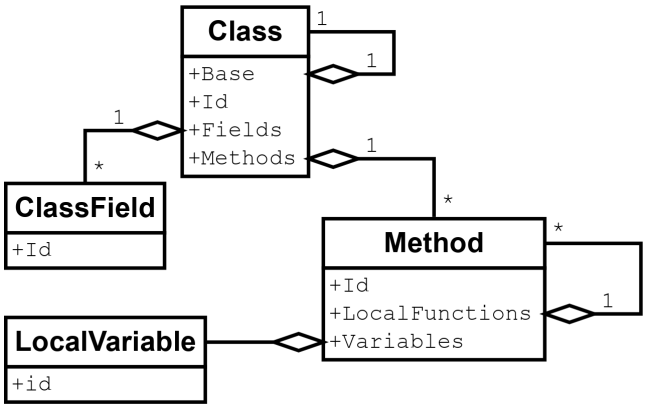

Figure 1: Top Level Entities Hierarchy

Classes, their fields and methods follow the structure of the original program and provide methods that makes it possible to find any entity by its id and location. Inheritance is supported because any class has a reference to the basic one that allows to determine concrete implementation of any method. Methods, in turn, contain local functions inside and local variables which also can be referenced during analysis.

Further, let's consider how the body of a method is defined (fig. **??**).

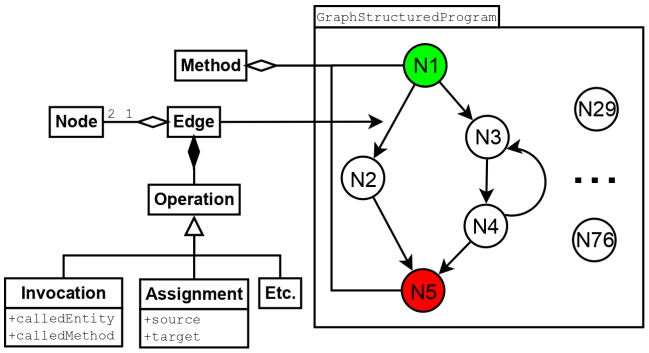

Figure 2: Method body

As it is said before, the body of a method is a part of the whole programs's CFG with the single entry point. Each edge represents one statement of the original program that is also translated into some instance of abstract statement. The basic set of statements contains the very special one

called Invocation that is the central entity of the whole idea of interprocedural analysis. Since we proposed not to replace the edge containing the invocation with the pair of edges that leads to entry and final points of a target respectively it is necessary to define another way of performing a jump. The offered solution is to give an opportunity to find the target of invocation right during the analysis that is implemented using the system of identifiers representing classes and methods. I.e. each invocation has the reference to the entity which method is being invoked and the identifier of the invoked method itself which could be used to find the entry point. Of course, there is a problem of deciding what is the real instance of the entity which method is being invoked. For example, the calling of a method of an object by some interface, could potentially lead to the invocation of any implementation of that interface. That is why the service also provides the analysis that collects any possible type of any variable in the program to solve this problem and then allows to find the set of targets of any invocation. (NEED-HELP: SHOULD I TELL HERE ABOUT ENTITIES SYSTEM???) So, since we define a data representation the next step is to define an automaton that gives a semantics to the each possible statement and by this perform the interpretation of the program.

3.5 Automata construction

The main purpose of the automaton is the interpretation of statements, i.e. if an automaton meets an invocation it should compute the target and perform a jump to the entry point. So, we need an abstraction that has the same computational power as PDA but is able to use any additional information while performing a transition. The proposed way of definition that meets these requirements is to implement an analyser just as inststance of the specific abstract class that provides methods which are responsible for PDA-like behaviour. This abstraction is called pushdown virtual machine (PDVM) (fig. 3).

Pushdown Virtual Machine

```
#Push(state,symbol,jump=<next position>)
#Pop(state,jump=<next position>)
#Skip(state,jump=<next position>)
#Save(state,jump=<next position>)
#Accept()
#Step(state,stack,input,position)
#Action(state,stack)
```

Figure 3: Abstract Pushdown Virtual Machine

It is designed to provide a set of specific methods that control PDA-like behaviour. The first important subset of them are Push, Pop, Skip and Save. Each of them corresponds to one transition of classic PDA but supports jumps as well. I.e they perform corresponding stack modification, change the state and then jump somewhere or just step to the next position if jump target is not specified. The only odd thing here is difference between Skip and Save. Both of them just stay stack unchanged and then performs other actions, however skipped input symbols are not added into traces during further results extraction and saved ones are.

Another contol method is Accept. Invocation of it leads to the accepting of all paths execution of which finishes in the current position and in the current state.

Finally, two remaining methods are needed to be implemented by developer using control methods. Step method is invoked for each next input symbol in a configuration. Action method is called in each new configuration just before input symbols reading. It can be used when it is needed to perform a chain of transitions without any movement.

So, using the PDVM and the information about representation of program we can implement our sample analysis.

TODO:EXAMPLE

However, a PDVM itself is not an analyzer yet, because it is still need to be executed by some algorithm.

3.6 Analysis execution

To transform an analysing PDVM into the complete analsysis it is necessary to have an opportunity to simulate it on the input graph and then collect results as accepted paths. So, service provides a wrapping class that takes the responsibility to solve these problems. It takes a PDVM and produces a method that returns the set of paths accepted by PDVM runned from given initial positions in the graph. Of course, due to possible acceptance of cycles, the set of accepted paths, in general, can be infinte, so, to get around this problem, the returned set of paths contains only paths which has not more than one expansion of any cycle. However, there is a way to get paths containing as many expansions as necessary. The simulation itself is performed by (TODO: SOME SHORT DESCRIPTION).

Finally, the analyser constructed this way can be integrated into the considered solution and then can be used to analyze the source code of any program that can be represented in the appropriate form. Further, let's consider the one implementation of plugin that was developed in pair with described solution and can interact with it to highlight discovered errors in real time.

4 EVALUATION

TODO: EVALUATION

5 CONCLUSION

TODO: CONCLUSION