CFL-Reachability Based Framework for Interprocedural Static Code Analysis Development

Ilya Nozhkin
Saint Petersburg State University
St. Petersburg, Russia
nozhkin.ii@gmail.com

Semyon Grigorev
Saint Petersburg State University
St. Petersburg, Russia
JetBrains Research
St. Petersburg, Russia
s.v.grigoriev@spbu.ru
semen.grigorev@jetbrains.com

ABSTRACT

We propose an extensible framework for interprocedural static code analysis implementation. Our solution is based on CFL-reachability: analysis is formulated in terms of context-free constrained reachability in the interprocedural graph. Extensible architecture allows one to implement new analysis and integrate it into the IDE of choice or static code analysis tool. To demonstrate the abilities of our solution, we implement the plugin which provides basic taint analysis and label flow analysis upon ReSharper infrastructure. We demonstrate its applicability for real-world problems.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Integrated and visual development environments**; **Software defect analysis**; *Software safety*; • **Theory of computation →** *Grammars and context-free languages.*

KEYWORDS

Static code analysis, interprocedural analysis, CFL-reachability, taint analysis, IDE, plugin, context-free languages, PDA

ACM Reference Format:
Ilya Nozhkin and Semyon Grigorev. 2019. CFL-Reachability Based Framework for Interprocedural Static Code Analysis Development. In *Proceedings of Central and Eastern European Software Engineering Conference Russia (CEE-SECR'19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static code analysis is an important part of modern software development tools. It takes care of verifying the correctness of some program's behavior freeing a programmer from this duty. Static analysis can be classified as intraprocedural or interprocedural depending on which scope of a program it uses. Intraprocedural analysis considers only one procedure whereas interprocedural inspects a program as a whole. Interprocedural analysis, in theory, are more precise due to the amount of available information.

```
File 1:

1  class A {
2    [Tainted]
3    int Source;
4  }
5
6  class B {
7    [Filter]
8    static int Filter(int d);
9  }
10
11 class C {
12   [Sink]
13   void Sink(int d);
14 }
```

```
File 2:

1  class D {
2    void Process(A a) {
3      int d = Read(a);
4      int f = B.Filter(d);
5      Consume(d);
6      AnotherConsume(f);
7    }
8
9    int Read(A a) {
10     return a.Source;
11   }
12
13   void Consume(int d) {
14     C c = new C();
15     c.Sink(d);
16   }
17
18   void AnotherConsume(int d) { ... }
19 }
```

Figure 1: Sample code in C# which represents a case for interprocedural taint analysis

One classical problem which requires interprocedural analysis is taint tracking problem (e.g. described in [2]). Input data can have an inappropriate format or contain an exploit such as SQL injection. Such data is called *tainted*. Tainted data can lead to incorrect behavior in case of incorrect format, or can cause a security issue.

A simple example of C# code which may require taint analysis is presented in fig. 1. We use this example to illustrate our solution. First of all, we introduce a number of entities which play important roles in analysis. The first is *source*: a field that potentially contains the tainted data, for example, the field *Source* of the class *A*. The second is *sink*: a method which is vulnerable to tainted data. In our example, the method *Sink* of the class *C* has this property. The third important type of entities is *filter* (or *sanitizer* in some other definitions). It is a method that checks the correctness of data passed into it. If data is incorrect, *filter* throws an exception or modifies data to ensure the correctness of the result. The method *Filter* of the class *B* in the given snippet is a filter.

We assume that each entity is annotated by a programmer with an appropriate attribute: *[Tainted]* for sources, *[Filter]* for filters and *[Sink]* for sinks. The taint analysis problem is to find all traces such that tainted data can flow into a sink bypassing any filter.

In our example the class *D* extracts data using the class *A* as a source, then validates it using class *B*, after that performs some computations involving the class *C*. Invocation of the method *Consume*

CEE-SECR'19, November 14–15(16), 2019, St.Petersburg, Russia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

leads to using the data which have not been properly validated. To identify such issues we need to run the taint analysis.

One of the well-known frameworks for interprocedural static code analysis is a CFL-reachability framework proposed by Thomas Reps [9]. This framework is generic: it provides the abstraction which allows one to implement many different types of interprocedural static code analysis, such as pointer analysis [15, 17], taint analysis [6], label flow analysis [7, 8], library summarization [13]. Moreover, CFL-reachability is a long-time studied framework and thus there is a number of specific solutions which demonstrate reasonable performance in practice [14].

