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
semyon.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, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analysis is an 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 and 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 {
  [Tainted]
  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 classic taint tracking problem (e.g. described in [**?**] and [**?**]) which is very advisable to solve using interprocedural analysis. The core idea of the problem is that some input data can have inappropriate format or contain an exploit, such data are called *tainted*, and this data can reach some vulnerable operation that would lead to an incorrect behaviour. Let's introduce the key entities using the listing at fig. 1 in C#.

The first type of them is *source*. In our case *source* is some field that can potentially contain the tainted data, for example, let it be the field *Source* of the class A. The second type is *sink*. *Sink* is a method which is vulnerable to tainted data. In our example, the method *Sink* of the class C has this property. And the third important type of entities is *filter* (or *sanitizer* in some other definitions). It is a method that checks the correctness of data passing through it and if they are incorrect, *filter* throws an exception or modifies them to ensure the correctness of the result. Let the method *Filter* of the class B in the given snippet be a filter.

Let's assume that each entity is marked by a programmer with an appropriate attribute: *[Tainted]* for sources, *[Filter]* for filters and *[Sink]* for sinks. So, the problem stands for finding all paths being passed through which the tainted data can flow into a sink bypassing any filter.

This problem is a special case of interprocedural label flow analysis, so there are several approaches of solving such problems. One of them is CFL-reachability and the solution involving this approach is present by Rehof et al. in [2]. Moreover, CFL-reachability is a long-time studied framework and thus there are a lot of other possible applications to static analyses and also there developed algorithms which can reach acceptable performance in practice (TODO: CITE GLL?). The main idea of this approach is to find paths in a graph that satisfy constraints defined by context-free grammar. In particular, a path is accepted if the concatenation of labels on its edges gives a word which can be derived in the grammar. However, in practice, there are a few drawbacks of such definition. Firstly, grammar-driven parsing is based on exact matching of terminals which forces to generate a very large grammars in case when edges contain some unique attributes. For example, brackets matching described in [2] requires to generate as many rules as there are call sites in the source code. Secondly, ???

Moreover, there is another engineering problem that it is needed to extract a graph from a program to perform further computations. So, the main purpose of our work is to implement a tool solving all mentioned problems and thus allowing to use CFL-reachability in practical cases. In the further sections:

- We describe scalable representation of a graph which allows to contain as much information about the program as necessary

- We introduce another approach for definition of constraints based on pushdown automata instead of grammars which makes formulating of analyses easier
- We present the extensible solution which allows to implement new types of analysis using introduced abtractions and the plugin which uses the solution to provide analysis results to ReSharper, Rider and InspectCode (source code and executables can be downloaded here: github.com/gsvgit/CoFRA)
- We evaluate it by testing on a few synthetic tests and estimate performance by running an analysis on a large open-source project

2 ANALYSIS DEFINITION

To apply the CFL-reachability framework, the first thing to do is to define the representation of a graph and path restrictions. We propose the following division. Let's graph be image of a program and stores the information about the source code that can be mapped back to locate issues in the sources that are found by analysis of the graph. Then restrictions on paths define the sequences of operations leading to an issue. In order to keep the expressive power of the CFL-reachability approach but make the implementation easier, we propose to use pushdown automata instead of grammars which have equivalent power [1]. Now, let's take closer look at each component and consider the construction of them in application to our example.

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. Their notation has the following form:

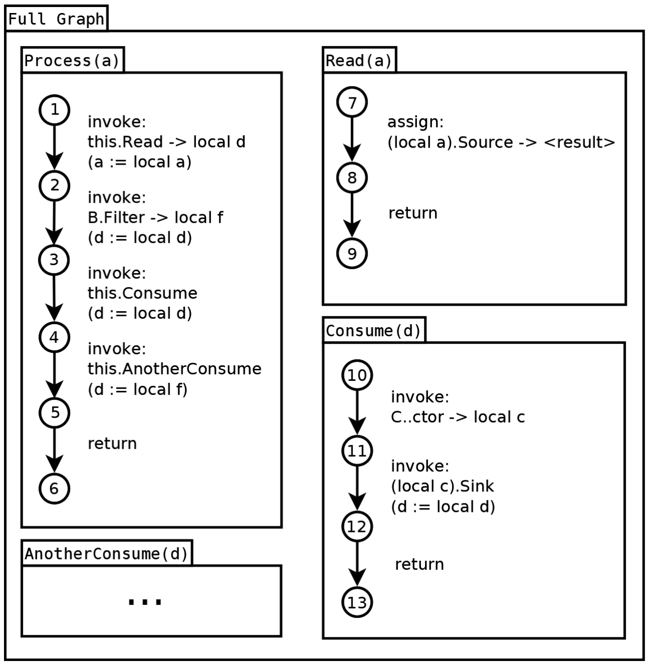
$$\begin{array}{l}\text{invoke:}\\ o.m \to v\\ (f_1 := c_1; \ldots ; f_k := c_k)\end{array} \qquad (1)$$

