Universidade do Minho
Escola de Ciências

Artur Jorge Gomes Queiroz

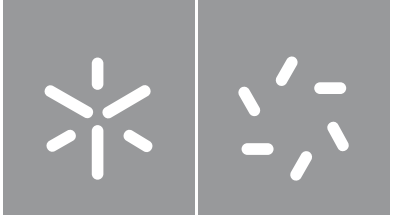
Scalable Detection of Security-Vulnerabilities in Source Code

outubro de 2022

Scalable Detection of Security-Vulnerabilities in Source Code

Artur Jorge Gomes Queiroz

UMinho | 2022



Universidade do Minho
Escola de Ciências

Artur Jorge Gomes Queiroz

Scalable Detection of Security-Vulnerabilities in Source Code

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho efetuado sob a orientação do(a)
Professor Doutor José Carlos Espírito Santo
e do
Professor Doutor José Nuno Oliveira

outubro de 2022

COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

ACKNOWLEDGEMENTS

First of all, I would like to give my thanks to Checkmarx, for proposing an interesting and a challenging theme for this dissertation, for giving the oportunity of studying in their mentorship and installations, and also for sharing the basic and pointing to the right direction to study for the theme of detection of vulnerabilities, which I was going fresh to this theme.

Many Thanks to the University of Minho, for the excellent conditions, professors and teaching everything that I've needed to complete this dissertation, for the part of Informatics and also for the part of Mathematics, a lot of tools taught to me were used and contributed, without a shadow of a doubt, in the completion of this dissertation.

For the supervisors of the part of Checkmarx, namely, Daniela da Cruz and Meir Benayoun, thanks for being with me week after week, going out of your way to help me in my lows and highs, for guiding me in my many ideas, until the right one showed up.

Now I would like to resort to my native language, for the rest of this chapter.

Para que esta dissertação fosse possivel logo desde o principio, tenho de agradecer ao meu diretor de curso deste mestrado, Pedro Patrício, pois apresentou-me à Daniela da Cruz que por parte da Checkmarx, junto com o Meir Benayoun, começou o processo de arranjar este tema tão interessante e atual.

Seria impossível não falar dos meus orientadores da parte da Universidade do Minho, que tiveram sempre disponiveis e a fazer as questões certas de forma a guiarem-me para o melhor caminho para a dissertação, e não me deixarem perder demasiado em soluções que não me levariam ao sucesso. Também a insistência para me fazer continuar e os seus conselhos gerais para métodos científicos e matemáticos.

Para a minha companheira, Bruna agradeço-te por todo o apoio emocional, principalmente para estes tempos de pandemia onde houve muitas mudanças, fazendo-me manter a calma e analisar aquilo que realmente me faria seguir em frente, desde o primeiro dia desta dissertação.

Tenho também de incluir os meus colegas, apesar de não contribuirem diretamente, a contribuição indireta foi igualmente importante, fazendo-me distrair e com isso descansar em momentos mais desgastantes.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

With the advent of web technologies, services are becoming increasingly accessible on-line, for user convenience. Because of this, such systems are more susceptible to vulnerabilities, and several tools have emerged to do code analysis, not only static, but also dynamic, with the aim to detect abnormalities, such as vulnerabilities.

State-of-the-art solutions for calculating data flow paths in the analysed code suffer from efficiency problems. Inefficiency appears because of the path representation, as it leads to a less efficient implementation.

This dissertation proposes a new approach that obtains data flow paths on demand, from a representation that can store efficiently a set of paths and that, as a result, can have better results in terms of time efficiency.

Benchmarks of the naive solution to the problem and the proposed solution in the dissertation show a drastic difference in time complexity, as the theory predicted, obtaining a time complexity $\geq \mathcal{O}(e^2)$, for the simple solution, and a $\mathcal{O}(n + e)$, for the solution presented here, where e is the number of edges and n the number of nodes of the DFG extracted from the source code.

RESUMO

Com o advento das tecnologias web, os serviços estão a tornar-se cada vez mais acessíveis on-line, para a conveniência do usuário. Devido a isso, os tais sistemas são mais suscetíveis a vulnerabilidades, e diversas ferramentas surgiram para fazer análise de código, não apenas estática, mas também dinâmica, com o objetivo de detectar anormalidades, como vulnerabilidades.

As soluções que compõem o estado de arte no cálculo de caminhos do data flow, têm problemas de eficiência nessa procura. A ineficiência aparece por causa da representação dos caminhos, pois leva a uma implementação menos eficiente.

Esta dissertação propõe uma nova abordagem que obtém os caminhos do data flow de forma "on demand", a partir de uma representação que guarda de forma eficiente um conjunto de caminhos e por consequencia consegue ter melhores resultados em termos de eficiência em tempo. Este mecanismo tem até uma boa maneira de generalizar para algo modular.

Testes de tempo da solução ingênua para o problema e a solução proposta na dissertação mostram uma diferença drástica na complexidade do tempo, como a teoria previa, tendo uma complexidade de tempo de $\geq \mathcal{O}(e^2)$, para a solução simples, e $\mathcal{O}(n + e)$, para a solução apresentada aqui, onde e é o número de edges e n o número de nodos do DFG.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

AST	Abstract Syntax Tree
BFS	Breath First Search
CFG	Control Flow Graph
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DFS	Depth First Search
LOC	Lines Of Code
OWASP	Open Web Application Security Project
SAST	Static Application Security Testing
SQL	Structured Query Language

1. INTRODUCTION

Nowadays, the digital world is growing, but must importantly the online world is growing. Because of this, people and companies are becoming more dependent on online services, while important and private information is stored in cloud based services. Vulnerability of such services means that the important and private data of their clients also becomes vulnerable to cyberattacks.

The important and private data can be passwords, credit card information, addresses, or even secret information that does not seem harmful to someone, but it can be so if exposed by an intruder who can put the identity of a person at risk, or even worse. If someone has your name, date of birth, address, Social Security number, phone numbers and more, each one of these data on its own could be thought out as not important, but all combined can enable a social hacking attack by impersonating an individual or group who is directly or indirectly known to the victims or by representing an individual or by a group in a position of authority. On the Internet, data has high value. If stolen, it can be collected, maliciously analysed and even sold.

Whenever you download an app, visit a website or use a social media platform, the company is likely to be collecting data about you. People are doing a lot more online activities through their computers and mobile devices today. We make purchases, look up medical conditions, interact with friends and relatives, organize vacations, almost anything imaginable. With these actions, people are inadvertently creating a huge digital footprint of data about themselves.

The number of cyberattacks is increasing yearly, with no signs of stopping [Alb19]. By detecting a vulnerability in a system, in production, sometimes it is too late, because it could already been used for malicious purposes. Sometimes, to correct the problem, a change in the source code of the application in question is needed. That is why applications who store important information need to test the product for vulnerabilities before launching it to the public.

The detection of vulnerabilities in the source code of an application is very helpful or even, in some cases, needed. Such detection can be done by human security analysts or by an automatic analysis tool (for example, a SAST). Without this security analysis, the only way to detect security vulnerabilities is exposing users to such vulnerabilities and experience cyberattacks. As in most cases that is unacceptable, the analysis of the source code itself is a very important activity.

1.1 VULNERABILITIES

One definition of what is a vulnerability is given in the RFC 4949 internet security glossary [Shi07]:

> *A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.*

This glossary regards, in an informal way, a vulnerability as a software security bug. When learning about vulnerabilities, examples are essential to build an intuitive sense of what is a vulnerability.

One such example could be injections, which are a broad class of vulnerabilities whereby an attacker supplies untrusted input to a program. This input gets processed by an interpreter as part of a command or query. In turn, this alters the intended execution of the command or query.

A powerful and stubborn example of an injection vulnerability is SQL Injection. Powerful, because it is one of the most dangerous issues for data confidentiality and integrity in web applications and has been listed in the OWASP Top 10 list of the most common and widely exploited vulnerabilities since its inception. Stubborn, because it is an old vulnerability, dating back to 1998 [rr98] which, still in 2021, is considered very dangerous, even reaching the first place in OWASP is Top 10 Web Application Security Risks in the years 2017 and 2020.

An example A good explanation of the SQL Injection vulnerability can be found in [Anl02], followed below. An SQL injection occurs when an attacker is able to insert a series of SQL statements into a 'query' by manipulating data input into an application.

A typical SQL statement looks like this:

```
select id, forename, surname from authors
```

This statement will retrieve the 'id', 'forename' and 'surname' columns from some 'authors' table, returning all rows in the table. The 'result set' could be restricted to a specific 'author' like this:

```
select id, forename, surname
from authors
where forename = 'john' and surname = 'smith'
```

An important point to note here is that the string literals 'john' and 'smith' are delimited with single quotes. Presuming that the 'forename' and 'surname' fields are being gathered from user-supplied input, an attacker might be able to 'inject' some spurious SQL into this query, by inputting values into the application like this:

```
Forename: jo'hn
Surname: smith
```

The 'query string' becomes this:

```
select id, forename, surname
from authors
where forename = 'jo'hn' and surname = 'smith'
```

When the database attempts to run this query, it is likely to return an error:

```
Server: Msg 170, Level 15, State 1, Line 1
Line 1: Incorrect syntax near 'hn'.
```

The reason for this is that the insertion of the 'single quote' character 'breaks out' of the single-quote delimited data. The database then tries to execute 'hn' and fails.

However, should the attacker have opted for,

```
Forename: jo'; drop table authors--
Surname:
```

then the outcome would be the deletion of the authors' table [Anl02].

Empirical Study By hiring web developers to conduct manual code reviews of a small web application which had seven known vulnerabilities, the empirical study reported in [EHR+13] found that:

1. none of the subjects spotted all confirmed vulnerabilities,
2. more experience does not necessarily mean that the reviewer will be more accurate or effective,
3. reports of false vulnerabilities were significantly correlated with reports of valid vulnerabilities.

The main conclusion is that humans are not as good at detecting vulnerabilities as we give ourselves credit for, at least well categorized ones. So the next step would be to use machines to detect them automatically. Of course, because the nature of a vulnerability is dependable on the system's security model, machine checking will not detect all of them, but a good part of them would be a helping hand.

Further to what has been mentioned already, the growth of hacker attacks that exploit every single vulnerability found in the victim company's software system can make a significant money loss for the victim and subsequently to the company. This motivates companies to make products with no vulnerabilities. Because of this, bug bounty programs exist that offer money incentives for bug finding (normally security bugs, i.e. vulnerabilities), which makes every day people, or more commonly white hats (ethical computer hackers) report the vulnerabilities found for money, which could save the company from an attacker. But still, the product is at risk, because it is exposed, therefore can be attacked, with a zero-day vulnerability. A zero-day vulnerability is a vulnerability that at the moment there is no documentation of its existence.

For this reason, companies rely heavily on software, made by people who specialize in vulnerability detection, which analyses their products and automatically produces reports with all the vulnerabilities detected. As the projects that will be analysed can have many LOC, it is important that this analysis is scalable, to be more profitable to use a code analysis tool than using one or more staff to analyse the source code manually.

1.2 SAST

SAST (standing for 'Static Application Security Testing') is a type of tool that tests the security of an application. Rather than running the application under test, it analyses the source code (or binary) and produces a report of what it regards as a security vulnerability. Because it is not a specific tool, it can be implemented as an academic or commercial tool. For example, a debugger is a type of tool that helps programmers to catch malfunctions of their programs. In this dissertation, vulnerabilities will be addressed as paths in the source code that data go through, starting from something that the application does not trust (for example, unfiltered user input) and ending in a critical part (for example, running a command in the system's server) of the application, which can lead to a vulnerability. The reference [Chr20] shows some open source SAST tools that exist and some interesting statistics on vulnerabilities.

SAST scans are based on a set of predetermined rules that define the coding errors in the source code that need to be addressed and assessed. SAST scans can be designed to identify some of the most common security vulnerabilities out there, like SQL injection, input validation, some stack buffer overflows and more. But it is not complete, because of the limited way that it analyses the code (statically) it has its limits on what it can detect.

The SAST approach has prevalence of false positives, which are supposed vulnerabilities that the SAST catches, but looking at it more carefully, it can be seen that it is not a vulnerability, just it "seems" like a vulnerability to the SAST tool. False positives entail a tremendous waste of time, as teams need to check and validate manually each individual security bug candidate to make sure it is not a false positive. This problem is not influenced by the changes on this dissertation.