It is important for static code analysis to be flow-sensitive that means correct handling call-return pairs: when we have an interprocedural graph, we have no explicit synchronization of call and return edges. As a result, one can find a path which contains incorrect sequences of calls and returns. Also, it is important to provide context-sensitivity: correct handling of read-write, get-set and other pairs of edges (operations). Note, that CFL-reachability framework allows one to implement analysis which can be flow-sensitive or context-sensitive, but not both [10]. In order to provide both flow- and context-sensitivity one can use more powerful framework [16]. In our examples, we implement flow-sensitive analysis, but one can modify it to be context-sensitive.

The main idea of CFL-reachability is to find paths in a graph that satisfy constraints defined by a context-free grammar. In particular, a path is accepted if the concatenation of labels on its edges forms a word which can be derived in the given grammar. However, in practice, there are few drawbacks of such a grammar-based definition. Firstly, grammar-driven parsing is based on exact matching of terminals which forces to generate a very large grammar in the case when edges contain some unique attributes. For example, brackets matching described in [8] or [15, 17] requires to generate as many rules as there are call sites in the source code. In our example, to handle all call-return pairs in paths correctly, the grammar should contain the following productions.

$$
\begin{aligned}
&\dots \\
&B \to (_{2:3}\ B\ )_{2:3}\ B \\
&B \to (_{2:4}\ B\ )_{2:4}\ B \\
&B \to (_{2:5}\ B\ )_{2:5}\ B \\
&B \to (_{2:6}\ B\ )_{2:6}\ B \\
&B \to (_{2:15}\ B\ )_{2:15}\ B \\
&\dots \\
&B \to \varepsilon \\
&\dots
\end{aligned}
\tag{1}
$$

Where $(_{\text{file:line}}$ means a call from the specific position and $)_{\text{file:line}}$ means a return to the specific position in the code.

Secondly, grammar provides an additional level of abstraction. This level is useful for algorithms formalization, but it is not as flexible enough for practical applications.

CFL-reachibility requires a graph representing the whole program. It is not feasible to construct such a graph from scratch each time a programmer modifies a program. We need to ensure that the graph is modified with minial effort. To create a generic framework we also need to make sure the graph representation is generic.

The main goal of our work is to implement a tool solving all mentioned problems and thus allowing to use CFL-reachability in practice. We make the following contributions in the paper.

- We describe the scalable representation of a graph which allows containing as much information about the program as necessary.
- We introduce the approach for the definition of constraints based on pushdown automata instead of grammars that eases formulation of analysis.
- We present the implementation of the proposed approach which allows one to create new types of analysis using introduced abstractions. Also we implement[1] a plugin which uses the solution to provide analysis results to ReSharper, Rider, and InspectCode.
- We evaluate the implementation on both synthetic tests and large open source projects.

2 ANALYSIS DEFINITION

To apply the CFL-reachability framework, the first thing to do is to define the representation of a graph and path constraints. We assume that a graph is an image of a program. It stores the information about the source code that can be mapped back to locate issues in the source code which are found by the analysis of the graph. Constraints on paths define sequences of operations leading to an issue.

In order to keep the expressive power of the CFL-reachability approach but facilitate implementation, we propose to use pushdown automata instead of grammars which still have equivalent expressiveness [5]. Further, we take a 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 the analysis is a union of control flow graphs each of which is extracted from one distinct method. The graph which corresponds to our example is shown in fig. 2.

Each edge contains an operation that represents a statement in the source code. The target of the edge indicates the position to jump after the execution of the operation. Each operation has a type and a set of attributes such as invocation target or passed arguments. The number of types is not fixed and some analysis-specific operations can be added if necessary. In our example, we consider three different types of operations: invocations, assignments, and returns. Each of them is an image of some source code instruction. Invocations are produced from call sites and have the same information as in the original code. Their notation has the following form:

$$
\begin{aligned}
&\text{invoke:} \\
&o.m \to v \\
&(f_1 := a_1; \dots; f_k := a_k)
\end{aligned}
\tag{2}
$$

Here m is a method of an object (or a class) o, v is a variable which stores result, and (f_i, a_i) are pairs of the formal and the actual parameters respectively.