Where o is an object or a class which method is called, m is the name of a method, v is a variable where the result is stored and f_i, c_i are pairs of formal and actual parameter respectively.

Each assignment corresponds to real assignment and is written in the following way:

$$\begin{array}{l}\text{assign:}\\ s \to t\end{array} \qquad (2)$$

Where s is a source of data and t is the a target variable.



Full Graph
Process(a)
invoke: this.Read -> local d (a := local a)
invoke: B.Filter -> local f (d := local d)
invoke: this.Consume (d := local d)
invoke: this.AnotherConsume (d := local f)
return
Read(a)
assign: (local a).Source -> <result>
return
Consume(d)
invoke: C..ctor -> local c
invoke: (local c).Sink (d := local d)
return
AnotherConsume(d)
...

Figure 2: Sample graph

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 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 to use the graph which is composed right of graphs of methods and has no any additional edges. Also, it does not require to modify any other method when some one is updated. Nevertheless, it is still needed to have a mechanism 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

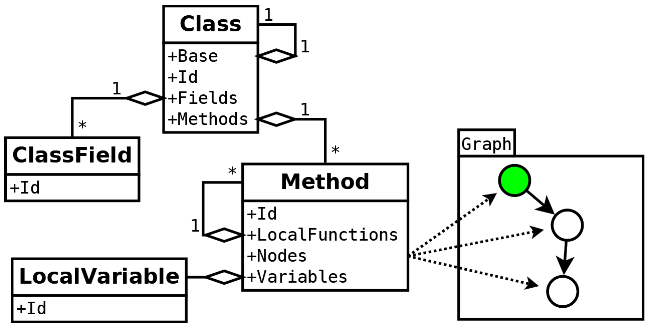


Figure 3: Metadata

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 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 trying to find all suitable entities.

2.2 PDA construction

Further, we need to define restrictions on paths in the graph in terms of pushdown automata. 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 (3).

$$\begin{aligned}
&1)\ \delta(q_0, i@\text{invocation}, \gamma) := \\
&\qquad \{(q_0, i :: \gamma, s_0), \ldots, (q_0, i :: \gamma, s_n), (q_0, \gamma, \nu) : \\
&\qquad\ s_0, \ldots, s_n \in R(i)\} \\
&2)\ \delta(q_0, a@\text{assignment}(v_s, v_t), \gamma) := \\
&\qquad \begin{cases} \{(v_t, \gamma, \nu), (q_0, \gamma, \nu)\}, & \text{if } \mathit{source}(v_s) \\ \{(q_0, \gamma, \nu)\}, & \text{otherwise} \end{cases} \\
&3)\ \delta(v, a@\text{assignment}(v, v_t), \gamma) := (v_t, \gamma, \nu) \\
&4)\ \delta(v, i@\text{invocation}, \gamma) := \\
&\qquad \bigcup_{j=0}^{n} \{(v_{j0}, i :: \gamma, s_j), \ldots, (v_{jm}, i :: \gamma, s_j), (v, \gamma, \nu) : \\
&\qquad\quad v_{jk} \in A(i, j, v)\}, s_j \in R(i) \\
&5)\ \delta(v, r@\mathit{return}, i :: \gamma) := \\
&\qquad \begin{cases} \{(RV(i), \gamma, T(i))\}, & \text{if } \mathit{returned(v)} \\ \emptyset, & \text{otherwise} \end{cases} \\
&6)\ \delta(q, \_, \gamma) := \{(q, \gamma, \nu)\}
\end{aligned} \tag{3}$$

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 (4).

$$A(i, j, v) := \begin{cases} \{q_f\}, & \begin{array}{l}\text{if } j\text{-th target of invocation } i \\ \text{is sink and } v \text{ is its argument}\end{array} \\ \{v_k : v \mapsto v_k\}, & \begin{array}{l}\text{if } j\text{-th target of i} \\ \text{is not filter}\end{array} \\ \emptyset, \text{otherwise} \end{cases} \tag{4}$$

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.