The security analysis of complex applications with a SAST tool requires many hours of scanning. As a result, sometimes teams schedule the scans to run offline (such as during nights). This practice breaks the agile software development values. This dissertation wants to make this problem disappear by lowering the complexity of a SAST program, by analysing its components and proposing a compelling optimization, by changing the current strategy of storing and identifying a vulnerability path.

Checkmarx Companies like Checkmarx are needed, due to the high demand for safe and more secure applications. IT companies are coming out of thin air, to help with this we need to make the process a little more manageable than requiring to know all the most known vulnerabilities and how to correct them. That is why there are companies like Checkmarx, with specialized people to work on their products, to make them more complete and correct, so that the customer does not need to meet all the requirements to make a secure application.

As mentioned earlier, due to the more popular methods of developing software, and due to the big quantity of new companies that create software, so there are a lot of LOC that needs to be analysed.

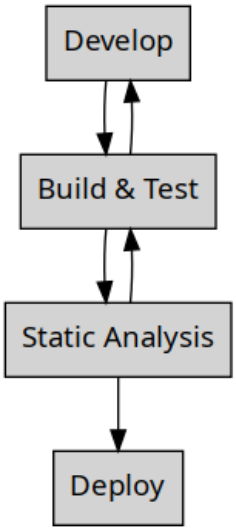


Figure 1.1. A simple pipeline to explain where in the software development process SAST is used

Therefore, needs to exist a quick way to analyse all of those LOC.

Checkmarx is aware of their tools being near and near to the limit of being scalable by today's standards. This is why they want solutions to help them in that regard, e.g. designing more efficient tools in detecting vulnerabilities. Such is the focus of this dissertation.

By examining SAST tools already available and seeing how they manage to implement the critical part, the analysis of the data flow graph, one can get an idea of what has been done in this regard.

Scalability Today, the volume of code that needs to be analysed is enormous. For example, big companies like Facebook have found vulnerabilities in source code that had hundreds of millions of lines of code [DFLO19]. At this scale it starts to be infeasible to have humans analysing the source code to find vulnerabilities, that is why the scanning with a static analysis tool is so important nowadays, especially if the pattern of the vulnerability is simple to detect.

The preferred software development model of companies nowadays is the incremental approach, e.g. the agile approach [OMGSC18], meaning that the source code keeps getting bigger as time goes by, and its analysis keeps taking longer to finish. This is why the analysis needs to be scalable, meaning analysis time being proportional to source code change. If this is not the case, then the time needed to analyse the source code will take an unrealistic time to finish and having human security analysts would be more reasonable.

1.3 DATA FLOW GRAPH

Dataflow is a way to describe a process by abstracting programs into computational actors and linking them according to the flow of data among them, so that actors can execute in parallel. This description has some good properties, like emphasizing the flow of data and being inherently parallel, therefore working well in large decentralized systems [Rum77].

Representing a program by a dataflow graph can be very helpful for the static analysis of source code [KSS17] by e.g. finding bugs, finding simple errors or (in the interest of this dissertation) finding vulnerabilities in source code.

The DFG shown in figure 1.2 represents a simple arithmetic expression, $(1 + 2) * 3$.

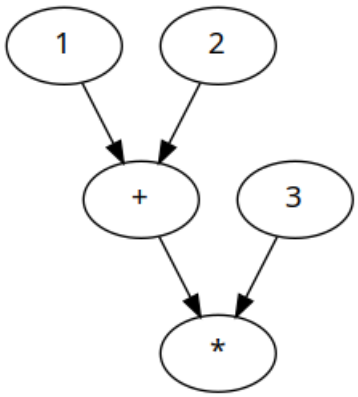


Figure 1.2. DFG of $(1 + 2) * 3$

From representing arithmetic expressions, DFGs scale up to doing the same for programs written in some programming language (then with some more primitives, i.e. literals, variables, operators, function calls and so on). To guarantee that one such DFG is a finite graph, it needs to have labels on the edges that identify if it enters or exits a function body.

1.4 GRAPH ALGORITHMS

An algorithm to find all the nodes between two given sets of nodes of a graph can be seen as the search starting from the first set of nodes to the second set of nodes. Such a task can be achieved by any graph search algorithm. For example, DFS (depth-first search) and BFS (breadth-first search) [CLRS09] work fine, because the order of the nodes does not matter to these algorithms. The implementation of DFS is simpler, and the property that finds one path very quickly is an advantage for a future idea, so it is better to use it instead of BFS.

The book [CLRS09] in chapter 22, section 3 describes the DFS algorithm as follows. "*A depth-first search strategy is, as its name implies, to search "deeper" in the graph wherever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored out-going edges. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until it discovers all the vertices that are reachable from the original source vertex. Should any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex*".

In this case, it does not need every detail in the [CLRS09], because the situation is more restricted — needs to have a set of source nodes, so it does not need to use any undiscovered vertices that are not in the source nodes. Also, it does not need to track the previous node of the search, it does not need the time and the colour can be abstracted, as if the node is visited or not.

The pseudo algorithm then is as follows:

```
DFS(G, sources):
    for each s in sources:
        if s is not visited:
            DFS-VISIT(G, s)

DFS-VISIT(G, s):
    send s
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-VISIT(G, w)
```

The image in figure 1.3 visually explains how the DFS works, by presenting a graph with the nodes being represented as a number which tells the order that each node was visited. The sources on this image are the two top nodes.

1.5 DISSERTATION STRUCTURE

This dissertation contains seven chapters, whose summary is presented below, excluding the introduction:

- State of the art: Presents tools named CodeQL and Pysa, which can give us a better overview of what tools exist that deal with the problem of detecting vulnerabilities.

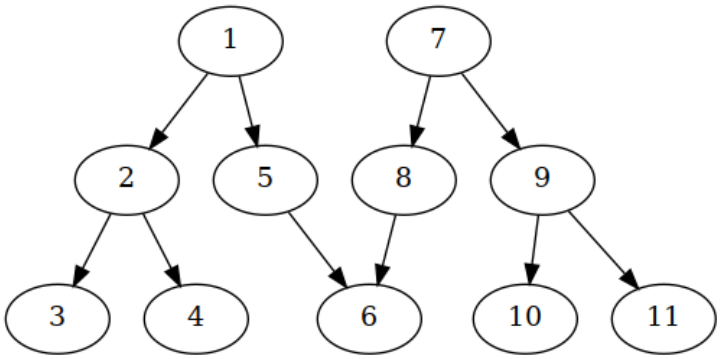


Figure 1.3. Image explaining the DFS, by giving the order via the number in the nodes

- Explanation of SAST: Explains how a SAST tool can work with a DFG as an auxiliary tool. With an example, follows all the process from source code and query to giving the malicious paths that make the source code vulnerable.
- Problem and Proposed Solution: Presents the problem to tackle and a solution to that problem, by modifying the return structure.
- Implementation: Implements a prototype SAST tool that will be used to show the benchmarks of the techniques presented.
- Benchmarking: Shows results from the past chapter, comparing the difference in time between the techniques presented.
- Conclusion: Concludes the dissertation, presenting an overview of the results obtained. Also, some guidelines for future work are suggested.

2. STATE OF THE ART

2.1 CODEQL

This section will address CodeQL [Git21], a query language for finding patterns in source code. Even though CodeQL is not a SAST tool, the part of the SAST that will be at focus later on relies on identifying patterns in source code, therefore the important part is basically how the query language functions.

The CodeQL language is very similar to the SQL language, but instead of searching relations between data, they search relation between elements of the source code. It is a logic programming language, so it is built up of logical formulas. CodeQL uses common logical connectives (such as 'and', 'or' and 'not'), quantifiers (such as 'forall' and 'exists'), and other important logical concepts such as predicates.

CodeQL also supports recursion and aggregates. This allows one to write complex recursive queries using simple CodeQL syntax and directly use aggregates such as count, sum, and average.

CodeQL queries typically look like this:

```
import <language>

from /* ... variable declarations ... */
where /* ... logical formulas ... */
select /* ... expressions ... */
```

For example, the outcome of the following query is the listing of all functions that have more than 7 arguments:

```
import python

from Function f
where count(f.getAnArg()) > 7
select f
```

The *from* clause defines a variable *f* representing a Python function. The *where* clause limits searching to functions *f* with more than 7 arguments. Finally, the *select* clause lists these functions.

CodeQL is versatile and clearly a valuable tool in code analysis, what matters to this dissertation is how efficient DFG extraction is in CodeQL and, if not efficient, what problems there are in their approach in this respect.

A DFG in CodeQL is computed using classes to model the program elements that represent the graph's nodes. The flow of data among nodes is modelled using predicates to compute the graph's edges. This by itself tells that the main focus is on correctness, and therefore it is not surprising that DFGs can be very large and slow to compute.

To overcome these problems, the strategy of CodeQL is to divide DFGs into two kinds:

- Local data flow, concerning the data flow within a single function. When reasoning about local data flow, one only considers edges between data flow nodes belonging to the same function. It is generally sufficiently fast, efficient, and precise for many queries, and it is usually possible to compute the local data flow for all functions in a CodeQL database.

- Global data flow, effectively considers the data flow within an entire program, by calculating data flow between functions and through object properties. Computing global data flow is typically more time and energy intensive than local data flow, therefore queries should be refined to look for more specific sources and sinks.

2.2 PYSA

This section will address Pysa (Python Static Analyzer) [Met22], an open source static analysis tool to detect and prevent security issues in Python code. Pysa is a feature of Pyre, which is an application that type checks python code. This static analyser uses Taint Analysis, to identify potential security issues.

Tainted data is data that cannot be trusted in any critical function. Pysa uses the flow of data from where the possible vulnerability begins (sources) to where it terminates in a critical location (sinks). This analysis uses user-created specifications, which provide rules that define what danger specific sources bring, and what danger specific sinks cannot accept to be tainted by.

```
x = some_function_that_returns_a_tainted_name()
f = f"Hi {x}!"
```

In the code above, x will be tainted with a certain annotation, by the result of the function $some\_function\_that\_returns\_a\_tainted\_name$, because of that, f will get marked as the same taint x had.

2.3 DATAFLOW ANALYSIS

Introduction Dataflow analysis is relevant for this work because it can be used to calculate a DFG of a source code with a CFG (Control Flow Graph) of that same source code. (A CFG is a graph of basic blocks of the program, where the basic blocks are in a "straight line" code sequence with no branches.) This process is used in some tools to get the DFG, even though this work did not use this process to generate the DFG.

Dataflow analysis is a way to obtain the possible set of values calculated at specific points in the source code that is being analysed. The source code's CFG is used to know how particular assignments propagate.

For example, in an **if** statement, the assignment before propagates its change to the **then** part and the **else** part. The code in a CFG basic block has:

1. One entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.

2. One exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

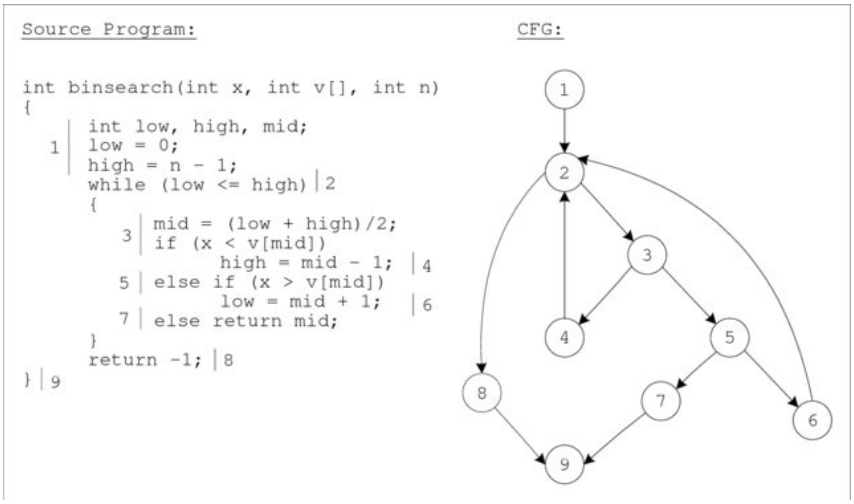


Figure 2.1. A CFG example.[AEK04]

Because of this, whenever the first instruction in a basic block is executed, the rest of the statements are necessarily executed exactly once, in order [Coc70]. Classical papers that introduced this way of analysis include [AC76] and [RHS95].