[1] Source code and executables can be downloaded here: github.com/gsvgit/CoFRA.

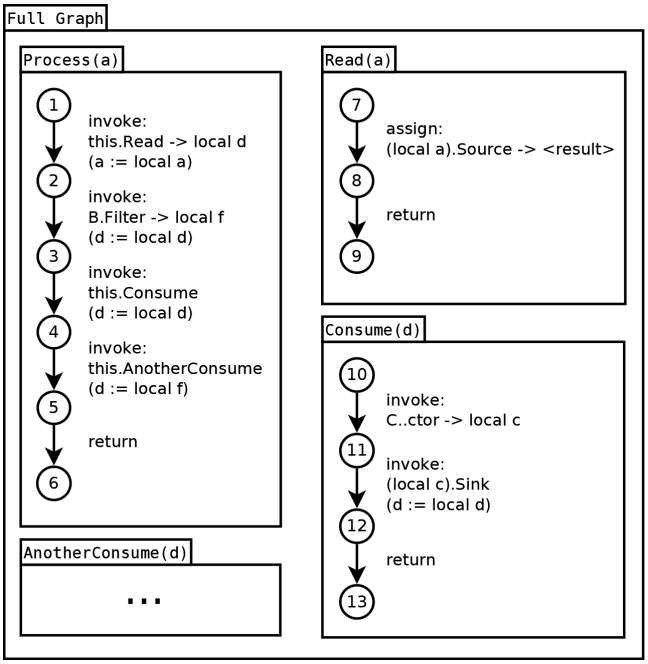


Figure 2: Extracted graph for the code presented in fig. 1

Assignment operation in the source code is represented in the following way:

$$\begin{aligned} &\text{assign:} \\ &s \longrightarrow t \end{aligned} \tag{3}$$

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

Return statement indicates the end of a method. It is necessary to add this statement even it does not exist in the source code (for example, if return type of method is *void*) to inform the analyzer about the return point.

Nodes correspond to positions between instructions in the source code.

Each constructed graph represents the content of a single method. To represent interprocedural jumps taknig place during the invocation, individual graphs should be interconnected. There are several ways 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 the target and one to emulate the return from the final node of the method to the caller. In this approach, connections should be updated whenever a method is removed or changed. To mitigate this shortcoming, we propose to instead resolve invocations dynamically during analysis. This way a graph is composed from the individual graphs for methods with no additional edges and no modifications are needed when methods are chaged. There still should be a *resolver*: a mechanism which collects all targets of an invocation by the references stored.

To implement a resolver, we use meta-information about the program structured as shown in fig 3.

There are several important aspects of which we should keep track.

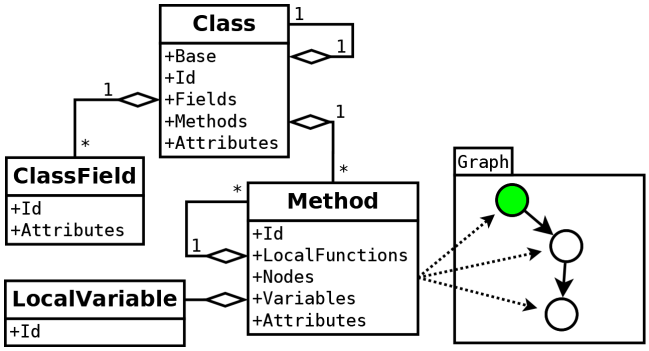


Figure 3: Structure of the metadata for extracted graph

(1) It is important to keep the hierarchy of inheritance to support polymorphic calls and invocations of methods of a basic class. This hierarch is supported by using the *Base* filed of the *Class* class.
(2) For each method we store which method it belongs to: each class contains a list of methods.
(3) To support anonymous functions invocations and passing of delegates we also keep track of the local functions. This information is stored in the *LocalFunctions* field of the *Method* class.
(4) Methods are linked to nodes of the graphs constructed for them to find for their entry points and to simplify metod's graph updating. The *Nodes* field of the *Method* class stores this information.
(5) To adjust analyzer's behaviour to attributes attached to entities, we store them in the field *Attributes* of classes *Class*, *Method* and *ClassField*.