2.3 Analysis execution

Let's use the constructed automaton to find issues in the sample source code. The goal is to find the sequence of operations which starts in the entry point and ends at the invocation which uses the data from an unfiltered source. Since the automaton accepts such sequences, we can simulate the switching of its configurations according to the input statements taken from the source graph. Let's write down this process step by step using the following notation. $(q, \gamma_1 :: \ldots :: \gamma_k, n)$ is the current configuration of the simulation where q is the current state, γ_j is a symbol on the stack and n is the current position in the input graph. $c_1 \xrightarrow[k]{r} c_2$ is the k-th transition which switches configuration c_1 to c_2 using rule r from equation (3). Local variables are written as $\langle$Name of the containing method$\rangle$.$\langle$Variable identifier$\rangle$. Since the automaton is non-deterministic, it can produce a graph of configurations, so let's explore only the branch where the final state is reached. The initial configuration is $(q_0, Z_0, 1)$. Configurations which appear after the first step are produced by accepting the first rule to the current configuration and the input symbol located at the edge between nodes 1 and 2. First of them is the one corresponding to the performed invocation of the *Read* method, the invocation statement is pushed onto the stack and the position is changed to 7. Second of them is the branch where invocation is just skipped. Let's continue with the first configuration $(q_0, i_1 :: Z_0, 7)$ where i_1 is the invocation statement. Next step changes the state to the local $\langle$result$\rangle$ variable according to the rule 2 because the right part of the assignment is a source. Configuration switches to $(\text{Read}.\langle\text{result}\rangle, i_1 :: Z_0, 8)$. Further step processes the return statement using rule 5 and performs two important actions. Firstly, it pops the invocation stored on the top of the stack and jumps to the return point. Secondly, there is performed a change of currently tracked variable from the local one of the method *Read* to the local variable d of the method *Process* because it stores a result of invocation. So, next configuration is $(\text{Process.d}, Z_0, 2)$. Processing of the edge between nodes 2 and 3 uses rule 4 and produces only one branch which just skips the invocation because the target is a filter and there is no need to enter this method. However, the invocation of *Consume* is processed fairly and there is produced the configuration $(\text{Consume.d}, i_2 :: Z_0, 11)$ which, further, iterates until the *Sink* invocation and reaches the final state q_f.

The full chain of configurations is shown in equation 5.

$$\begin{aligned}
&(q_0, Z_0, 1) \xrightarrow[1]{1} (q_0, i_1 :: Z_0, 7) \xrightarrow[2]{2}\\
&(\text{Read}.\langle\text{result}\rangle, i_1 :: Z_0, 8) \xrightarrow[3]{5}\\
&(\text{Process.d}, Z_0, 2) \xrightarrow[4]{4} (\text{Process.d}, Z_0, 3) \xrightarrow[5]{4}\\
&(\text{Process.d}, i_2 :: Z_0, 10) \xrightarrow[6]{4} (\text{Consume.d}, i_2 :: Z_0, 11) \xrightarrow[7]{4}\\
&(q_f, i_3 :: i_2 :: Z_0, \langle \textit{Sink} \text{ entry point}\rangle)
\end{aligned} \tag{5}$$

So, the path in the graph which is present in equation (6) is accepted by the automaton and contains an issue.

$$\begin{tikzcd}
1 \arrow[r, dashed, "\substack{\text{invoke:}\\ \text{this.Read} \to \text{local d}\\ (\text{a := local a})}"] & 7 \arrow[r, "\substack{\text{assign:}\\ (\text{local a}).\text{Source} \to \langle\text{result}\rangle}"] & {} \\
8 \arrow[r, dashed, "\text{return}"] & 2 \arrow[r, "\substack{\text{invoke:}\\ \text{B.Filter} \to \text{local f}\\ (\text{d := local d})}"] & 3 \arrow[r, dashed, "\substack{\text{invoke:}\\ \text{this.Consume}\\ (\text{d := local d})}"] & {} \\
10 \arrow[r, "\substack{\text{invoke:}\\ \text{C..ctor} \to \text{local c}}"] & 11 \arrow[r, dashed, "\substack{\text{invoke:}\\ (\text{local c}).\text{Sink}\\ (\text{d := local d})}"] & \langle \textit{Sink} \text{ entry point}\rangle
\end{tikzcd} \tag{6}$$

Where dashed arrows indicate jumps and solid ones correspond to straightforward transitions.

