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.

3.4 Service

Service is implemented in C# but due to socket-based connection can be used with any other plugin implementation that supports interchange protocol.

The protocol is based on request-response pattern where plugin acts as a master and sends requests. The smallest set of requests that make performing of the full analysis cycle (TODO: WHAT CYCLE) possible is present in table. 1. Note that updating is performed by file but not by class because of partial classes that can be not seen fully in IDE.

Request	Sent or received data
Update File	Sent: The whole information about updated file that is represented as hierarchical structure that contains classes which in turn contain fields and methods each of which is a CFG and some additional information such as attributes, local functions and variables.
Perform Analysis	Sent: The type of analysis that is needed to be runned
Get Analysis Results	Received: The set of paths that are returned by analyser

Table 1: Requests

Now, let's consider the journey of data after receiving of a file updating request. The main problem is that data received from plugin have a bit easier but less informative structure in case of interprocedural analyses. I.e. generated method graphs have only such information that can be extracted directly from the source code of one method. For example, all invocations has only symbolic information about target such as namespace, name and signature but analyser needs to know at least how to find the entry point of graph corresponding to the target of the invocation. So, before data are inserted into the full graph of the program they are processed by the chain of entities called resolvers. Each of them finds some global entity corresponding to a certain piece of data, adds it back and passes the whole result to the next resolver. After the source data are passed through the full chain of resolvers it is ensured that any entitiy such as class, field, method or even lambda that is referenced somewhere inside the method can be found during the execution of an analyzer. The result of resolving that represents a method with all necessary metadata is added into the service's database.

In case of example presented above it means that (TODO: EXAMPLE).

So, aggregated graphs and their metadata form the database that is always present in memory, updates after each request and follows the structure of the original program but makes it possible to represesnt analyses in terms of reachability problems. It is also important that the active database has some redudant information that is important for analyses and updating but can be omitted on serialization. That is why the database cannot be partially dumped into the disk and

stays in the memory during the whole session. Furthermore, the process of loading of the database from a disk has a stage that is similar to the chain of resolvers and it is aimed to restoring of all additional information.

Since the database is constructed the next step is to identify how PDA-based analyses are implemented using considered solution.

3.5 Automata construction

TODO: SHORT INTRODUCTION. MORE PROPER STYLE. The main entity that is used for the construction is the abstract PDA class that is needed to be inherited by concrete analysis implementation. Due to generalization of PDA and its slightly different meaning in case of static analyses it is named pushdown virtual machine (PDVM) (fig. 1).

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 1: 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.

TODO:EXAMPLE

Since a PDVM is constructed it can be runned from any set of positions. This opportunity is provided by a separate generalized subsystem that takes any representation of graph and any PDVM and performs computations.

3.6 Automata simulation

The very core of the whole solution is the alogrithm that performs the simulation of automata on a graph input. It is completely independent from other parts of the solution and is designed as abstract as possible to support any definition of graph and any additionaly computations during simulation. Such approach requires to implement a set of specific entities that provides all necessary information to the simulator. The whole infrastructure is presented at fig. 2.

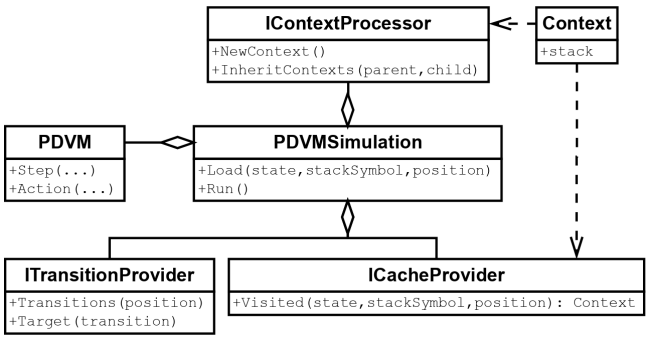

Figure 2: PDVM Simulating Infrastructure

Although the service part of the solution provides all interfaces that are present in the diagram, it is still necessary to consider how to implement them manually to give an opportunity to adapt the simulator to some unusual data representation. The first entity is a transitions provider that represent a graph structure itself. It has two methods: the one that returns a set of transitions (edges) outgoing from the given position (node) and the another that returns a target position of the given transition. Both of them are alternately called by the simulator to propagate the simulation flow through the graph.

The second entity is a context. It represents a state of simulator possibly with some additional user information. Note that the state of simulation and the state of a PDA are different things. This state is a tuple of PDA's state, top of the stack and the position in the input. And it leads us to the main idea of the simulating algorithm that is based on assumption that there is no need to process some context more that one time. So, the third entity is a cache provider that is used just to check whether the automaton has already been in the certain state with the certain top of the stack in the given position or not and if it has then returns the previously processed context of the simulation. Such approach makes it possible to use some intermediate results of computation to make next step that ensures polinomial time and space complexity and also provides the opportunity of handling of infinite cycles during simulation.

The simulator switches from one context to another on the each step and such process called an inheritance. It can produce some new context during the transition or just switch to an existing context. So the last entity is a context

processor that is responsible for creating of new contexts and propagating of some user information during the inheritance.

The implementation of all of these entities allows to simulate any PDVM that uses the corresponding types of states, transitions and stack symbols. The last step is to load a set of initial contexts using the appropriate method of a simulator and then run it.

However, it is also important to get the results of simulation. It can be implemented using context processor that builds some structure during the inheritance. The standard implementation provides one processor that allows to extract paths accepted by an automaton. To realize what it actually does it might be useful to understand an input graph as a finite state automaton. Then the simulation of PDVM performs the intersection of FSA and PDA that produces another PDA which states are just contexts of simulation and transitions are their switches. So, the standard context processor just builds such PDA. Further, extraction of any word that is accepted by this PDA gives a path in the original graph that is accepted by the original PDA.

TODO: WHAT'S ELSE???

3.7 Plugin

TODO: PLUGIN

4 EVALUATION

TODO: EVALUATION

5 CONCLUSION

TODO: CONCLUSION