Dataflow obtains particular information at each point in a procedure. It is enough to obtain this information at the boundaries of basic blocks, since it is easy to calculate the information at points in the basic block. In forward flow analysis, the block can be seen as a function of states. This function is the composition of the effects of the statements in the block. The entry state of a block is a function, called **join**, of the exit states of its predecessors. This gives a set of dataflow equations, for each block b,

$$out_b = trans_b(in_b)$$

$$in_b = join_{p \in pred(b)}(out_b)$$

where $trans_b$ is the transfer function of the block b. It works on the entry state in_b, giving the exit state out_b. The join operation $join$ combines the exit states of the predecessors $p \in pred_b$ of b, giving the entry state of b.

After solving these equations by fixpoint reasoning, the entry and exit states of the blocks can be used to derive properties of the program at the entry and exist of a block. The transfer function, is basically the specific behaviour of each statement separately, and can be composed to get information at a point inside a basic block.

Each particular type of dataflow analysis has a specific transfer function, for each statement, and join operation. Normally the states are a set of values and the join operation is a simple union or an intersection.

2.3.1 Example

A simple example of a dataflow analysis is *reaching definitions*, which is explained below using an example.

Reaching definitions is the problem of which definitions are "active" in a given definition. Assume, for example, the following code, whose structure is a sequence of an identifier followed by a definition, separated by a colon. A definition is an attribution of an expression to a variable.

```
d1 : a := 0
d2 : b := a
```

Clearly, d1 reaches d2. However, in

```
d1 : a := 0
d2 : a := 1
d3 : b := a
```

Here, d1 does not reach d3, because d2 kills the value defined in d1, no longer reaching d3, but d2 reaches d3.

The CFG of the latter piece of code is given by a directed graph between, in this case, the definitions. The edges point to the next node to be executed in the program.

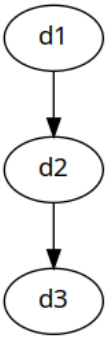


Figure 2.2. The CFG of the simple example above.

Let $DEFS[y]$ be the set of all definitions that assign to some given variable y. The dataflow equations used for a given basic block S in reaching definitions are:

$$REACH_{in}[S] \;=\; \bigcup_{p \in pred[S]} REACH_{out}[p]$$

$$REACH_{out}[S] \;=\; GEN[S] \cup (REACH_{in}[S] - KILL[S])$$

$$GEN[d : y := f(x1, \ldots, xn)] \;=\; \{d\}$$

$$KILL[d : y := f(x1, \ldots, xn)] \;=\; DEFS[y] - \{d\}$$

Because of the simple nature of dataflow equations and the consensus of CFGs, dataflow analysis is very appealing for static analysis.

In the small example above, dataflow analysis can be done by, first calculate the GEN and $KILL$ of each node, and the $DEFS$ of each variable.

	$DEFS$
a	$\{d1, d2\}$
b	$\{d3\}$

	GEN	$KILL$
d1	$\{d1\}$	$\{d2\}$
d2	$\{d2\}$	$\{d1\}$
d3	$\{d3\}$	$\emptyset$

The initial values of $REACH_{in}$ and $REACH_{out}$ are all $\emptyset$.

	first iteration		second iteration		third iteration	
	$REACH_{out}$	$REACH_{in}$	$REACH_{out}$	$REACH_{in}$	$REACH_{out}$	$REACH_{in}$
d1	$\{d1\}$	$\emptyset$	$\{d1\}$	$\emptyset$	$\{d1\}$	$\emptyset$
d2	$\{d2\}$	$\{d1\}$	$\{d2\}$	$\{d1\}$	$\{d2\}$	$\{d1\}$
d3	$\{d3\}$	$\{d2\}$	$\{d2, d3\}$	$\{d2\}$	$\{d2, d3\}$	$\{d2\}$

The last 2 iterations are the same, which means the fixed point was reached on this algorithm, and that means it has an answer to the problem of reaching definitions. Every definition reaches to itself and $d2$ reaches $d3$, which is the expected solution.

2.4 SUMMARY

In this chapter, is explored a tool and a technique widely used to catch vulnerabilities in source code, CodeQL. This was introduced to the reader with examples, presented its advantages and how the tool bypasses its disadvantages. Then Dataflow Analysis was presented to the user via the theory followed by an example.

This will be followed, in the next chapter, by a study of SAST approaches.

3. EXPLANATION OF SAST

3.1 INTRODUCTION

SAST tools perform, as the acronym tells, static application security testing, i.e. they are designed for analysing application source code (byte code and binaries) for coding and design conditions that are indicative of security vulnerabilities. The static part of the name refers to the analysis of the application at compile-time.

The main research problem addressed in this dissertation is the "scalable detection of security-vulnerabilities in source code". Because security vulnerabilities are so dependable on the system's security model, one cannot really perfectly formalize any arbitrary vulnerability. Checkmarx has developed a tool that, based on a query language, describes the vulnerability, which lets one find a vulnerability with some information extracted from the source code. Although not perfect, it does catch some patterns. One of the most important information and unscalable, is all the influence paths in the DFG, between two groups of nodes, the sources (where it "starts") and the sinks (where it "ends"). That information is asked by queries, with different sources and sinks.

From the previous paragraph, we can assume that finding influence paths is a way to describe a given vulnerability. So the problem can be rephrased to "scalable detection of influence paths in the data flow graph of the source code".

"Scalable" means "ability to scale up" in size without degrading the execution time, and because the structure of the source code is growing at a fast rate we can say that a linear complexity, in the quantity of edges in the DFG of the source code is "Scalable". So we can again rephrase the research question above to "linear detection of influence paths in the data flow graph of the source code".

The following is an example that will help the reader to understand the problem, get some hints for the solution and show all its most important components.

3.2 EXAMPLE

Consider the following definition of a known vulnerability, "Deserialization of untrusted data":

> *Data which is untrusted cannot be trusted to be well formed. Malformed data or unexpected data could be used to abuse application logic, deny service, or execute arbitrary code, when deserialized. [Fou22]*

In the listing 3.1 we present an example of a source code. In this context, it is irrelevant how the implementation of an integer parser is done, or even the memory allocation of a matrix. When in the source code is commented **pseudo**, then it means that the implementation is not really important, and

can be seen as a direct influence from the inputs to the outputs, we can do it by using operators that "join" 2 variables, like "+". That influence is needed for the continuity and correctness of the DFG construction, because without it, we would not be able to connect the variables that are influencing each other.

```
class Main {
    public int parseInt(String i){
        //pseudo
        return i;
    }

    public int[][] initMatrix(int r, int c){
        //pseudo
        return r + c;
    }

    public static void main(String[] args){
        int rows = parseInt(args[0]);
        int columns = parseInt(args[1]);
        int[][] matrix = initMatrix(rows, columns);
        //...
    }
}
```

Listing 3.1. Semi-Code excerpt suffering from the "deserialization of untrusted data" vulnerability.

In the example of the code excerpt 3.1, the input could be a negative number and the size of the matrix makes no sense, or could be an enormous number. Both cases can be problematic, the first would stop the program, and the host could be a server, the second could possibly cause a DOS attack, by giving a lot of space in the matrix, then some operation acting on it would delay or even deny the service.

All this may happen because the data from the input are not trusted and there is no sanitization of the input. In this example, a possible sanitization could be to set some boundaries on the number given by the input.

The user input can be seen as untrusted data and the $initMatrix$ as a potentially vulnerable function with this consideration it can be seen that the code 3.1 is affected by the vulnerability "Deserialization of untrusted data",

3.3 DATA FLOW GRAPH

A DFG is a graph extracted from a source code where the nodes can be function calls, variables, constants and operators, the edges denote a dependency between nodes, but to keep the graph finite we need annotations on some edges to identify from which call we enter and exit a function definition.

In the representations of the DFG, the vertices are denoted by i_n, where the i is the name of the function calls, variables, constants, or operators mentioned in the source code, and the n is the identifier of the associated vertex.

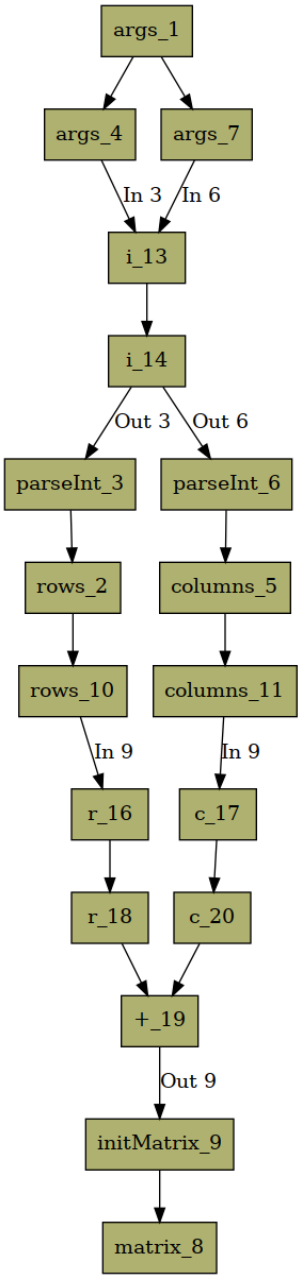


Figure 3.1. DFG of listing 3.1

In figure 3.1 one can see that the input **args** is influencing the **matrix** and its initialization, **initMatrix**.

3.4 ATOMIC QUERIES

The atomic queries are the queries we can make to the DFG and the other extra information, but because we are targeting only the DFG, then that is all we need.

The most important atomic query we can make is "what are the paths from the vertices of the set A and the vertices of the set B". Assuming DFG, $Vertex$ and $Path$ suitably defined, let us call this $InfluencingOn : DFG \times \mathcal{P}(Vertex) \times \mathcal{P}(Vertex) \to \mathcal{P}(Path)$.

There exists other Atomic Queries, like $InfluencedBy$ and $InfluencingOnAndNotSanitized$, but we can argue that $InfluencedBy$ is just the $InfluencingOn$, but with the sources and sinks swapped. And $InfluencingOnAndNotSanitized$ that receives sources, sinks and sanitizers, is just the "composition" of the paths from $InfluencingOn$ of sources and not sanitizers, and $InfluencingOn$ of not sanitizers and sinks. It is convenient to talk about sanitizers when analysing a DFG, sanitizers are vertices that if they appear in the middle of the path, means that the path is not a vulnerability, for that reason the $InfluencingOnAndNotSanitized$ atomic query is also very important, but as we argued above is nothing that we cannot do with just $InfluencingOn$.

If we continue to examine each of the "Atomic Query", it can be seen that the most important "Atomic Query" is the $InfluencingOn$. Because of that, we will focus only on this "Atomic Query", and how to make it scalable.

Let us call the DFG in figure 3.1 $DFGDes$, then

$$InfluencingOn(DFGDes, \{1, 4, 7\}, \{8\}) = \{[1, 4, 13, 14, 3, 2, 10, 16, 18, 19, 9, 8],$$
$$[4, 13, 14, 3, 2, 10, 16, 18, 19, 9, 8], [7, 13, 14, 6, 5, 11, 17, 20, 19, 9, 8]\}$$

The figure 3.2 is the visual representation of the result of the "Atomic Query" above.

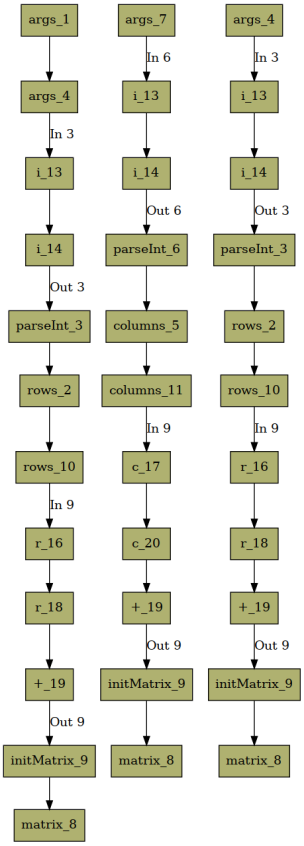


Figure 3.2. Influenced paths of 3.1 using the query that starts at 1, 4 and 7 and ends at 8.