3 SOLUTION

Using the described idea of automata-based CFL-reachability approach, we have implemented the solution in practice. Our 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.

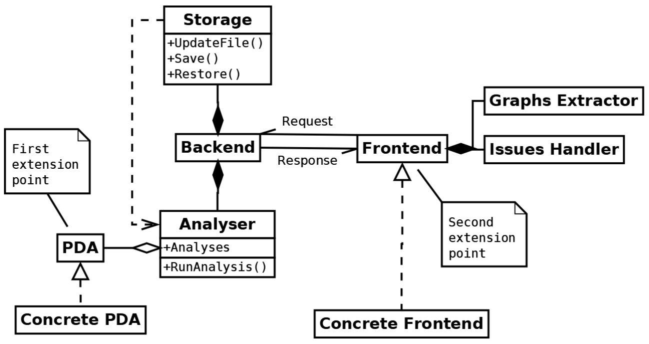


Figure 4: Solution structure

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 improving the one which is provided by IDE by adding dynamic invocations resolving such as lambdas propagation, 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 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 each of which leads to an error, maps it to the source code and does something with this information. For example, it can highlight pieces of code which participate in the error producing.

Since a frontend is completely separate from the backend and the only requirement for it is to follow the protocol, the frontend can be considered as the second extension point. I.e. it is possible to replace the currently implemented frontend with any other implementation having the same functions as the original one including graphs extraction and results processing. The current implementation is also open to modifications which add support for new types of analysis or any other features which requires interaction with IDE.

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 the analysis, for example, when the IDE performs the code highligighting, the frontend asks the backend for found issues, maps results to the source code and highlights corresponding lines.

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.

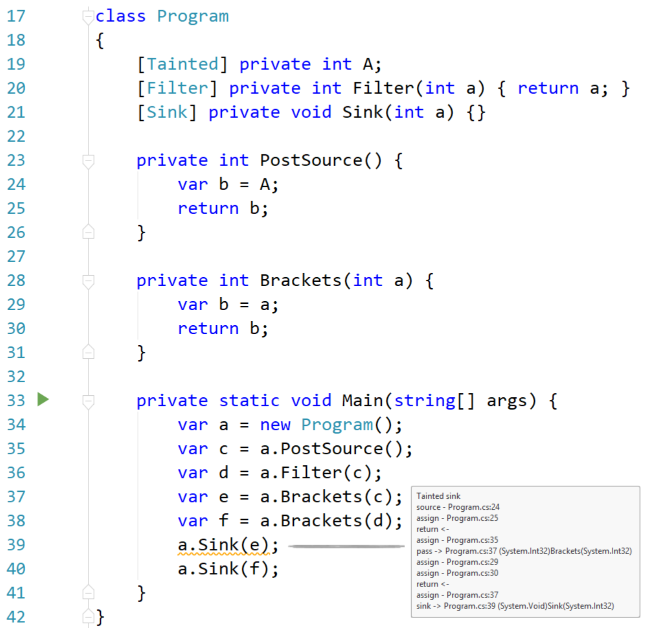
The first analysis which has been implemented is the considered taint tracking analysis. It is defined just by the PDA constructed in the section 2 translated into the code with some slight modifications which make it possible to process interactions with object fields. To provide more information about an issue found by this analysis, the higlighting is accompanied by bulbs containing the full path of tainted variable from the source to the sink represented as the sequence of operations.

4.1 Sample cases

Let's look how the resulting solution works on a few common cases which help to illustrate some properties of the analyser. Each case is shown at a screenshot 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. This example illustrates the most common cases of interprocedural data passing. *Brackets* method gets the data, possibly performs some computations on them and returns the result. Invocations at lines 37 and 38 shows that the solution can distinguish two data flow paths despite both of them passes through the same method. So, e becomes tainted because c is tainted and f does not because d is clear. Moreover, the solution can track paths where passes and returns do not form the correct bracket sequence that is shown by method *PostSource* which does not take any parameter and just returns tainted data.