This structure also contains class fields and local variables of a method which can be referenced by operations. Given a class name and a method identifier, the resolver traverses the hierarchy to find all necessary entities.

2.2 Pushdown automata construction

We define path constraints in terms of pushdown automata. Formally, nondeterministic pushdown automaton or PDA [5] 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 $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*)$ is a transition relation which takes the current state, an input symbol, the top of the stack and computes a new state and the sequence of stack symbols to replace the top one. We also impose the following restriction on the transition relation. Only one symbol can be pushed during the transition. It does not affect the expressive power of the resulting abstraction: a sequence of symbols can be pushed one by one using the chain of states connected by ε-transitions each of which pushes one symbol.

Denote the set of all edges in the control flow graph as Σ. All other sets can be chosen arbitrary.

Invocation semantics is to jump from the current position to the entry point of the target instead of following the current edge. To reflect this behaviour we change the codomain of δ to $\mathcal{P}(Q \times \Gamma^* \times$

$N \cup \{v\}$), where N is a set of graph nodes and v is a dummy value which signifies no jump is needed.

To illustrate the construction of PDA by example we perform the taint tracking analysis described in the introduction. Let $Q \coloneqq V \cup \{q_0, q_f\}$, where V is a set of all local variables of every method and q_0 and q_f are dummy initial and final states, so $F \coloneqq \{q_f\}$. $\Gamma \coloneqq I \cup \{Z_0\}$, where $I \subset \Sigma$ is a set of all edges containing an invocation and Z_0 is a dummy initial stack symbol. δ is defined by the case analysis (4).

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

Here we use the following notation.

- $v@\mathit{pattern}$ means that v must be an object which is constructed by *pattern*.
- *source* checks if a variable is a source.
- *returned* checks if current variable is a return value of some method.
- T returns the target node of an edge.
- RV returns the variable which stores the result of an invocation.
- R is the resolver returning entry points of all possible targets of an invocation.
- A is defined by the equation (5).

$$
A(i, j, v) \coloneqq \begin{cases} \{q_f\}, & \text{if } j\text{-th target of the invocation } i \\ & \text{is a sink and } v \text{ is its argument} \\ \{v_k : v \mapsto v_k\}, & \text{if } j\text{-th target of i} \\ & \text{is not filter} \\ \varnothing, & \text{otherwise} \end{cases} \tag{5}
$$

Here $v \mapsto v_k$ means that v is passed as the k-th parameter and becomes the local variable v_k of the target.

2.3 Analysis execution

In this section we describe how the constructed automaton can be used to find an issue in the sample source code. The goal is to find the sequence of operations which starts in the entry point of the source methods and ends in the entry point of a sink method. Since the automaton accepts such sequences, we can simulate the switching of its configurations according to the input statements taken from the source graph. We use the following notation to describe the PDA steps.

- $(q, \gamma_1 \ldots \gamma_k, n)$ 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 (4).
- ⟨Name of the containing method⟩.⟨Variable identifier⟩ is a local variable.

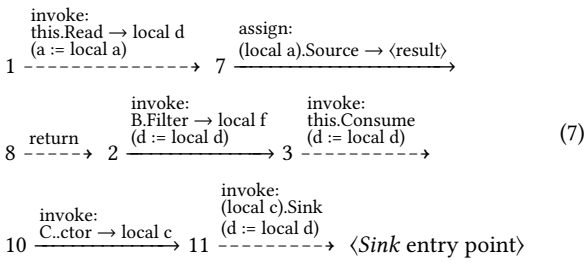
Note, that the automaton is non-deterministic, thus it can produce a graph of configurations, but in this example, we only explore a single branch in which the final state is reached. To get the optimal way to simulate non-deterministic PDA, one should use techniques from generalized parsing. In our work we use GLL-based [3, 11] interpreter.

According to example, we start with the configuration $(q_0, Z_0, 1)$. 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 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 is the branch where invocation is skipped. Since the second configuration does not lead to an error, we 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 ⟨result⟩ variable according to the rule 2 because the right part of the assignment is a source. Configuration switches to (Read.⟨result⟩, $i_1 Z_0$, 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, *Read.⟨result⟩* is changed to *Process.⟨d⟩* because it stores a result of the invocation. The next configuration is (Process.d, Z_0, 2). Processing of the edge between nodes 2 and 3 uses rule 4 and produces only one branch which skips the invocation because the target is a filter and there is no need to enter this method. The invocation of *Consume* is a regular call, thus the next configuration is (Consume.d, $i_2 Z_0$, 10). From this point algorithm iterates until the *Sink* invocation and reaches the final state q_f.