3.5 QUERIES

The query is a vulnerability description, that SAST can search with. The queries do not change the complexity, just when calling the atomic queries, because the complexity of the query, without the atomic queries, does not depend on the source code, but only on the intricacy of the vulnerability itself.

The queries already have the DFG, the other extra information and all the atomic queries available. So we can see a query as a function that receives the DFG, extra information and the atomic queries, and returns a set of paths. The inputs of the query can be accessed by calling by its name, for example if we want the DFG, we can just call **DFG**, which is a variable with the DFG stored in it.

The description of the queries are given with an imperative language, that is not very important in our context.

With figure 3.1 we can detect a "deserialization of untrusted data", by admitting that the input is not a trusted source, and the function $initMatrix$ is a vulnerable function, that requires the inputs to be positive integers and not very large, depending on the machine we are working with. Because exists a path from $args$ (untrusted source) to $initMatrix$ (vulnerable function).

A simple query that can catch this vulnerability, in this example, can be like:

```
var inputs = Find(DFG, "args");
var vulnerable = Find(DFG, "initMatrix");

return InfluencingOn(DFG, inputs, vulnerable);
```

Where $Find : DFG \times String \to \mathcal{P}(\mathbb{Z})$ is the function that returns the keys of the nodes that have the exact name as the input.

3.6 SEARCH

We talked about what the atomic queries are, but here we are going to talk about how do they search the DFG.

The search consists of getting all the shortest unique paths, from the set of vertices A to B. Unique means that we cannot have 2 paths that start and end at the same vertices, for example, the path $[1, 2, 4]$ starts and ends in the same vertices as $[1, 3, 4]$. Only caring about the unique paths causes ambiguity in what path to choose, but when it is possible to have a cycle in the graph, that means we only care about finitely many paths and not infinitely many paths.

The algorithm is a special graph search algorithm for this type of graph, because when it arrives at a

node, it checks if the In and Out labels of the edges, are correctly sequenced, if so continue the process, if not backtrack the path.

Then, when it finds a path that the first vertex is in A and the last is in B, then it needs to verify that this path found is unique in the current state of the return set, if so, adds to the set, if not, continues the search.

3.7 SUMMARY

In this chapter, the main concern was to give a brief explanation of how a SAST based on the source code's DFG works and its problems in finding a malicious path in the source code.

Started by presenting a semi source code example, of a "deserialization of untrusted data", to guide the explanation with something tangible, followed by the DFG of that same semi source code.

Then the explanation of how the atomic queries and the queries work, with the assistance of the example presented in the beginning of the chapter.

After that we talked about the inner works of the atomic queries, by explaining how it can search the DFG, for paths. With that, we can now understand the problem in hand.

4. PROBLEM AND PROPOSED SOLUTION

This chapter addresses the subtle problem which the search approach presented in the previous chapter suffers from, and discusses the minimum time complexity such search algorithm can exhibit. It also discusses an even worse problem, should the implementation follow a particular strategy, to be explained.

The solution proposed in this dissertation will be explained in detail. It is based on the idea of creating a subgraph of the DFG, henceforth referred to as *the sub-DFG*. Great benefit in time complexity will come from the use of this sub-DFG. We will stress on the improvement of the process as a whole, by describing this as a pipeline. This is followed by determining the time complexity of the proposed solution and discussing its main advantages and disadvantages.

4.1 THE PROBLEM

Let us look at the complexity of searching the paths between two sets of vertices.

The result of the search in the previous chapter is a set of paths of the DFG. With this in mind, we can see the following case scenario, when we have n source nodes linked to one node, and that same node linked to another n sink nodes. This makes the total number of edges equal to $2 * n$. Since the number of sources and the number of sinks are both n, the number of paths is $n * n$. Because each path has 2 edges, the total number of edges in the return value of the search is $2 * n^2$, which is equal to $e^2/2$, where e is the number of edges in the DFG. Therefore, the complexity of any algorithm that makes that searches for all the possible paths has to have a time complexity $\geq \mathcal{O}(e^2)$.

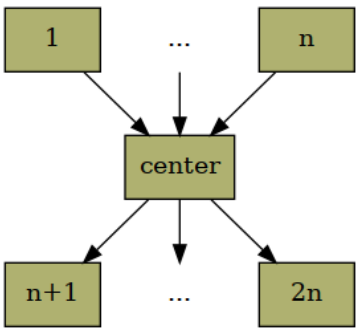


Figure 4.1. The case where it must return $\mathcal{O}(e^2)$ vulnerable paths.

Actually, the time complexity is even worse when we look closely at an actual implementation that searches the DFG for every possible path: we need to take into consideration the time it takes to reject the paths that are already chosen. We can construct the following case scenario with only 1 source and sink node, where the source node is linked to two other nodes, and those two nodes linked to yet another node, repeating this process k times, with the last node being the sink node. In this scenario, the number of possible paths be 2^k. Since this graph has 4 edges by each k repetition, the number of possible paths

is $2^{e/4}$, where e is the number of edges. With this type of implementation, we have a time complexity $\geq \mathcal{O}(2^{e/4})$.

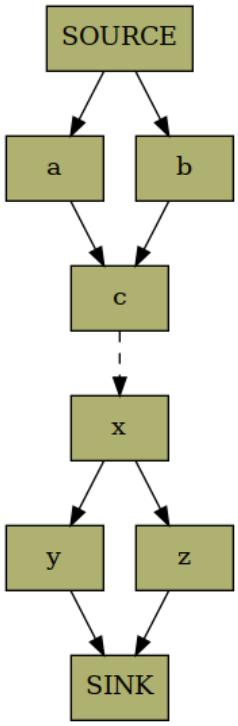


Figure 4.2. The case where we need time complexity $\mathcal{O}(2^{e/4})$ for a specific algorithm that searches for paths.

4.2 SUGGESTED SOLUTION

When this problem was presented, a possible solution was also presented. Here we will see what was the idea and why this is not a solution to this problem.

4.2.1 Idea

In the DFG of the example, at figure 3.1, we can see that the 2 paths have $[13, 14]$ in common, because in both paths there is a call to the same function, $parseInt$. There is also $[19, 9, 8]$ in common, because the junction caused by the sum in the pseudocode of $initMatrix$. In this example, there is only a simple function, but in an industrial scale we can see lots of functions reused everywhere. If there was a way to have that already calculated, and give the result when needed, it would speed up the process.

Because, if the path in the function $parseInt$ was already calculated, then, instead of calculating for both of the paths, we would calculate only on one of them and the other would receive the result of the calculation for the other path, saving, in this case, one search of the function $parseInt$.

The function calls were the more critical part, because it is very common the re-usability of functions already written and tested.

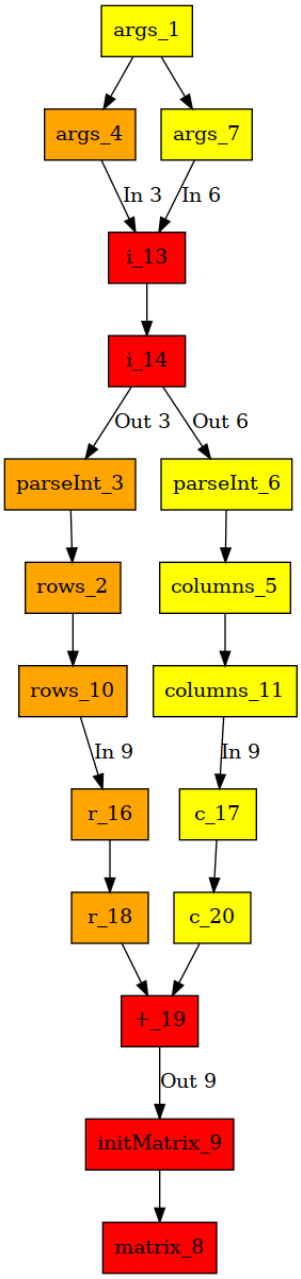


Figure 4.3. Vulnerable paths of 3.2 as a heat map of node encountering, where yellow means 1, orange 2 and red means 3 encounters.

To implement this idea, we would need to have for each function an already calculated set of paths from each formal parameter to each return node. Analysing the function only one time, resulting in the improvement the idea talked about. At least in the path from inside the function, which is the critical part. The case of junction of the binary operators would not be treated, because it is not a problem that causes big concern, statistically speaking.

For example, the min function implemented as such:

```
public int min(int a, int b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}
```

The min function will have a DFG as such:

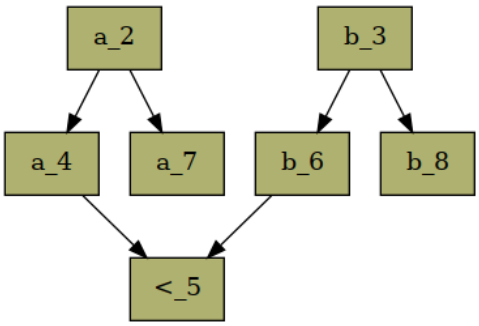


Figure 4.4. DFG of the min function implemented above.

So the function min after being analysed should have something like this:

- For the case of entering from the input 2 and exiting from 7 should give $\{[2, 7]\}$.
- For the case of entering from the input 2 and exiting from 8 should give $\{\}$.
- For the case of entering from the input 3 and exiting from 7 should give $\{\}$.
- For the case of entering from the input 3 and exiting from 8 should give$\{[3, 8]\}$.

Why it is not a Solution Even when all the function paths are calculated, the process still has a time complexity $\geq \mathcal{O}(e^2)$, because of the quantity of paths in the return set as previously explained in section 4.1.

The idea would only help in creating the paths, but the quantity of paths would continue the same and as before unscalable, which means this is optimizing the process, but the complexity maintains. That means this is not a solution for the problem we want to solve.

4.2.2 What a Solution Should Do

The solution has to treat the problem of having a large quantity of elements in the return set, but without reducing the quantity of elements, because every element is important information, a potential vulnerability in this case.

That can be achieved by having a more condensed structure, instead of a simple set. That structure has to be able to access the individual paths, and remove them, like what a set is capable of doing.

Going a step back and reflecting on the purpose of the In and Out edges, they exist to be feasible to represent the DFG of different calls of the same function, and to not repeatedly represent the same function. This solution solves the same problem that we have now, but instead of the dependencies in the function, we have the function paths.

4.3 THE SOLUTION

Instead of having a set of paths, a path for each vulnerability candidate, we can have a subgraph of the DFG to encapsulate all of those paths, and only those paths. From now on, let us call that subgraph of the DFG by the name sub-DFG.

The process is the same as before until the search of the atomic query, which, instead of searching every path and removing the duplicated instances, will postpone the path search by first calculating the sub-DFG of the given DFG, with the sources and sinks the atomic query receives from the query.

The figure 4.5 shows how we described the SAST beforehand, getting the DFG by the source code, which is a much more complicated process than the arrow suggests, after the DFG is calculated, the queries find the sources, sinks and so on for the specific vulnerability, to call the atomic queries which finds the vulnerable paths, after that the queries can also modify the set of paths, like remove paths, add paths and so on.

The figure 4.6 shows basically the same as the figure 4.5, except the part where the atomic queries were calculating the vulnerable paths. Now they are calculating the sub-DFG and storing it like before we were storing the vulnerable paths. Then, if the user wants to see a vulnerable path, he can use the sub-DFGs to ask for it and receive it.