Secondly, the solution has the 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



```
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      private static void Main(string[] args) {
34          var 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)
```

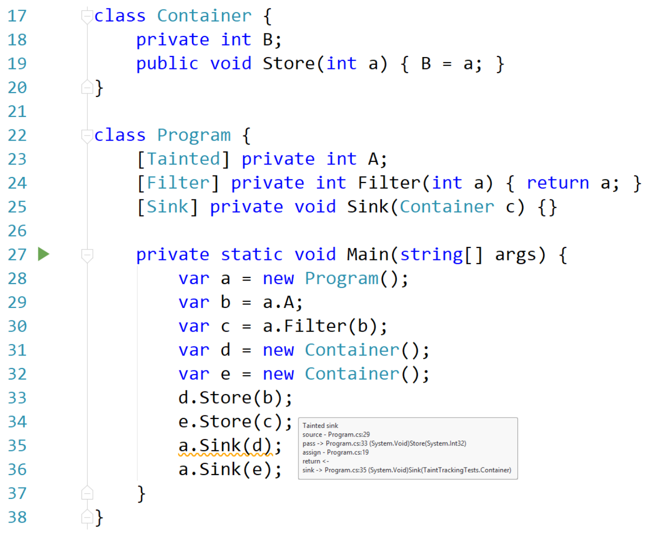
Figure 5: Flow sensitivity

methods and by outer code interacting with their fields directly. The first case is shown at fig 6. There is the field B at the line 18. This field can be used widely in the logic of the *Container* class and by this the tainting of this field is considered as the tainting of the whole object. However, while processing of the method *Store* during the analysis it is hard to decide what the object need to be tainted because in the inner context of *Store* it is just *this* object. I.e. we must consider the calling context to make such decision. So, the solution provides this opportunity which is shown by lines 33-36 where the first invocation of *Store* leads to the tainting of object d and the second invocation does not taint object e.

Finally, the solution works with any type of recursion and does not fall into infinite cycles. It can be seen at fig. 7. This snippet contains two mutually recursive methods which pass the data to each other. The solution checks all possible paths of passing even those which includes cyclic invocations and returns the passed variable to the point corresponding to the initial invocation.

4.2 Performance

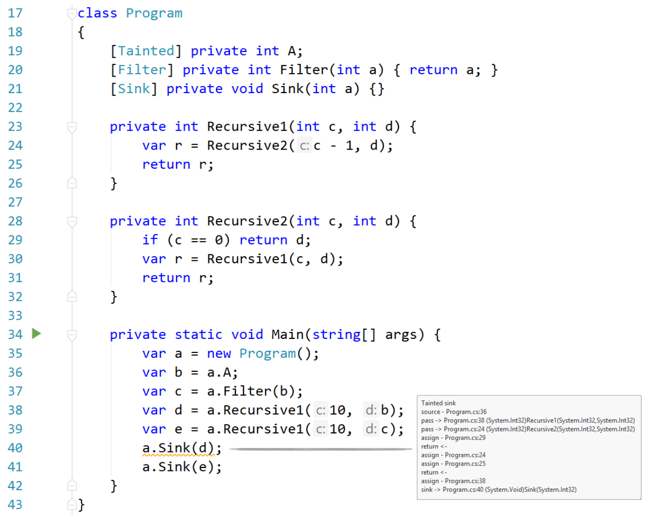
It is also necessary to measure the performance of the resulting solution. Because the implemented taint tracking analysis forces to mark all participating entities manually, it is difficult to perform it on some large project. However, there is another intermediate analysis which is runned before any other one to collect some information required by resolver. In particular, it tracks propagation of all variables to discover all possible concrete types of each variable. So, it involves each variable and each method in the whole program and thus the



```
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      private static void Main(string[] args) {
28          var 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: Tainting of an object by its own method



```
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 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      private static void Main(string[] args) {
35          var 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      }
43  }
```

```
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: Recursive methods processing

time and space required for execution of this analysis may be consistent estimation of the efficiency of the solution.

The code base which has been chosen as a source of data is the full solution of the Mono project. The solution has been tested on a computer running Windows 10 with quad-core Intel Core i7 3.4 GHz CPU and 16 GB of RAM. The results is shown in the table 1.

Project	Classes	Methods	Execution time (s)	Allocated memory (GB)
Mono	21013	192745	21 ± 0.5	~ 4.2

Table 1: Performance

5 CONCLUSION

TODO: CONCLUSION

REFERENCES

[1] John E. Hopcroft and Jeffrey D. Ullman. 1990. *Introduction To Automata Theory, Languages, And Computation* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. *SIGPLAN Not.* 36, 3 (Jan. 2001), 54–66. https://doi.org/10.1145/373243.360208