The complete chain of configurations is shown in equation 6.

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

Equation 7 is the path in the graph which contains an issue.



(7)

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

3 IMPLEMENTATION DETAILS

Using the described idea, we developed the framework which makes it possible to implement interprocedural static code analysis based on CFL-reachability approach. 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 in fig. 4.

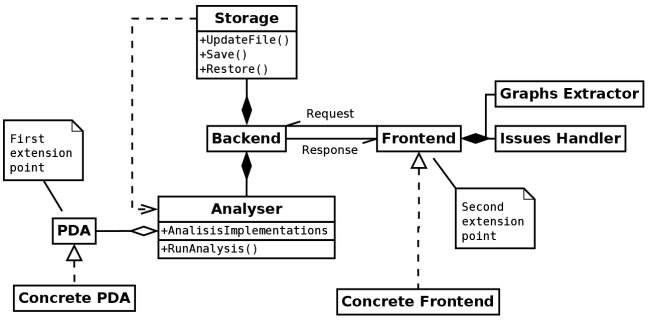


Figure 4: Solution structure

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. The first is a database which provides the continuous incremental updating of the graph and its metadata It also supports storing the data to a disk and reloading it at the beginning of the session. The second is responsible for the execution of analyses. It contains an implementation of the resolver improving the one which is provided by the IDE by adding dynamic invocations resolving such as lambdas propagation, and the algorithm of PDAs running. This subsystem is the first extension point which provides API for new analysis creation. We provide several pre-implemented analyses in the backend. One can implement a new analysis by implementing an appropriate PDA as an instance of generic abstract class. New analysis is run using the existing internal algorithm of PDA simulation. The result of analysis is a finite subset of paths in the graph which are accepted by the PDA.

The second 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 one is a results interpreter which receives the set of erroneous paths in the graph, maps them to the source code and translates them to human-readable format. Using this subsystem, one can, for example, highlight erroneous code or propose the ways to improve it.

Since a frontend is completely separate from the backend and the only requirement for it is to follow the communication protocol, the frontend can be considered as the second extension point. It can be replaced with another implementation which also provides features of graph extraction and further 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 an IDE.

The protocol itself is based on request-response pattern. The frontend informs the backend about changes in the source code and asks it to update the database accordingly. To highlight the source code in an IDE, the frontend asks the backend for the analysis results and map them back to the code.

4 EVALUATION

In order to test the resulting solution we have implemented the frontend as a plugin using ReSharper SDK[2], so it can be installed into ReSharper[3], Rider[4] and InspectCode[5]. 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 sample analysis which has been implemented is the taint tracking analysis. It is defined by the PDA constructed in section 2. We slightly modified it to better process interactions with object fields. We accompany code highlighting with the complete path of a tainted variable from a source to a sink as a sequence of operatoins. In ReSharper this information is shown when mouse is hovering over the problematic code.

4.1 Taint analysis sample cases

The resulting solution has been tested on a few common cases which can be found in the repository[6]. We demonstrate the principal properties of the analyzer with three of them. We illustrate the examples with screenshots of the Rider IDE. We relocated the tooltips to not cover the source code.

The first feature is ensuring flow sensitivity. This means that when the flow of variables being passed into and returned from the method is processed, we distinguish between different call sites and returns are done to the appropriate return points. This is shown in 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 show that the solution can distinguish two data flow paths despite both of them passes through

[2]ReSharper developer's guide: https://www.jetbrains.com/help/resharper/sdk/README.html. Access Date: 15.08.2019
[3]ReSharper: https://www.jetbrains.com/resharper/. Access Date: 15.08.2019
[4]Rider IDE: https://www.jetbrains.com/rider/. Access Date: 15.08.2019
[5]InspectCode tool: https://www.jetbrains.com/help/resharper/InspectCode.html. Access Date: 15.08.2019
[6]Test cases: github.com/gsvgit/CoFRA/tree/master/test/data/TaintAnalysis