With the sub-DFGs calculated, we can also have a different way of visualizing the vulnerabilities without

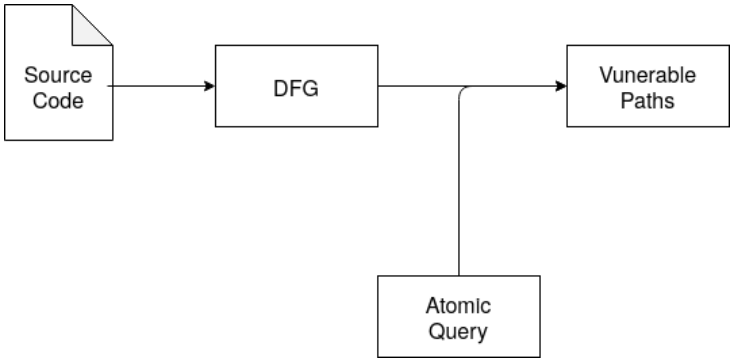


Figure 4.5. pipeline of the simple design

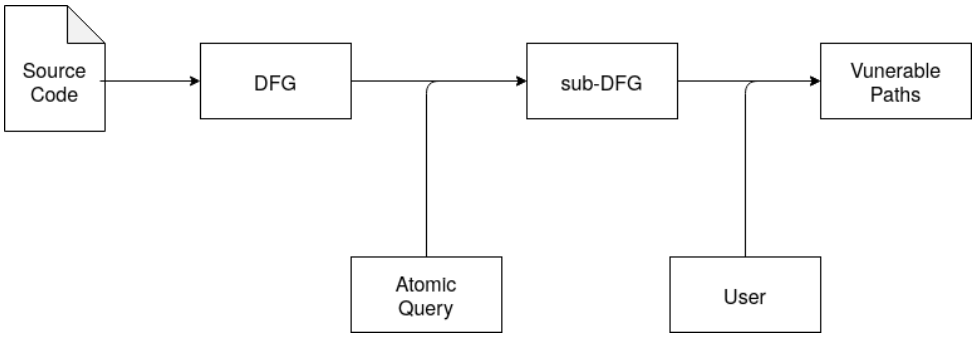


Figure 4.6. changed pipeline for the proposed solution

having to search for vulnerable paths. This makes the sub-DFG a better alternative than simply making a by-demand search in the DFG.

4.4 HOW TO GET THE SUB-DFG

With the source nodes and sink nodes given by the query to the atomic query, the sub-DFG must have all the paths that go from source nodes to the sink nodes. The $dfs(DFG, sources)$ represents all the nodes that are influenced by $sources$. The $rdfs(DFG, sinks)$ represents all the nodes that are influencing the $sinks$. So the sub-DFG can be calculated by filtering the DFG by the intersection of $dfs(DFG, sources)$ and $rdfs(DFG, sinks)$, hence the following equality:

$$subDFG = subgraph(DFG, dfs(DFG, sources) \cap rdfs(DFG, sinks))$$

We are going to explore how to calculate the sub-DFG, of figure 3.1, named $DFGDes$, with the atomic query $InfluencingOn(DFGDes, \{4\}, \{8\})$.

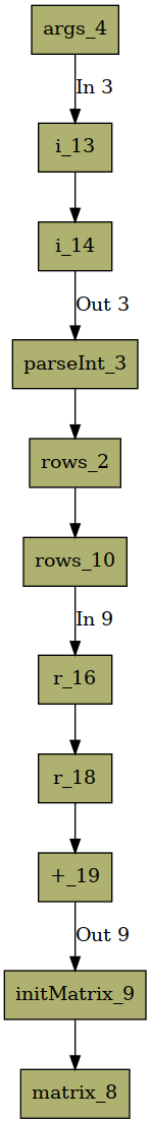


Figure 4.7. sub-DFG of 3.1 using the Atomic Query $InfluencingOn(DFGDes, \{4\}, \{8\})$

The sub-DFG has also the information of the sources and sinks within the structure, but because we are more interested in the graph part than this embedded information, for the next sections and chapters we are going to omit this information and refer to the sub-DFG as only the part of the graph. Specially when talking about the search process where we need the information of the sources and sinks to start and end the search.

4.5 HOW TO SEARCH A PATH IN THE SUB-DFG

To find a path in the sub-DFG we cannot simply use a normal DFS, because we need to have in consideration the call edges we already entered and exited, so we need to use a search graph algorithm with a stack to capture what call we entered, when exiting the procedure we remove it from the stack. We can ensure that we end up in a defined sink, because we constructed the sub-DFG, so that every node influences a sink.

A valid path can be informally defined as a path that can be traversed by an evaluation of the source program. When we consider a language which does not have procedure calls, the path could be any path on the DFG, but procedure calls are present in the languages we want to discuss. To define what a valid path is, we can see how the paper [RHS95] in definition 2.3 has done it, we can ignore every normal transition in a path, i.e. we can validate a path only by what procedures it enters and exits. Let us represent an edge that enters a procedure call i by "$(_i$" and an edge that exits a procedure call i by "$)_i$". With this, we can say that a path is valid if it can be built by the following rules, for $i \in Call$:

$$
\begin{array}{lcl}
matched & \rightarrow & (_i \; matched \;)_i \; matched \\
 & | & \varepsilon \\
validInside & \rightarrow & validInside \; (_i \; matched \\
 & | & matched \\
validOutside & \rightarrow & matched \;)_i \; validOutside \\
 & | & matched \\
valid & \rightarrow & validOutside \; validInside
\end{array}
$$

The definition 2.3 in the paper [RHS95] shows that the valid is our $validInside$, because the language the paper was presenting has only procedures that cannot modify the state outside of it. Because of that, in the paper, it did not make sense to validate a path that says that a local variable is influencing a global one. But, for us, it is possible that a procedure could modify the global state, by for example reading an input and assigning a value to a global variable. Because of that, we present a $validOutside$ which says that a path can be valid if it starts in a procedure and ends outside that procedure.

The search part of the process only runs after the sub-DFG of all queries are calculated, so this process of calculating paths is only for the evaluation of the vulnerabilities found and wanted to be checked. Because of this, it is a very light calculation.

Below, we can see a pseudocode of how to catch flows or paths of a DFG or sub-DFG. In our case, we need this part to execute on demand by the user, so that it executes only when the user wants a path, or some paths.

G stands for graph, *sc* stands for source, *sks* stands for sinks, *callS* stands for call stack and *fl* stands for flow.

```
flows(G, sc, sks):
    flowsAux(G, sc, sks, [], [sc])

flowsAux(G, sc, sks, callS, fl):
    mark sc as visited
    for all neighbours w of sc in G:
        e := edge(sc, w)
        if w is not visited and e is valid in callS:
            callS' := update callS with e
            fl' := insert w in fl
            futureflows := flowsAux(G, w, sks, callS', fl')
            if w is in sks:
                fl', futureflows
            else:
                futureflows
```

Below we can see 2 functions, one to validate if the edge makes sense given a call stack, the other to update a call stack given a valid edge.

```
valid(e, callS):
    if callS is empty:
        e is not (Out i)
    lc := last call of callS
    if e is (Out i) and lc != i:
        false
    else:
        true

update(callS, e):
    if callS is empty:
        if e is (In i):
            insert i in callS
        else:
```

```
            callS
        lc := last call of callS
        if e is (Out i) and lc = i:
            remove lc of callS
        if e is (In i):
            insert i in callS
        else:
            callS
```

4.6 COMPLEXITY

In chapter 4.1, we argued that the complexity of the atomic query to the vulnerable paths is $\geq \mathcal{O}(e^2)$, where e is the quantity of edges in the DFG. In the solution presented we can argue that the complexity of the atomic query to a sub-DFG, defined in 4.4 is $\mathcal{O}(e + n)$, where n is the quantity of the nodes in the DFG, because the algorithms for subgraph, intersection and DFS are all $\mathcal{O}(e + n)$.

If the user wants to store all the paths, it has all the problems we discussed earlier again. A simple alternative is to guarantee the user only uses a limited number of paths, but that is limiting the information given. So one strategy where we do not store all the paths, and we do not limit the information the user can have, is to give one vulnerable path at each time, when the user asks for the next we calculate and give that next vulnerable path. This strategy can give all the vulnerable path, storing only one path and the sub-DFG for each query. The time complexity until the first path is given, after the sub-DFG calculation, is $\mathcal{O}(e + n)$, which is the time complexity of a DFS.

4.7 ADVANTAGES AND DISADVANTAGES OF SUB-DFG

The paths returned from the simple search have too many repeating nodes and edges, but returning the sub-DFG removes the repeated ones, making a more advantageous solution by having a very efficient storing capability for the nodes. The waiting time can be reduced, by using the right strategy talked in this chapter. If we wanted to create an automatic analysis tool, that gives extra information, like where to make changes in the code, the changes would be minimal, we do not need to run it in each path, but in sub-DFG which represents all the vulnerable paths, making this extra information much more quick to calculate.

The disadvantages of using a sub-DFG is the migration, from the simple method to this method, which means not everything can be reused, special information from the paths need a rewrite, for example the quantity of vulnerable paths, some scores that we could have by analysing each path independently, etc. Even the API of the queries needs to be rewritten, for example the addition of paths, the removal of paths,

the concatenation of paths, etc. If we return lists, we can have simpler algorithms and simplify the next processes, so returning a graph makes the next process a lot difficult. With lists, we could know very easily how many vulnerabilities there were and also evaluate each path for a score of risk, but with a graph that part is not that simple.

Another disadvantage of a DFG or a sub-DFG is that it does not have a good property of a graph, one that we expect by default, which is that, if two nodes are adjacent, then it means the first is influencing the second. But in a DFG of a program that is not the case, because in searching a path, if we previously entered through a call of a procedure, then we have to exit through the same call, that is, if it is possible to exit to another call, we cannot, because we did not enter from it previously. So, to know if one node is influencing another, we need to know the history of the path we took to reach that node, so that we can verify if the path did not go through a call of a procedure to another.

For example, in the 3.1 we can see that the path $[7, 13, 14, 3]$ does not make sense, because in the context of 3.1 we are entering in $parseInt_6$, through the second formal parameter, and exiting from $parseInt_3$. This means that the connection between 14 to 3 depends on the history of the path, which is not a thing that happens in a normal graph.

4.8 SUMMARY

This chapter begins by introducing the subtle problem we are facing, that at first glance is hard to catch, following it up by a suggestion that could improve the time it takes to give the vulnerable paths to the user, here we also discuss that it could improve, but it would not make the solution a scalable one.

We talked about a solution that makes the process scalable, by introducing a sub-DFG which we proceed to elaborate on how to calculate it from the DFG and a set of sources and sink nodes, given from a query. After, we also talked on how to search a path on it.

Ending with a discussion on the time complexity and with a note on the advantages and disadvantages of this solution.

5. IMPLEMENTATION

5.1 INTRODUCTION

To show this solution is a better alternative to other solution, we first need an implementation of both in the same environment, so we can have a common ground. This environment is needed, to simulate the true product, without any obstruction unnecessary for the purpose of this dissertation, and also to show my version of the product without any problem.

This environment is going to be written in Haskell, because of its power to generalize and its vast available packages.

5.2 LANGUAGE

To make the construction clear, was chosen one language, even though this type of construction can be generalized to other languages.

To be even more clear, we are going to focus on only the important part of the language. So the language we are going to talk is made up called MiniLang and defined in figure 5.1 and represented as Haskell datatype in the code 5.1.

Syntax

(Programs)	P	::=	$(S, F_1 \cdots F_n, M_1 \cdots M_m)$	(the default class)
(Fields)	F	::=	$x := E$	(field assignment)
(procedure declarations)	M	::=	$f(x_1, \cdots, x_n) = S$	
(Statements)	S	::=	$S; S$	(seq. composition)
		|	$IF(E, S_1, S_2)$	(if then else)
		|	$W(E, S)$	(while)
		|	$S(E)$	(expression-to-stmt)
		|	$R(E)$	(return)
		|	$\star$	(empty, skip)
(Expressions)	E	::=	L	(literal)
		|	I	(identifier)
		|	$E_1\ op\ E_2$	(binary operation)
		|	$f(E_1, \cdots, E_n)$	(procedure call)
		|	$x := E$	(assignment)

$op \in \{+, -, *, \cdots\}$

Figure 5.1. Language syntax.

```
data P' f m fn i l op =
    P'
        { mainM :: m fn i l op
        , fields :: [f fn i l op]
        , methods :: [m fn i l op]
        }
    deriving (Eq, Ord, Read, Show)

data F' e fn i l op =
    F'
        { lhs :: i
        , rhs :: e fn i l op
        }
    deriving (Eq, Ord, Read, Show)

data M' s fn i l op =
    M'
        { methodName :: fn
        , methodParams :: [i]
        , methodBody :: s fn i l op
        }
    deriving (Eq, Ord, Read, Show)
```

```haskell
data S' e fn i l op
    = Comp (S' e fn i l op) (S' e fn i l op)
    | IF (e fn i l op) (S' e fn i l op) (S' e fn i l op)
    | W (e fn i l op) (S' e fn i l op)
    | S (e fn i l op)
    | R (e fn i l op)
    | NOP
    deriving (Eq, Ord, Read, Show)

data E fn i l op
    = Lit l
    | Ident i
    | BinOp (E fn i l op) op (E fn i l op)
    | Call fn [E fn i l op]
    | Assign i (E fn i l op)
    deriving (Eq, Ord, Read, Show)

type P = P' F M

type F = F' E

type M = M' S

type S = S' E
```

Listing 5.1. Language syntax in Haskell.

5.3 DFG

For the purpose of this dissertation, *DFG* is a directed, weighted graph $DFG = (VI, vl, EI, el)$ where

- $Op = \{+, -, *, \cdots\}$
- $VI \subseteq \mathbb{Z}$ (the nodes of the graph, vertex identifiers)
- $vl : VI \rightarrow Fn + I + L + Op$ (a vertex-labelling function)
- $EI \subseteq VI \times VI$ (the edges of the graph, edge identifiers)

- $el : EI \rightarrow VI + VI + 1$ (the arcs of the graph weighted by references to other vertices).

In **el**, the first **VI** is the identifier of the method call, which is the label of the edge, that goes from the expressions given to the method call, to the input variable declarations in the method declaration. The second **VI** is the identifier of the method call, which is the label of the edge, that goes from an expression vertex to the vertex of the method call that the **VI** represents. The **1** in the end is to represent a "normal" edge, which is an edge that does not have any information, only says the first influences the second. An example can be seen in 3.1.

The representation in Haskell datatype is in the listing 5.2.

```
type Key = Int

type Fun = Key

type Call = Key

data NodeL fn i l op
    = NodeLit l
    | NodeIdent i
    | NodeFN fn
    | NodeOp op
    deriving (Read, Eq, Ord)

data EdgeL
    = Normal
    | In Call
    | Out Call
    deriving (Read, Eq, Ord)

data Gr a b =
    Gr
        { vl :: Map Key a
        , el :: Map (Key, Key) b
        }

type DFG fn i l op = Gr (NodeL fn i l op) EdgeL
```

Listing 5.2. DFG in Haskell.

In the Haskell datatype we did not define the EI or the VI, because the Map is a partial function with a domain with the number of elements less than 2^{63} and the elements of the domain need to have

a total order, this restriction is not very worrying, so we can continue with it. That means that EI can be defined to be the domain of el and the VI as the domain of vl.

This can be proven to be a graph by defining Gr as an instance of the class $Graph$. Because Gr can also be altered, we can instantiate $DynGraph$ for convenience. Both of this can be seen in 5.3.

```
instance Graph Gr where
    empty = Gr M.empty M.empty
    isEmpty (Gr n e) = M.null n && M.null e
    match i gr@(Gr n e) =
        case n !? i of
            Nothing -> (Nothing, gr)
            Just nl -> (Just (lti, i, nl, lfi), Gr nni eni)
                where nni = M.delete i n
                      eni = M.filterWithKey (\(l, r) _ -> l /= i && r /= i)
                          e
                      lfi =
                          map (\((_, r), lbl) -> (lbl, r)) $
                          M.toList $ M.filterWithKey (\(l, _) _ -> l == i)
                              e
                      lti =
                          map (\((l, _), lbl) -> (lbl, l)) $
                          M.toList $ M.filterWithKey (\(_, r) _ -> r == i)
                              e
    mkGraph ln le =
        Gr
            { vl = M.fromList ln
            , el = M.fromList $ map (\(a, b, c) -> ((a, b), c)) le
            }
    labNodes (Gr n _) = M.toList n
    noNodes (Gr n _) = M.size n
    nodeRange (Gr n _) =
        let n' = map fst $ M.toList n
         in (minimum n', maximum n')
    labEdges (Gr _ e) = map (\((a, b), c) -> (a, b, c)) $ M.toList e

instance DynGraph Gr where
    (lti, i, il, lfi) & (Gr n e) = Gr n' e'
      where
        n' = M.insert i il n
        e' =
            mconcat
                [ M.fromList [((l, i), lbl) | (lbl, l) <- lti]
                , M.fromList [((i, r), lbl) | (lbl, r) <- lfi]
```

```
                , e
                ]
```

Listing 5.3. Instances of $Graph$ and $DynGraph$.

5.3.1 Union

A very useful operation to construct the DFG, is the union of graphs, to implement that we are going to create an instance of a Semigroup, which is possible, because the union operation is associative. Also, a Monoid for convenience, and because we have an obvious identity element, which is the graph with no vertices or edges. Both implementations are in 5.4

```
instance Semigroup (Gr a b) where
    (Gr v1 e1) <> (Gr v2 e2) = Gr (v1 <> v2) (e1 <> e2)

instance Monoid (Gr a b) where
    mempty = Gr mempty mempty
    mconcat = (\(vs, es) -> Gr (mconcat vs) (mconcat es)) . unzip . map
        from
      where
        from = \(Gr v e) -> (v, e)
```

Listing 5.4. Monoid DFG instance in Haskell.

5.3.2 Information on a Method Definition

Because we have function calls, we need certain information from the function definition, when we get to one function call. That is because if it exists a cycle in function calls, like a recursive function, then when a program is analysed, and we encounter a function call, if we also analyse the function definition, then we can have an infinite cycle and the analysis not ending.

To avoid the earlier behaviour, we can introduce some information already calculated before the analysis. For that, we will define a $MethodInfo$, defined in 5.5.

```
data MethodInfo fn i l op =
    MethodInfo
        { name :: fn
        , formalParams :: [i]
        , returns :: [Key]
        }
```

```
    deriving (Show)
```

Listing 5.5. definition of $MethodInfo$.

This $MethodInfo$ can be seen as a datatype that we can increment whenever we feel the need to do it, for now the information that is needed is in 5.5.

The worst part of this way of analysis, is that we have to be aware to change this datatype to all the things we want to analyse.

5.3.3 Construction of DFG

The parser is going to be language-java. And then transformed to the language at figure 5.1, if possible.

A thing can be transformed to a *DFG* if there exists a function that goes from that thing to a DFG. Because the DFG needs to have a unique identifier for each vertex, we need to annotate the FN, I, L and Op with a Key, which in this case is an Int like in the listing 5.6.

To not have an infinite cycle in the analysis, we have the annotation of fn with a $MethodInfo$ seen in 5.6. Since the function definition can be absent, we have a $Maybe$ to represent that behaviour.

```
type MI fn i l op = MethodInfo (Key, fn) (Key, i) (Key, l) (Key, op)

class HasDFG d where
    dfg :: (Ord fn, Ord i, Ord l, Ord op)
        => d (Key, (Maybe (MI fn i l op), fn)) (Key, i) (Key, l) (Key, op)
        -> DFG fn i l op
```

Listing 5.6. HasDFG class.

Now if we want P to have a DFG, P, F, S, M and E needs to be instantiated, like the listing 5.7 and 5.11.

```
instance (HasDFG f, HasDFG m) => HasDFG (P' f m) where
    dfg (P' ma f me) = dfg ma <> df <> dme
      where
        df = mconcat $ map dfg f
        dme = mconcat $ map dfg me

instance (HasDFG e, HasInfimum e) => HasDFG (F' e) where
    dfg (F' (z, i) e) = dfg e <> Gr [(z, NodeIdent i)] [((ze, z), Normal)]
      where
        ze = fst $ infimum e
```

Listing 5.7. Instances of P and F.

The composition of Statements is more elaborated, let us follow an example below of 2 assignments.

```
int x = 2;
int y = x + 3;
```

Now we need to connect the variable x in the first line to the x in the second line. To do that, we need to store the information that the x is an "output" variable, to the first line, and the x is an "input" variable, to the second line. Having those "input" and "output" variables, it is possible to connect them with an edge.

That information can be stored like $(Is, instances)$ where

- $Is \subseteq I$
- $instances : Is \to \mathcal{P}(\mathbb{Z})$, the function that goes from a variable name to a set of instances of that variable.

The function that connects them, called, $connect$ can be defined like the code 5.8.

For example, if the example above has the identifiers as following:

```
int x_0 = 2_1;
int y_2 = x_3 +_4 3_5;
```

the function connect can be applied to the example as:

$$connect\{("x", \{0\})\}\{("x", \{3\})\} = \{((0, 3), Normal)\}$$

```
connect :: Ord i => Map i (Set Int) -> Map i (Set Int) -> Map (Int, Int)
    EdgeL
connect o i =
    M.fromList $ -- Map (Int, Int) EdgeL
    S.toList $ -- [((Int, Int), EdgeL)]
    S.map (\x -> (x, Normal)) $ -- Set ((Int, Int), EdgeL)
    M.foldr S.union [] $ -- Set (Int, Int)
    M.map (uncurry cartesianProduct) $ -- Map i (Set (Int, Int))
    M.intersectionWith (,) o i -- Map i (Set Int, Set Int)
```

Listing 5.8. definition of $connect$.

To collect the "input" and "output" variables, we can define a type class that has the functions $inVars$ and $outVars$.

```
class IOVars a where
    inVars :: Ord i => a fn (Key, i) l op -> Map i (Set Key)
    outVars :: Ord i => a fn (Key, i) l op -> Map i (Set Key)
```

Listing 5.9. class $IOVars$.

We have to instantiate S and E, that can be done like in the listing 5.10.

```
instance IOVars e => IOVars (S' e) where
    inVars (Comp s1 s2) = M.unionWith S.union iv1 iv2
      where
        iv1 = inVars s1
        iv2 = S.foldr M.delete (inVars s2) (keysSet $ outVars s1)
    inVars (IF e s1 s2) = M.unionWith S.union iv1 iv2
      where
        iv1 = inVars $ Comp (S e) s1
        iv2 = inVars $ Comp (S e) s2
    inVars (W e s) = inVars (IF e s NOP)
    inVars (S e) = inVars e
    inVars (R e) = inVars e
    inVars (NOP) = []
    outVars (Comp s1 s2) = M.unionWith S.union iv1 iv2
      where
        iv1 = S.foldr M.delete (outVars s1) (keysSet $ inVars s1)
        iv2 = outVars s2
    outVars (IF e s1 s2) = M.unionWith S.union iv1 iv2
      where
        iv1 = outVars $ Comp (S e) s1
        iv2 = outVars $ Comp (S e) s2
    outVars (W e s) = outVars (IF e s NOP)
    outVars (S e) = outVars e
    outVars (R e) = outVars e
    outVars (NOP) = []

instance IOVars E where
    inVars (Lit l) = []
    inVars (Ident (z, i)) = [(i, [z])]
    inVars (BinOp e1 _ e2) = inVars (Comp (S e1) (S e2))
    inVars (Call _ le) = inVars $ P.foldr (\e s -> Comp (S e) s) NOP le
    inVars (Assign i e) = inVars e
    outVars (Lit _) = []
```

```
    outVars (Ident _) = []
    outVars (BinOp e1 _ e2) = outVars (Comp (S e1) (S e2))
    outVars (Call _ le) = outVars $ P.foldr (\e s -> Comp (S e) s) NOP le
    outVars (Assign (z, i) e) = unionWith const [(i, [z])] $ outVars e
```

Listing 5.10. IOVars instances of S and E.

The rest of the DFG instances, S, M and E, can be implemented like in the listing 5.11.

```
instance (HasDFG e, IOVars e) => HasDFG (S' e) where
    dfg (Comp s1 s2) = dfg s1 <> dfg s2 <> Gr [] es
      where
        es = connect (outVars s1) (inVars s2)
    dfg (IF e s1 s2) = d1 <> d2
      where
        d1 = dfg (Comp (S e) s1)
        d2 = dfg (Comp (S e) s2)
    dfg (W e s) = dfg (IF e (Comp s (IF e s NOP)) NOP)
    dfg (S e) = dfg e
    dfg (R e) = dfg e
    dfg NOP = Gr [] []

instance (HasDFG s, IOVars s) => HasDFG (M' s) where
    dfg (M' _ ps b) = dfg b <> Gr ns es
      where
        ns = fromList [(z, NodeIdent i) | (z, i) <- ps]
        mps = fromList [(i, [z]) | (z, i) <- ps]
        es = connect mps (inVars b)

instance HasDFG E where
    dfg (Lit (z, l)) = Gr [(z, NodeLit l)] []
    dfg (Ident (z, i)) = Gr [(z, NodeIdent i)] []
    dfg (BinOp e1 (z, op) e2)
     =
        dfg e1 <>
        dfg e2 <> Gr [(z, NodeOp op)] [((z1, z), Normal), ((z2, z), Normal)
           ]
      where
        (z1, _) = infimum e1
        (z2, _) = infimum e2
-- if the function is not defined, then let's suppose every input
   influences the output
    dfg (Call (z, (Nothing, fn)) le) = mconcat ld <> Gr [(z, NodeFN fn)] es
```

```
      where
        lz = map (fst . infimum) le
        ld = map dfg le
        es = fromList [((e, z), Normal) | e <- lz]
    dfg (Call (z, (Just (MethodInfo _ ps rs), fn)) le) =
        mconcat ld <> Gr [(z, NodeFN fn)] (es <> rs')
      where
        lz = map (fst . infimum) le
        ld = map dfg le
        es = fromList $ map (\e -> (e, In z)) $ zip lz $ map fst ps
        rs' = fromList [((x, z), Out z) | x <- rs]
    dfg (Assign (z, i) e) = dfg e <> Gr [(z, NodeIdent i)] [((ze, z),
       Normal)]
      where
        ze = fst $ infimum e
```

Listing 5.11. Instances of S, M and E.