the same method. So, e becomes tainted because c is tainted and f does not because d is clear.

```
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

The second feature is context sensitivity. Context sensitivity is necessary to track the propagation of objects that are tainted by assigning of some fields inside them both by their own methods and by outer code interacting with their fields directly. However, it is impossible to provide true context-sensitivity since it cannot be expressed alongside with flow-sensitivity [12]. As a result, in our analysis we provide a limited context sensitivity, which is shown in fig 6. There is the field B in 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 needs to be tainted because in the inner context of *Store* it is just *this* object. We must consider the calling context to make such decision. The solution provides this opportunity which is illustrated 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. This case is demonstrated in fig. 7. This snippet contains two mutually recursive methods which pass the data to each other. The solution checks all possible paths of passing including those with 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. To measure performance we could have used the taint analysis, but it requires to equip all points of interest by corresponded attributes manually. Considering this, we decided to instead implement a different kind of analysis which tracks the propagation of all

```
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

variables and by this explores any possible path in the graph. Thus, the time and space required for its execution should be a consistent estimation of the efficiency of the solution.

The code base which has been chosen as a source of data is the full solutions of a few big projects: Mono[7], EventStore[8] and OpenRA[9]. The analyzer 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 are shown in the table 1. Execution time does not include the time required for graph construction.

[7] Source code of mono project: https://github.com/mono/mono. Access Date: 15.08.2019
[8] Source code of EventStore project: https://github.com/EventStore/EventStore. Access Date: 15.08.2019
[9] Source code of OpenRA project: https://github.com/OpenRA/OpenRA. Access Date: 15.08.2019

Project	Classes	Methods	Execution time (s)	Allocated memory (GB)
Mono	21013	192745	21 ± 0.5	~ 4.2
EventStore	3828	22796	2.7 ± 0.1	~ 0.49
OpenRA	2767	15451	2.3 ± 0.1	~ 0.4

Table 1: Performance

We conclude that our solution is able to process real-world projects in acceptable time.

5 CONCLUSION

We propose and implement in C# the generic framework for interprocedural static code analysis. This framework allows one to implement arbitrary interprocedural analysis in terms of CFL-reachability. With the proposed framework, we implement a plugin upon ReSharper infrastructure which provides simple taint analysis and demonstrate that our solution can handle important real-world cases. We show that the proposed framework can be used for real-world analysis.

One possible direction for future work is improving the implemented framework: tuning performance and improving APIs. We believe, that the best way to do this is by employing the framework for more analyses and real-world projects.

Another direction is a practical evaluation of automatic fix location prediction by using minimum cuts method [1]. Such function may be helpful for end-users: tool can propose possible fixes of the detected problem, not only report on it.

We want to compare the proposed approach with other generic CFL-reachability based approaches for interprocedural code analysis such as the generation-based approach [4]. We should compare both frameworks and specific tools created with frameworks.

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100.

REFERENCES

[1] Andrei Marian Dan, Manu Sridharan, Satish Chandra, Jean-Baptiste Jeannin, and Martin Vechev. 2017. Finding Fix Locations for CFL-Reachability Analyses via Minimum Cuts. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 521–541.

[2] Jonas De Bleser, Quentin Stiévenart, Jens Nicolay, and Coen De Roover. 2017. Static Taint Analysis of Event-driven Scheme Programs. In *Proceedings of the 10th European Lisp Symposium*. ACM, 80–87.

[3] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104

[4] Nicholas Hollingum and Bernhard Scholz. 2017. Cauliflower: a Solver Generator for Context-Free Language Reachability. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 171–180. https://doi.org/10.29007/tbm7

[5] 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.

[6] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

[7] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. Existential Label Flow Inference Via CFL Reachability. In *Static Analysis*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 88–106.

[8] 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

[9] Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS '97)*. MIT Press, Cambridge, MA, USA, 5–19. http://dl.acm.org/citation.cfm?id=271338.271343

[10] Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 162–186. https://doi.org/10.1145/345099.345137

[11] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.

[12] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

[13] Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. 2017. Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 880–908.

[14] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. *SIGOPS Oper. Syst. Rev.* 51, 2 (April 2017), 389–404. https://doi.org/10.1145/3093315.3037744

[15] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2001420.2001440

[16] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. *SIGPLAN Not.* 52, 1 (Jan. 2017), 344–358. https://doi.org/10.1145/3093333.3009848

[17] Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464