When executing the *dfg* described in 5.7, and beautifying the output, we get the results in 3.1.

5.4 SUB-DFG

To create a sub-DFG, we can use the expression below.

$$subDFG = subgraph(DFG, dfs(DFG, sources) \cap rdfs(DFG, sinks))$$

The implementation can be seen below.

```
influencingOn ::
    DFG fn i l op       -- DFG
    -> [Key]            -- sources
    -> [Key]            -- sinks
    -> DFG fn i l op    -- sub-DFG
influencingOn d sources sinks = subgraph d nodesForSub
    where
        nodesForSub = intersect belowSources aboveSinks
        belowSources = dfs d sources
        aboveSinks = rdfs d sinks
```

Listing 5.12. Definition of influencingOn for the sub-DFG.

To compare the simple solution to this one, we can also implement the search of the simple solution as in the section 5.5.

5.5 SEARCH

The detection of a vulnerability with the DFG can be very complex at times. Because of that exists a component named "Query" that can process and query very specific questions, called "Atomic Query", like $InfluencingOn$ that receives sources and sinks, then gives all the possible paths from the sources to the sinks. $InfluencedBy$ that is just the $InfluencingOn$, but with the inputs swapped. It is convenient to talk about sanitizers when analysing a DFG, sanitizers are vertices that if they appear in the middle of the path, means that the path is not a vulnerability. That said, $InfluencingOnAndNotSanitized$ receives sources, sinks and sanitizers, then gives the "composition" of the paths from $InfluencingOn$ of sources and not sanitizers, and $InfluencingOn$ of not sanitizers and sinks.

If we continue to examine each of the "Atomic Query", it can be seen that the most important "Atomic Query" is the $InfluencingOn$. Because of that, we will focus only on this "Atomic Query", and how to make it scalable.

Because the DFG tries to condense the dataflow, by identifying the function body, and only writing it once, a valid path has some restrictions. To demonstrate these restrictions, we can look at the example in the figure 3.1, there we can see that we can enter in the $parseInt\_3$ by going from $args\_4$ to i_13, but when getting out we could get out from i_14 to $parseInt\_3$ or to $parseInt\_6$, but the valid way to get out is only by $parseInt\_3$, because we never entered from the call $parseInt\_6$.

To check if a path in a DFG is a valid path, we can do it by traversing the paths, from the source vertex to the sink vertex, and when we find a $Normal$ we continue traversing, when we find an $In\ i$ we store the i in the stack, when we find the $Out\ i$ we remove the i from the top of the stack, if the i is in the top. This algorithm can be written like in the code 5.13.

```
type FlowL fn i l op = (NodeL fn i l op, [(EdgeL, NodeL fn i l op)])

check :: FlowL fn i l op -> Bool
check f = maybe False (null . fst) $ resState
  where
    edgeSeq = map fst $ snd f
    resState = checkWithState ([], edgeSeq)

checkWithState :: ([Int], [EdgeL]) -> Maybe ([Int], [EdgeL])
checkWithState x@(_, []) = Just x
checkWithState (s, Normal:t) = checkWithState (s, t)
```

```
checkWithState (s, (In i):t) = checkWithState (i : s, t)
checkWithState (i':s, (Out i):t)
    | i == i' = checkWithState (s, t)
checkWithState _ = Nothing
```

Listing 5.13. Algorithm to check Influence.

To search a valid path, we can use the implementation of the simple solution. This search originally occurs in the DFG and not on the sub-DFG, but because the sub-DFG is a subgraph of the DFG it can also be used on the sub-DFG. This implementation also searches for all the paths at once, but can also search on demand, with some tweaks.

The return of $influencingOn$ is the accumulation of the flows that start from a single source, therefore the implementation is the union of the flows from a single source. Here we also use parallel processes to handle each search from a single source, to also show that the bottleneck of this strategy does not depend on the optimizations done, but depends on the data structure itself.

```
influencingOn ::
    DFG fn i l op
    -> [Key]
    -> [Key]
    -> Flows
influencingOn df sources sinks = unionsWith smallestFlow flowOfSources
    where
        flowOfSources = L.map (\s -> influencingOnS df s sinks) sources `
            using` parBuffer 100 rseq
```

Listing 5.14. Definition of influencingOn.

To iterate through the DFG, we need to have the additional information of the flow (fl) we are currently on, and the call stack ($callS$) to have a way to know if we should exit in a certain call or not.

```
influencingOnS ::
    DFG fn i l op
    -> Key
    -> [Key]
    -> Flows
influencingOnS df source sinks =
    influencingOnSWithState df (Flow (source, [])) [] sinks
```

Listing 5.15. Definition of influencingOnS.

The code below can be viewed as a DFS, but it does not track which nodes were visited, instead has a flow which is the path already taken, only when a node does not have a neighbour or already is in the flow, that we stop or backtrack the search. This algorithm does not have the same complexity of the DFS, because it can visit a node or an edge multiple times.

```
influencingOnSWithState ::
    DFG fn i l op
    -> Flow
    -> [Call]
    -> [Key]
    -> Flows
influencingOnSWithState df fl@(Flow (curr, ct)) callS sinks
    | curr `elem` ct = mempty
    | curr `elem` sinks = insertWith smallestFlow (idFlow fl) fl
       neighborFlows
    | otherwise = neighborFlows
    where
        linksNodes = lsuc df curr
        neighborFlows = unionsWith smallestFlow $
            L.map
            (\(n,l) ->
                case l of
                    Normal -> influencingOnSWithState df (Flow (n, curr :
                       ct)) callS sinks
                    In i -> influencingOnSWithState df (Flow (n, curr : ct)
                       ) (i:callS) sinks
                    Out i -> case callS of
                        (j:callS') | i == j -> influencingOnSWithState df (
                            Flow (n, curr : ct)) callS' sinks
                        _ -> mempty
            )
            linksNodes
```

Listing 5.16. Definition of influencingOnSWithState.

With the auxiliary functions implemented below. We give a way to identify a flow by the two nodes that start and end the path, which in this case we only care about the smallest for each path identified by that way, and because of that we also give a way to choose what flow is the smallest.

```
-- first and last node of the path
idFlow :: Flow -> (Key, Key)
idFlow (Flow (h, [])) = (h, h)
idFlow (Flow (h, t)) = (h, last t)
```

```
lengthFlow :: Flow -> Int
lengthFlow (Flow (_,t)) = 1 + length t

smallestFlow :: Flow -> Flow -> Flow
smallestFlow f f'
    | lengthFlow f < lengthFlow f' = f
    | otherwise = f'
```

Listing 5.17. Auxiliary functions for the implementation of influencingOn.

5.6 SUMMARY

We formulated the syntax of a simple imperative made up language, so we can work with it for demonstration purposes. Formulated also the data structure of the DFG, some useful operation on it and how to get that from a source code of our simple imperative language.

With the DFG and an atomic query $InfluencingOn$, we showed how to construct a sub-DFG, this process can be used to create a query language that has more operations, but the $InfluencingOn$ is the main operation that needed attention, because other ones can be constructed with it, or are simple lookup algorithms, unions, or intersections.

In the end we focused on how to search a path for the user to visualize, which is a more involved graph search algorithm than a normal DFS, with a stack for call entrances and exits.

6. BENCHMARKING

6.1 INTRODUCTION

For what I have collected in the actual product, when the solution here was implemented at Checkmarx, to show that this proposal worked, unfortunately performed worse than the solution proposed here.

The two possible causes for this to happen is, that the sources are in parts very disconnected of the DFG, which means that the parallelism had an advantage, because went through more frequently in less repeated edges than very repeated edges. Or the DFG was so big, that the limits on the current implementation were reached, and it got advantages in every query.

But the reader has to remember that this is comparing the simplest solution, which takes only 1 CPU core, against a product that is in production which uses multiple CPU cores and with some cases limited, for performance's sake.

So to have both of the approaches on a common ground, the implementation on the previous chapter was created. For that reason, these implementations are prepared to be compared against each other and gather some data.

6.2 PREPARATION IN THE IMPLEMENTATION

To be able to compare both solutions needs the ability to generate some random programs, but because the queries can perform in any graph, the test of performance of the queries just needs to generate a random graph.

This generation is just the generation of a graph, because the algorithm does not differentiate between call edges and normal edges, which makes call edges not very important in this part of the work, and it can also happen that a program can have this structure, i.e. without calls.

The generation process can just use a generator for graphs of the package *fgl-arbitrary*, and treat the nodes as variables, literals, operators, etc. and the edges like an influence between nodes of the program. To generate the source and sink nodes, we just need to generate two random sets of nodes.

6.3 TIME

After having the simple implementation and the proposed solution of this dissertation, then it just needs to test each in time execution. The following graphs measure the time by the execution of the program and writing the result to a file. This is to emulate writing to a database or to the monitor of the user.

The vertical axis is the average time it takes to execute an $influencingOn$ in a given DFG, sources and sinks. The horizontal axis is the number of edges that a DFG has when given to the atomic query $influencingOn$.

These tests were done in a machine with the following CPU with 8 cores, i7-10510U @ 1.80GHz. It was measured 1 million tests with CPU time for each iteration.

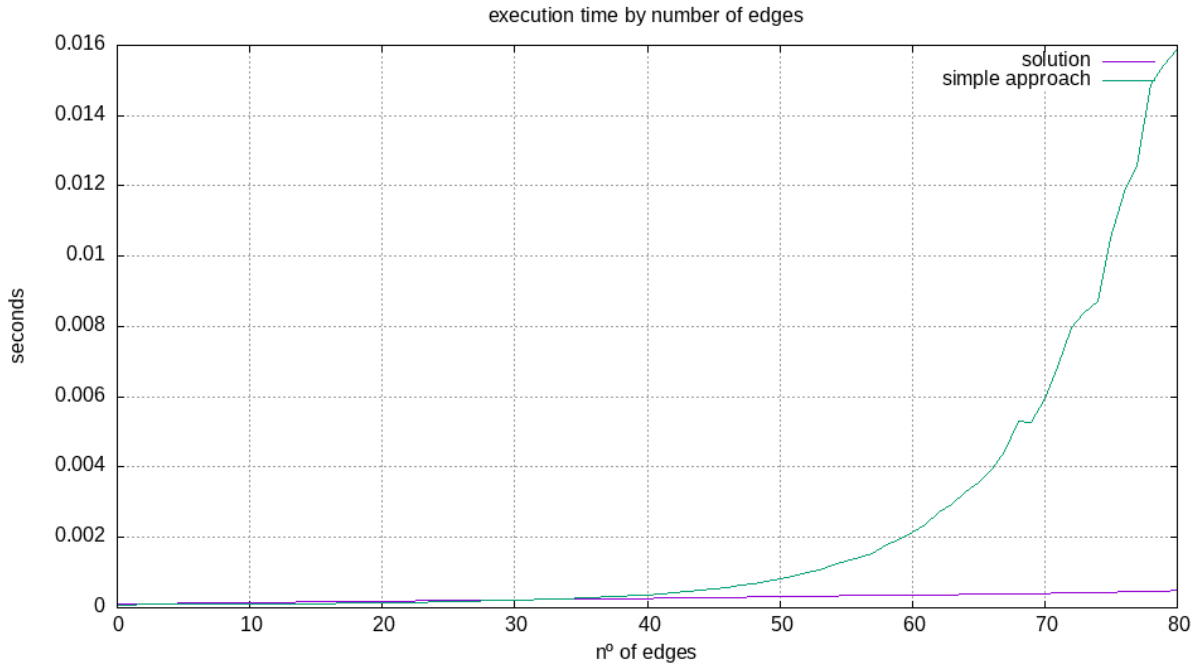


Figure 6.1. The mean of the execution time for each approach, by number of edges of the input DFG.

As it can be seen just for this few values of quantity of edges, the difference is clear and it can verify the theoretical values.

6.4 SUMMARY

This chapter compares both of the implementations by time taken to execute and write it to a file, the simple implementation and the solution presented in this dissertation.

In the end, it is concluded that the solution presented in this dissertation is by far the best option in terms of time complexity.

7. CONCLUSION

The main goal of this work is finding a linear way to detect a vulnerability in a DFG, which the simple implementation discussed in the beginning of this dissertation does not do, in a specific type of SAST tool. The solution presented does this successfully by coming up with a more compact structure that can represent all the key expected outputs. After having this structure, the algorithms needed are just simple graph algorithms. But even then one can see a big difference in performance, also by delaying the hard process to the point where users can control how much information they want.

Dividing the main problem in two parts, (a) finding the subgraph and then (b) returning specific paths, shows better results than just navigating through the same connections from a DFG directly, because the user can control how many paths to return and keep the execution time reasonable, instead of calculating every single path and give all of those paths without any input from the user.

With this linear strategy, it can be said that the growth of the online services can happen with a viable way of detecting vulnerabilities, the source code can also grow in a way that the time grows with it, in the same magnitude as the number of LOC.

The more vulnerabilities identified before the application launch, the less impact the cyberattacks have in the world. The contribution of this dissertation is to help in this regard, by presenting a strategy that can make the vulnerability detection less of a hassle, by reducing the time a tool needs to detect it and present it to the user.

Even though this solution has a better time complexity because it delivers a more compact report of vulnerabilities, it also means that the user will receive a graph instead of a set of vulnerable paths, which is a different perspective of the vulnerabilities. With one approach, it gets one path each time, making that path clearer to understand and easy to share and talk about with other colleagues, and even getting different ways of evaluation from the machine, for example, calculating the number of vulnerable paths, or the number of vulnerable nodes in the code. With the other approach, it gets a sub-DFG, which is hard to understand as a whole, but it gives more information on the vulnerability structure and can also get part or all the paths on the vulnerabilities, but with a price, which is the time it takes. These points emphasize that this solution is not a substitution for the current approach. It only gives a different and faster alternative with limited information if it does not expand the sub-DFG into paths.

7.1 PROSPECT FOR FUTURE WORK

7.1.1 Optimization by dynamically incrementing the DFG

The approach proposed in this dissertation helps programmers to create programs without vulnerabilities. If the programmer is constantly modifying small parts of the code, it is a waste of computational

power and time if all the things analysed are deleted, and it needs to start over again, both on creating the DFG and the sub-DFG for all the queries.

This optimization seems like the simpler one, but it can have a larger impact on the cycle of the implementation of a program. It was even discussed in the beginning of this dissertation as another possible optimization to this process, very briefly.

The incremental part can be adding something to the implementation, e.g. a new function, some lines etc., or deleting something from the implementation. All of those things are mere nodes and edges in the graph, with that, the new DFG can be written like so: $NewDFG = (DFG \cup Inc) \setminus Dec$. The first problem is how can we get Inc and Dec, from the differences in the program files. Then we need to find an efficient transformation T, such that:

$$NewSubDFG = T(subDFG, Inc, Dec)$$

One idea on the efficient transformation part, is to store not only the sub-DFG, but also the result of $dfs(source, DFG)$ and the result of $rdfs(sinks, DFG)$, because an unordered DFS can be computed dynamically given some increments and decrements, efficiently. With the ability to update the $dfs(source, DFG)$ and the $rdfs(sinks, DFG)$ dynamically, we can also update the sub-DFG dynamically.

7.1.2 Optimization with DAG

When we calculate the sub-DFG we calculate all the nodes that are reachable from the sources and all the nodes that can reach the sinks, only then we intersect the results. The results from the dfs or $rdfs$ can produce a lot of wasted computation, especially when the sources and sinks are in a small distance from each other and the DFG has very long paths. Also, when we have cycles, and have sources and sinks inside a strongly connected component of the DFG, the sub-DFG should be obviously all the strongly connected component. This optimization can take care of these 2 problems.

Before any query evaluation we can modify the DFG, linearly, so that it is much more simple to calculate the sub-DFG for each query. That modification needs to be great at calculating the graph reachability problem, and a DAG seems like a good alternative.

To transform a generic directed graph to a directed acyclic graph (DAG), we can use not the graph, but the condensation of the graph, to have the advantages of a DAG, and still not losing any information from the original directed graph. Condensation groups vertices from a strongly connected component to a single vertex, where that vertex has all the information, like the old vertices and its internal connections, also groups the edges, from strongly connected components to other strongly connected components, with

that in the end we do not have a strongly connected component in the condensation, therefore achieving a DAG.

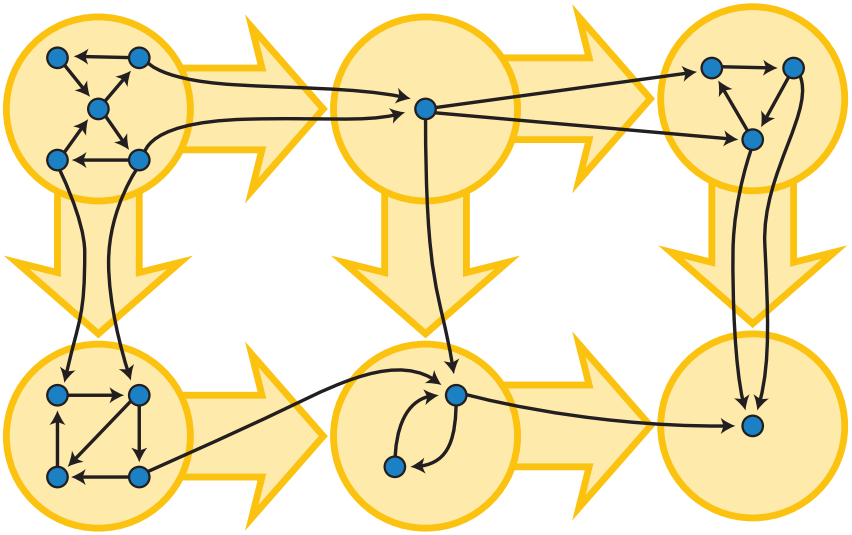


Figure 7.1. The yellow directed acyclic graph is the condensation of the blue directed graph.

This optimization consists in creating the condensation of the DFG, so that we can create a partial order of the nodes, making the process of finding the sub-DFG simpler and more efficient. For example, if 2 nodes are in the same strongly connected component, then they are automatically influencing each other inside the component, filtering out some unnecessary edges. And also, if we have a sink with lots of nodes after it, that are not sinks themselves, or a source with lots of nodes before it, that are not sources themselves, then it is useless to search there for parts of paths.

A recent approach on reachability queries that could be utilized in this situation, by condensing the DFG, can be found in [WYLJ14].

7.1.3 Optimization by introducing parallel algorithms

There are essentially no good parallel algorithms known for the most basic problems on general directed graphs, especially when the graph is sparse [Fin18]. In our case, the DFG is a graph representing the information flow of a program, so we can admit that the graph is not sparse.

Because of how we calculate the sub-DFG, we want the reachable nodes from the source nodes and the nodes that reach the sink nodes. The paper [Fin18] creates a probabilistic work efficient algorithm to solve a simple problem, which in the paper's words is described as:

Perhaps the most basic problem on directed graphs is the single-source reachability problem: given a directed graph $G = (V, E)$ and source vertex $s \in V$, identify the set of vertices reachable by a directed path originating at s.

Even though this problem is for only one source node, we can easily adapt the graph to have one extra node with neighbours as the source nodes only. The paper [Fin18] presents a randomized parallel algorithm for digraph reachability and related problems with expected work $\tilde{\Omega}(m)$ and span $\tilde{\mathcal{O}}(n^{2/3})$, and hence parallelism $\tilde{\Omega}(m/n^{2/3}) = \tilde{\Omega}(n^{1/3})$, on any graph with n vertices and m arcs. This makes it a good candidate for a parallel algorithm to make the calculation of the sub-DFG a bit faster.

7.1.4 Restricting the sub-DFG

It is possible for the sub-DFG to have nodes and edges that do not make part of any valid path, making those nodes and edges irrelevant. One example of this can be thought as a source code with two unrelated parts, where one part has a source node and the other has a sink node. If it has a procedure call, of the same procedure, in both of this unrelated parts, it would exist an invalid path that enters from one call and exits to the other call creating a connection from two unrelated parts of the source code, therefore that connection creates a potential path in the sub-DFG, even though that path is an invalid one.

The big problem here is the relaxation of the call indexes on the construction of the sub-DFG. To make the sub-DFG not relaxed on the call indexes, we need to make sure if we enter a procedure from call i and we cannot get out from the call i, we must influence a sink before the procedure ends.

With the previous property, we can create a search algorithm that for any path taken ends in a sink node. With an hash map to associate a call index to something that indicates if it only has an edge entering the call, or has both edges, entering and exiting the call. And with annotations inside each procedure, that annotates all the nodes that influence only the sink nodes, and not the return nodes. We can have a search that when it enters a procedure, via a call, we can look up the call index (a) if the call only enters, and does not exit the procedure, we can follow only the annotated nodes, with that we can guarantee that we will always end up in a sink node, i.e. we do not need to backtrack to find a path in the sub-DFG (b) if the call enters and exits, we can continue the search normally.

With these modifications, we can have certainty that the search does not need to backtrack, because we do not have the situation of checking if the call index is exiting the procedure. Now we can argue that finding a valid path in the sub-DFG has complexity $\mathcal{O}(l)$, where l is the length of the maximum path in the graph.

With these modifications, if a procedure does not have any annotations, all the call indexes of that procedure that can only enter the call, cannot influence any sink node, therefore it can be removed from the sub-DFG. Leaving the sub-DFG with only relevant nodes and edges.

BIBLIOGRAPHY

[AC76] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[AEK04] R. Al-Ekram and K. Kontogiannis. Source code modularization using lattice of concept slices. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 195–203, 2004.

[Alb19] Marwan Albahar. Cyber attacks and terrorism: A twenty-first century conundrum. *Science and Engineering Ethics*, 25, 08 2019.

[Anl02] Chris Anley. Advanced SQL injection in SQL server applications. 2002.

[Chr20] Gentsch Christoph. Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C. *DLR-Interner Bericht. DLR-IB-DW-JE-2020-16. DLR DW. 37 S.*, 2020.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Coc70] John Cocke. Global Common Subexpression Elimination. *SIGPLAN Not.*, 5(7):20–24, July 1970.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, July 2019.

[EHR+13] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An Empirical Study on the Effectiveness of Security Code Review. In Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato, editors, *Engineering Secure Software and Systems*, pages 197–212, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Fin18] Jeremy T. Fineman. Nearly Work-Efficient Parallel Algorithm for Digraph Reachability. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, page 457–470, New York, NY, USA, 2018. Association for Computing Machinery.

[Fou22] The OWASP® Foundation. OWASP deserialization of untrusted data. `https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data`, 2022. Accessed: January 31, 2023.

[Git21] GitHub. CodeQL. `https://codeql.github.com/`, 2021. Accessed: January 31, 2023.

[KSS17] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2017.

[Met22] Meta Platforms, Inc. Pysa. `https://pyre-check.org/docs/pysa-basics/`, 2022. Accessed: January 31, 2023.

[OMGSC18] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 86–103, Cham, 2018. Springer International Publishing.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[rr98] rain.forest.puppy <rfpuppy@iname.com>. NT Web Technology Vulnerabilities. `https://web.archive.org/web/20140319065810/http://www.phrack.com/issues.html?issue=54&id=8#article`, 1998. Accessed: January 31, 2023.

[Rum77] J. Rumbaugh. A Data Flow Multiprocessor. *IEEE Transactions on Computers*, C-26(2):138–146, 1977.

[Shi07] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, RFC Editor, 8 2007.

[WYLJ14] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability Querying: An Independent Permutation Labeling Approach. *Proc. VLDB Endow.*, 7(12):1191–1202, aug 2014.